WebOTX アプリケーション開発ガイド

WebOTX アプリケーション開発ガイド

バージョン： 7.1

版数： 初版

リリース： 2007 年 7 月

目次

7. CORBA 3
7.2. Object Broker 3
7.2.1. ORB上のプログラム 3
7.2.2. ORBの動作 4
7.2.3. オブジェクトリファレンス 5
7.2.4. プログラム作成の流れ 6
7.2.5. Java言語を用いたアプリケーションの作成と実行 7
7.2.6. Object Broker C++の機能 9
7.2.7. Object Broker Javaの機能 50
7.2.7.1. Object Broker Javaのプロパティ設定 50
7.2.7.2. 呼び出しタイムアウトの設定 52
7.2.7.3. フック 53
7.2.7.4. インタフェースリポジトリ 56
7.2.7.5. Dynamic Invocation Interface (DII) 62
7.2.7.6. Dynamic Skeleton Interface (DSI) 65
7.2.7.7. サーバ処理のスレッドポリシー選択 68
7.2.7.8. 文字コードセット 69
7.2.7.9. コードベース 72
7.2.7.10. クライアント側コネクションの強制切断 75
7.2.7.11. アプレットプロキシ 75
7.2.8. Object Brokerが提供するサービス 77
7.2.9. アプリケーション設計/コーディング時の秘訣 79
7.2.10. CORBA/IIOPによる通信の仕組み 81
7.2.11. Javaマッピング 84
7.2.12. C++マッピング 92
付録. WebOTX Object Broker C++/Java™ 機能差分 124

7.CORBA

本章では、WebOTX Developer の機能を使いこなすための詳細な説明を行います。また、WebOTX が提供する API の利用方法について説明します。

7.2.Object Broker

この節では Object Broker を使った CORBA アプリケーションのプログラミング・開発について説明します。

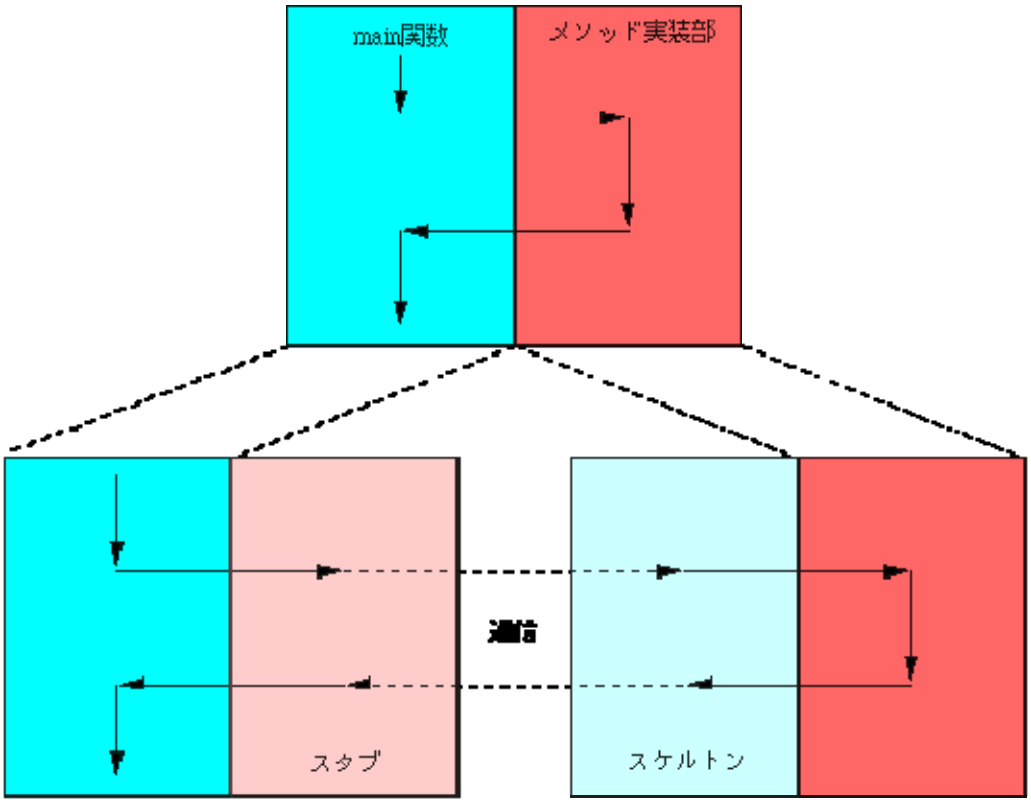
7.2.1.ORB 上のプログラム

ORB 上でのプログラムは、Java や C++の通常のプログラミングとほとんど変わりません。

メソッドを呼び出したとき、通常のアプリケーションでは同一プロセス内で実行されますが、ORB を使用するアプリケーションでは ORB の仲介によりサーバプロセスで実行されます。

ORB のクライアントは、通常のアプリケーションと同様にオブジェクトに対してメソッドを呼び出しますが、この呼び出しに用いられるオブジェクトは、メソッド処理の実装部を持たずに、ORB を介したメッセージの送受信を行って、あたかも同一プロセス内で処理が行われたかのようにふるまいます。この通信に関する部分のプログラムのことをスタブと呼びます。

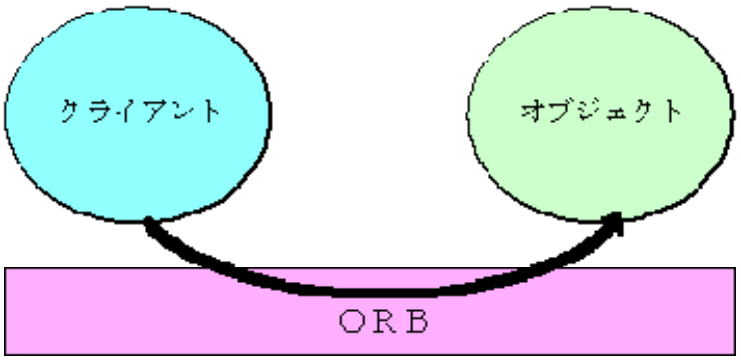
一方、サーバ側にも ORB からメッセージを受け取り、メソッド処理の結果を返すための通信部分を持ちます。この通信部分のプログラムのことをスケルトンといいます。メソッド処理実装部をスケルトンからの派生クラスとして実装することで、メソッド処理実装部とスケルトンを関連付けます。



既存のクライアントプログラムやサーバプログラムを利用する場合を除いて、クライアントプログラム、サーバプログラムの両方を作成する必要があります。

7.2.2.ORB の動作

ORB はクライアントから実行されたサーバオブジェクトへの要求を仲介します。クライアントはサーバオブジェクトのオペレーション呼び出しおよび属性の操作を要求することができます。また、サーバオブジェクトのオペレーションの処理中に他のサーバオブジェクトを呼び出すときは、そのサーバオブジェクトはクライアントにもなります。



本文中、オブジェクトリファレンスを持ち、クライアントから呼び出すことのできるオブジェクトを CORBA オブジェクトと呼びます。また、オブジェクトの実装を提供するプログラムを、サーバプログラムと呼びます。クライアントプログラム内には、サーバプログラム内の CORBA オブジェクトをあらわすための Java/C++ オブジェクト(スタブ)が作成されます。クライアントは、このスタブオブジェクトを通してサーバオブジェクトにアクセスします。

7.2.3.オブジェクトリファレンス

実装オブジェクト

実装オブジェクトとは、IDL で定義したインタフェースを実装したサーバオブジェクトです。サーバントと呼ぶこともあります。

実装オブジェクトは、サーバプロセス中に 1 つだけ存在します。

POA に設定した方針（ポリシー）によって、実装オブジェクトのふるまいが決められます。通常、実装オブジェクトは、クライアントから呼び出される前に生成し、POA と関連づけられます。POA の方針がサーバントマネージャ（org.omg.PortableServer.ServantManager）を利用するように設定されているときは、クライアントから呼び出しを受けた時点で生成することもできます。

オブジェクトリファレンスとは

オブジェクトリファレンスは、実装オブジェクトを参照するオブジェクトで、アクセスに使うプロキシです。実装オブジェクトはサーバプロセス中に 1 つだけ存在しますが、オブジェクトリファレンスは他のプログラム（プロセス）に渡すことができ、複数のコピーを同時に複数のプログラム上で使用することができます。

オブジェクトリファレンスは、org.omg.PortableServer.POA クラスの次のメソッドによって生成されます。

- create_reference
- create_reference_with_id
- servant_to_reference

これらのメソッドからは、1 つのオブジェクトリファレンスが返却されます。Object Broker ではオブジェクトリファレンスの作成は、サーバプロセスをはじめ、クライアントおよびインストーラなど第三者のプログラムでも可能です。ただし、実装オブジェクトに対応付けられたオブジェクトリファレンスは、通常はサーバプロセス上でしか作成することができません。

オブジェクトリファレンスの取得

クライアントからサーバプロセス上の実装オブジェクトを呼び出すためには、このサーバオブジェクトのオブジェクトリファレンスを取得する必要があります。クライアントのプログラミング上はオブジェクトリファレンス＝スタブと考えてもかまいません。

サーバにおいても、クライアントからの要求によって作成されたオブジェクトを戻り値として返すときなどにオブジェクトリファレンスが必要になります。

オブジェクトリファレンスの取得には、次の方法があります。

- 名前サービスを使用する

  サーバでは、名前サービスの bind、rebind を呼び出してオブジェクトリファレンスを登録します。クライアントでは、名前サービスの resolve を呼び出してオブジェクトリファレンスを取得します。また、オブジェクトの URL を指定して org.omg.CORBA.ORB.string_to_object を呼び出すことで名前サービスからオブジェクトリファレンスを取得することもできます。

- 文字列形式から復元する

  オブジェクトリファレンスは、内部構造が ORB の実装に依存するため、サーバとクライアント間でオブジェクトリファレンスをそのままの形式で渡すことはできません。サーバでは、オブジェクトリファレンスを org.omg.CORBA.ORB.object_to_string で文字列形式に変換し、ファイルなどを介してクライアントに受け渡します。クライアントでは、文字列形式のオブジェクトリファレンスを org.omg.CORBA.ORB.string_to_object で復元します。これにより、名前サービスが使用できない場合に通信が可能となります。

7.2.4.プログラム作成の流れ

ORB を利用したアプリケーション作成のながれを簡単に説明します。

1. インタフェース定義を行います。
2. 1.で作成した IDL 定義ファイルを IDL コンパイラでコンパイルします。
3. アプリケーションの処理を記述します。
4. サーバプログラムおよびクライアントプログラムの両者を記述します。
5. Java/C++コンパイラによりコンパイルします。

インタフェース定義は CORBA の仕様で定められているインタフェース定義言語(Interface Definition Language：IDL)で記述します。特定のプログラミング言語（C、C++、Java、SmallTalk など）に依存しない形式でメソッドの引数や戻り値などのインタフェースを記述するための言語です。IDL の言語仕様にしたがって記述されたインタフェースを IDL コンパイラにかけることで、特定のプログラミング言語にマッチしたスタブ/スケルトンを自動生成します。スタブ/スケルトンをリンクすることで比較的簡単にクライアント/サーバアプリケーションを作成できます。

Object Broker Java™ は、Java 言語仕様に基づいたソースコードを出力するコンパイラを提供しています。

Object Broker C++は、C++言語仕様に基づいたソースコードを出力する IDL コンパイラを提供しています。

7.2.5.Java 言語を用いたアプリケーションの作成と実行

開発環境の設定

Javaプログラムを作成するためには、JDKもしくはJava IDE(統合開発環境)が必要になります。WebOTX開発環境を使用することもできます。これらの設定方法については、各製品の説明書を良くお読みください。

CLASSPATH 環境変数

Object Broker Java のアプリケーションを作成するためには、以下のファイルを CLASSPATH 環境変数に登録する必要があります。

- jsocks.jar
- ospiorb50.jar
- ospiname50.jar
- ospievnt50.jar

Windows では、これらのファイルは<WebOTX インストールフォルダ>¥ObjectBroker¥lib に含まれています。

HP-UX、Solaris、Linux の場合は、/opt/ObjectSpinner/lib に含まれています。

PATH 環境変数

Object Broker Java のアプリケーションを作成するためには、以下のディレクトリを PATH 環境変数に登録する必要があります。

- <WebOTX インストールフォルダ>¥ObjectBroker¥bin　(Windows)
- /opt/ObjectSpinner/bin　(HP-UX、Solaris、Linux)

ORB 実装クラスの指定

Object Broker Java のアプリケーションを実行するためには、ORB 実装クラスを指定する必要があります。以下のいずれかの方法で指定してください。

- Java システムプロパティに指定する

  次のように、アプリケーション起動時の Java システムプロパティに指定します。ORB 初期化より前の位置であれば、プログラム中に java.lang.System.setProperty メソッドで記述することもできます。

```
> java -Dorg.omg.CORBA.ORBClass=jp.co.nec.orb.OSPORB
 -Dorg.omg.CORBA.ORBSingletonClass=jp.co.nec.orb.OSPORBSingleton
 <AP のクラス名>
```

- ORB.init メソッドの第 2 パラメータに渡す

  java.util.Properties オブジェクトにプロパティを設定し、ORB.init(String[], java.util.Properties)の第 2 パラメータに渡します。

```
      :
      :
java.util.Properties props = new java.util.Properties();
props.setProperty("org.omg.CORBA.ORBClass", "jp.co.nec.orb.OSPORB");
props.setProperty("org.omg.CORBA.ORBSingletonClass",
   "jp.co.nec.orb.OSPORBSingleton");

ORB orb = ORB.init(args, props);
     :
     :
```

- ユーザのホームディレクトリ(${user.home})の orb.properties に指定する

  次のように記述したファイルをユーザのホームディレクトリに orb.properties という名前で作成します。そのユーザ権限で動作するすべてのアプリケーションで設定が有効になりま

す。アプリケーションを起動するときの設定は必要ありません。

```
org.omg.CORBA.ORBClass=jp.co.nec.orb.OSPORB
org.omg.CORBA.ORBSingletonClass=jp.co.nec.orb.OSPORBSingleton
```

・ Java のホームディレクトリ(${java.home})配下の lib ディレクトリの orb.properties に指定する

次のように記述したファイルを Java のホームディレクトリ配下の lib ディレクトリに orb.properties という名前で作成します。その Java ランタイムで動作するすべてのアプリケーションで設定が有効になります。アプリケーションを起動するときの設定は必要ありません。

```
org.omg.CORBA.ORBClass=jp.co.nec.orb.OSPORB
org.omg.CORBA.ORBSingletonClass=jp.co.nec.orb.OSPORBSingleton
```

7.2.6.Object Broker C++の機能

本項の記述は、Object Broker C++にのみ当てはまります。

オブジェクトリファレンスの取得

クライアントがある CORBA オブジェクトを呼び出すためには、この CORBA オブジェクトのオブジェクトリファレンスを取得する必要があります。クライアントのプログラミング上はオブジェクトリファレンス＝スタブと考えてもかまいません。

一方、サーバにおいても、クライアントからの要求によって作成されたオブジェクトを戻り値として返すときなどにオブジェクトリファレンスは必要になります。

オブジェクトリファレンスの取得方法は 3 通りあります。

(1) オブジェクト呼び出しの戻り値、引数などで取得する。

(1-1) 一般のオブジェクト呼び出しの戻り値、引数で取得する。

ユーザが作成した任意のオブジェクトリファレンスを返すオペレーションを利用し取得する方法です。

(1-2) CORBA::ORB::resolve_initial_references の戻り値として取得する。

CORBA::ORB::resolve_initial_references オペレーションはサービス識別子を引数としてとり、指定したサービスに対する初期オブジェクトリファレンスを戻り値として返します。

calc2 サンプルのクライアントプログラム（エラー処理は省略）

```
// C++
int main (int argc, char **argv) {
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv, "", env);
        CORBA::Object_var obj =
                orb->resolve_initial_references("NameService", env);
        CosNaming::NamingContext_var nmctx =
                CosNaming::NamingContext::_narrow(obj);
        ...
}
```

(1-3) 名前サービスの戻り値として取得する。

名前サービスに登録されたオブジェクトリファレンスを取得するためには CosNaming::NamingContext::resolve オペレーションを使用します。プログラム例は「10.1.3. 名前によるオブジェクトリファレンスの参照」を参照してください。

(2) 文字列形式から復元する。

サーバで作成したオブジェクトリファレンスを文字列に変換してファイルなどに書き出し、クライアントで読み取るときに文字列形式から復元する場合に用います。オブジェクトリファレンスは内部構造が ORB の実装に依存するため、サーバとクライアント間でオブジェクトリファレンスをそのままの形式で渡すことはできません。サーバでは、オブジェクトリファレンスを CORBA::ORB::object_to_string で文字列形式に変換します。クライアントでは、文字列形式のオブジェクトリファレンスを CORBA::ORB::string_to_object で復元します。これにより、CORBA2.0 に準拠した他社の ORB との通信が可能となります。

(3) POA のオペレーションを使用してオブジェクトリファレンスを得る

POA のオペレーションを使用して、あるサーバントに対応するオブジェクトリファレンスを得たり、新たにオブジェクトリファレンスを作成したりすることができます。

他社の ORB との接続は、接続のための初期オブジェクトリファレンスを文字列形式で取得し、以後、そのオブジェクトに対する呼び出しを通して順次必要なオブジェクトリファレンスを取得することで可能になります。初期オブジェクトに関してはプログラムに埋め込む、ファイルで転送する、ORB 以外の通信で転送するなどの方法が考えられます。

CORBA オブジェクトと実装オブジェクト

CORBA オブジェクトは PortableServer::POA::create_reference または PortableServer::POA::create_reference_with_id などの POA クラスのメソッドにより作成されます。作成時に 1 つのオブジェクトリファレンスが返却されます。WebOTX Object Broker では CORBA オブジェクトの作成はサーバプロセスをはじめ、クライアントおよびインストーラなど第三者のプログラムでも可能です。ただし、実装オブジェクトに対応付けられたオブジェクトリファレンスは、通常はサーバプロセス上でしか作成することができません。

オブジェクトリファレンスは、「7. オブジェクトリファレンスの取得」で列挙した方式により他のプログラムに渡すことができます。オブジェクトリファレンス自体は Java または C++オブジェクトですが、これは CORBA オブジェクトの実体ではなく、アクセスに使うプロクシですので、複数のコピーが同時に複数のプログラム上で利用できます。実体となる実装オブジェクト(サーバント)はサーバプロセス中に 1 つだけ存在します。

サーバプロセス上では実装オブジェクト(サーバント)を作成します。

1 つの CORBA オブジェクトに対して 1 つの Java または C++実装オブジェクト(サーバント)が存在します。実装オブジェクト(サーバント)のふるまいは POA の方針に依存します。通常、POA に対応づける実装オブジェクトは、CORBA オブジェクトへの呼び出しが発生する前に作成します。PortableServer::ServantManager を利用する POA のときは、呼び出しが発生した時に作成することもできます。

スケルトンクラスの_this メソッド

WebOTX Object Broker C++ では IDL コンパイラが生成するスケルトンクラスの_this メソッドは、基底クラスである PortableServer::DynamicImplementation クラスの_this メソッドをオーバーライドしています。

スケルトンクラスの_this メソッドは、インタフェース固有の型("インタフェース名"_ptr)でオブジェクトリファレンスを返します。

リクエスト処理中にこのメソッドを呼び出したときは、呼び出し対象のオブジェクトを返します。

リクエスト処理以外(main 関数など)の場所でこのメソッドを呼び出したとき、以下の動作をします。

・このサーバントが活性化していないとき、サーバントのデフォルト POA(_deafult_POA メソッドの戻り値)の ImplicitActivationPolicy が IMPLICIT_ACTIVATION ならばサーバントを活性化します。デフォルト POA の方針値が NO_IMPLICIT_ACTIVATION ならば PortableServer::WrongPolicy 例外が throw されます。

・サーバントを活性化した POA の IdUniquenessPolicy が UNIQUE_ID で、かつ、ServantRetentionPolicy が RETAIN ならばサーバントに対応づけられているオブジェクトを返します。POA の方針がそれ以外の組み合わせならば PortableServer::WrongPolicy 例外が throw されます。

POA の各方針については、リファレンスマニュアル「2.2. POA」を参照してください。

タイムアウトの設定方法

実用的なアプリケーションでは、ネットワークエラーやサーバのクラッシュを検出する必要があります。ネットワーク環境下では、これらの状況の検出はタイムアウトによって行われるのが普通です。WebOTX Object Broker では、メソッドを呼び出してから応答が返る前までの間に、何かしらのエラーが起きたと判断するまでの時間を設定する API を提供します。

タイムアウトの設定方法(C++編)

WebOTX Object Broker C++のタイムアウト設定としては以下のものがあります。

1. スレッド単位でタイムアウトを設定する。
2. スレッド単位のタイムアウトを明示的に設定しないときに使われる初期値をプロセス単位で指定する。
3. また、プロセス単位の設定を省略したときに、すべてのプロセスで一律に使われるシステム単位の設定

をする。

クライアントでのタイムアウト設定

・スレッド単位で設定する方法

スレッド単位の設定は次の関数により行います。

CORBA::ORB::_request_timeout(CORBA::ULong, CORBA::Environment&)

クライアントがサーバメソッドを呼び出し始めてから、応答が返るまでの待ち時間を設定します。設定した時間内に通信処理が完了しないときは、CORBA::NO_RESPONSE の例外が発生します。詳しい説明は関数の説明を参照してください。

・プロセス単位で設定する方法

プロセス単位の設定は次の関数により行います。

CORBA::ORB::_process_request_timeout(CORBA::ULong, CORBA::Environment&)

_request_timeout によりスレッドごとのタイムアウトが設定されていないスレッドで、最初にメソッド呼び出しを行うときに、_process_request_timeout で設定された値がスレッドごとのタイムアウトとして設定されます。以後、_request_timeout で変更しないかぎり、そのスレッドではこの値がスレッドごとのタイムアウト値として使われます。

・すべてのプロセスのデフォルト値の設定方法

Windows 版ではレジストリで、HP-UX, Solaris, Linux 版では/usr/lib/ObjectSpinner/conf/orbconf ファイルなどで指定する方法です。

RequestTimeout

RequestTimeout での設定は、各プロセスで_process_request_timeout が呼ばれていないときに使われるデフォルト値となります。つまり、_process_request_timeout でタイムアウト値が設定されていないプロセスでは、ここで設定した値が_request_timeout のデフォルト値として使われます。

詳しい設定の方法は運用ガイドの「1. 環境設定について」を参照してください。

サーバでのタイムアウト設定

・プロセス単位で設定する方法

プロセス単位の設定は次の関数により行います。

CORBA::ORB::_server_request_timeout(CORBA::ULong, CORBA::Environment&)

クライアントからの要求を受信し始めてから、応答を送信し終わるまでにかかる経過可能時間を設定します。この時間内に受信あるいは送信処理を完了できなかったときは、クライアントにエラーが返ります。また、この時間にはアプリケーションメソッド内での処理を含みます。詳しい説明は関数の説明を参照してください。

・すべてのプロセスのデフォルト値の設定方法

Windows 版ではレジストリで、HP-UX, Solaris, Linux 版では/usr/lib/ObjectSpinner/conf/orbconf ファイルなどで指定する方法です。

ServerRequestTimeout

すべてのサーバプロセスで使われる_server_request_timeout のデフォルト値を設定します。
_server_request_timeout で明示的に設定したプロセス以外ではこの値が使われます。

これらの設定を使って、処理が終了するまで待ち続けるようにするには値に 0 を設定します。

以下にタイムアウト値を設定する例を示します。

```
CORBA::ORB_ptr orb = ...; // CORBA::ORB 疑似オブジェクト用

CORBA::Environment env;

// プロセス単位でタイムアウト値を 1 秒に設定する
orb->__process_request_timeout(1L, env); // クライアント用
orb->__server_request_timeout(1L, env); // サーバ用

// スレッド単位でタイムアウト値を 1 秒に設定する
orb->__request_timeout(1L, env);
```

関数呼び出しを行ってから応答が返るまでの間に、設定した時間が過ぎてしまったときは、CORBA::NO_RESPONSE システム例外が返ります。

タイムアウトの設定には優先順位が付けられています。同時に複数の設定がされていたときは、より優先度の高い設定が有効になります。設定の優先順位については運用ガイドの「1. 環境設定について」および、各関数の説明を参照してください。

フック

WebOTX Object Broker C++ のフックには以下の種類があります。

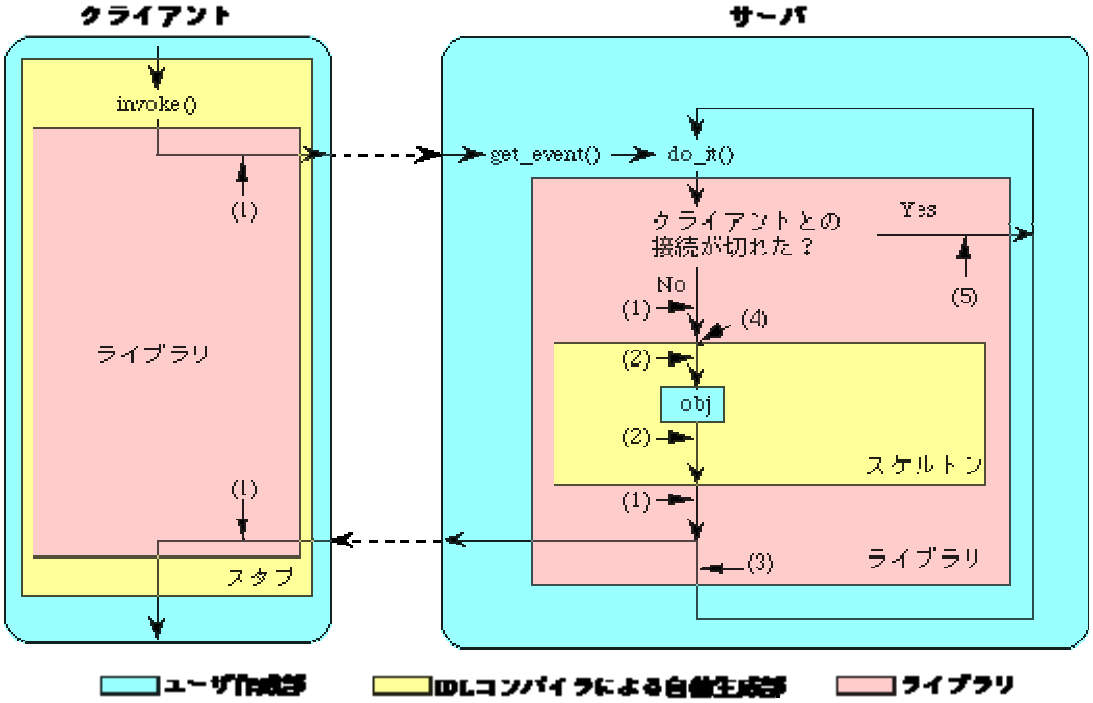
(1)メッセージパケットを操作するフック

(2)メソッドインプリメンテーション呼びだし前後に組み込むフック

(3)サーバのクライアントへのリプライ送信後に組み込むフック

(4)呼び出すオブジェクトをサーバ側で変更するためのフック

(5)クライアントとの接続が切れたときに呼ばれるフック



フックのログ出力:

ロギング機構の設定(運用ガイドの「1. 環境設定について」を参照)でロギングのモードを“INFORMATION”にすることで、ユーザ定義関数の処理に移った/抜けたという情報をログファイルに出力することができます。

メッセージパケットを操作するフック

メッセージパケットを操作するフックは 4 種類あります。

(a) クライアントがサーバにリクエストメッセージを送信する直前のフック

(b) サーバがリクエストメッセージを受け取った直後のフック

(c) サーバがリプライメッセージを送信する直前のフック

(d) クライアントがサーバからのリプライメッセージを受け取った直後のフック

これらを使うと、

メッセージパケットを書き換える

サーバメソッドを呼び出さずにリプライメッセージを作成する

ことができます。

ユーザ定義関数を登録するには、Ob_MesBufHook オブジェクトを作成します。このとき、上記(a)から(d)のどれなのかを示す値をコンストラクタの引数として渡します。設定する値は以下のとおりです。

フックの種類	値
クライアントがサーバにリクエストメッセージを送信する直前のフック	CLNT_PRE_SEND
サーバがリクエストメッセージを受け取った直後のフック	SERV_AFT_RECV
サーバがリプライメッセージを送信する直前のフック	SERV_PRE_SEND
クライアントがサーバからのリプライメッセージを受け取った直後のフック	CLNT_AFT_RECV

つぎにフックオブジェクトに関数を登録します。

(a)と(d)に挿入できるユーザ定義関数インタフェースはつぎのとおりです。

```
void func(Ob_MesBuf&, CORBA::Request_ptr, CORBA::Environment&);
```

(b)と(c)に挿入できるユーザ定義関数インタフェースはつぎのとおりです。

```
void func(Ob_MesBuf&, CORBA::Environment&);
```

フックオブジェクトには複数の関数が登録でき、関数は登録された順番に呼び出されます。また、フックオブジェクトも複数作成することができ、作られた順番にすべてのオブジェクトに登録されている関数が呼び出されます。関数はフックオブジェクトのライフタイムの間だけ登録されています。

ユーザ定義関数の登録のしかたと、(a)と(d)のフックを使った例を次に示します。

(a)のフックを使う場合、IIOP::MessageHeeader と IIOP::ReplyHeader(GIOP1.2 の場合は IIOP::RequestHeader_1_2)各メンバに値を設定する必要があります。

```
// クライアントがリクエストを送信する前に呼び出すフック
void
clnt_pre_send_hook(Ob_MesBuf& mb, CORBA::Request_ptr req, CORBA::Environment& env)
{
        // メッセージバッファの内容を packet.log ファイルに出力する
        mb.dump("packet.log");

        // リクエストメッセージに対するリプライメッセージをクライアントで
        // 作成し、サーバメソッドを呼ばないようにする
        ...

        // リプライメッセージは、Ob_MesBuf& CORBA::Request::__reply()に
        // 書き込む
```

```
        IIOP::MessageHeader ihead;
        ihead.flags = (CORBA::Octet)((Obi_my_byteorder()&1)|(ihead.flags&~1));

        //リプライメッセージを表す
        ihead.message_type = IIOP::Reply;

        //GIOP マイナーバージョン１の場合
        ihead.GIOP_version.minor = 1;

        CORBA::Ulong* message_size_addr = 0;
        if(!(ihead._encode(req->__reply(), message_size_addr))) {
           //error 処理
        }
        if(!(req->__reply().begin_encode_message(message_size_addr))) {
           //error 処理
        }

        IIOP::ReplyHeader rhead;
       //rhead.service_context   ＊必要なら ServiceContext を設定
       rhead.request_id = リクエストメッセージの request_id と同じ値を設定

       //通常のリプライ（エラーなし）の場合
       rhead.reply_status = IIOP::NO_EXCEPTION;

       rhead._encode(req->__reply());

       CORBA::Long nRet;
       ...
       req->__reply() <<= nRet;
       ...
       //戻り値等の encode 終了後に呼ぶ
       req->__reply().end_encode_message();

       //サーバメソッドを呼ばないようにする
       req->__not_call();

}


// クライアントがリプライを受信した後に呼び出すフック
void
clnt_aft_recv_hook(Ob_MesBuf& mb, CORBA::Request_ptr req, CORBA::Environment& env)
{
       // clnt_pre_send_hook で作ったリプライメッセージの内容を
       // packet.log ファイルに出力する
       req->__reply().dump("packet.log");

       // __not_call()にしてあるときは req->__reply()の返り値を
       // デコードするので、req->__reply()を mb にコピーする必要はない
}

main()
{
       ...

       Ob_MesBufHook hook(CLNT_PRE_SEND);
       hook <<= clnt_pre_send_hook;
       Ob_MesBufHook hook2(CLNT_AFT_RECV);
       hook2 <<= clnt_aft_recv_hook;
       ...
```

```
}
```

サーバ側のフックも同様の手順で登録できます。

メソッドインプリメンテーション呼びだし前後に組み込むフック

サーバスケルトンからメソッドを呼び出す前後で、引数やリターン値を書き換えたり、オブジェクトを差し替えたりすることができます。このフックはインタフェースに依存します。

以後の説明では、

"interface"は IDL で定義したインタフェース名

"INTERFACE"は IDL で定義したモジュール名とインタフェース名を大文字にして'_'でつないだもの

"METHOD"は IDL で定義したメソッド名を大文字にしたもの

を表すとします。

ユーザ定義関数を登録するには、まず、インタフェースごとにIDLコンパイラによって自動生成されるフックのオブジェクトを作成します。そのとき、呼び出しの前か後かを示す値をコンストラクタの引数として渡します。引数として渡す値は以下のとおりです。

フックの種類	値
メソッドインプリメンテーション呼びだし前のフック	OB_*"INTERFACE"*_PRE_*"METHOD"*
メソッドインプリメンテーション呼びだし後のフック	OB_*"INTERFACE"*_AFT_*"METHOD"*

つぎにフックオブジェクトに関数を登録します。

ここに登録できるユーザ定義関数インタフェースは次に示す規則で決定されます。

引数	タイプ	意味
第一引数	インタフェース_ptr 型	オブジェクトへのポインタ
第二引数	オペレーションのリターン型	リターン値
それ以降	オペレーションの引数型	引数
最後の引数	CORBA::Environment&	例外値

登録するときは Ob_ProcAddr 型でキャストする必要があります。

フックオブジェクトには複数の関数が登録でき、関数は登録された順番に呼び出されます。また、フックオブジェクトも複数作成することができ、作られた順番にすべてのオブジェクトに登録されている関数が呼び出されます。

関数はフックオブジェクトのライフタイムの間だけ登録されています。

以下に例を示します。

```
/* IDL */
interface MyIntf {
        string myop(in long l);
};


/* C++ */
void hook_func(MyIntf_ptr& o, char* ret, CORBA::ULong l, CORBA::Environment& env)
```

```
{
        ...
}

// OB_MYINTF_PRE_MYOP は myop() を呼び出す前のフックに対する値。
// 呼び出し後は OB_MYINTF_AFT_MYOP。
main()
{
        ...

        Ob_MyIntfHook hook(OB_MYINTF_PRE_MYOP);
        hook <<= (Ob_ProcAddr)hook_func;
        ...

}
```

サーバのクライアントへのリプライ送信後に組み込むフック

特定の実装メソッドが呼ばれたときに、リプライ送信後の操作を追加することができます。リプライ送信後に呼び出したいフックは、実装メソッド内で指定します。

ユーザ定義関数を登録するには、Ob_SentHook オブジェクトを作成するときに関数のアドレスを引数として渡します。

このフックは他のフックとは異なり、関数を 1 つだけ登録することができます。そして、フックオブジェクトのライフタイムに関係なく、登録した関数を一度呼び出すと関数の登録が解除されます。

このフックに登録できるユーザ定義関数インタフェースはつぎのとおりです。

```
void func(CORBA::Environment&);
```

サーバを終了させるメソッドの例を示します。動作の概要は以下のとおりです。

メソッド内でフック関数をセットします。

クライアントにリプライメッセージを送信したあとでフック関数が呼び出されます。

呼び出されたフック関数の中で exit します。

```
// IDL
interface MyIntf {
        ...
        void exit(string mesg);
};


// C++
void server_exit(CORBA::Environment& env)
{
        exit(0);
}

void MyIntf::exit(const char* mesg, CORBA::Environment& env)
{
        ...

        if (strcmp(mesg, "exit") == 0) {
```

```
                Ob_SentHook hook(server_exit);
        }
        ...
}
```

呼び出すオブジェクトをサーバ側で変更するためのフック

サーバを呼び出すためのオブジェクトリファレンスと、オブジェクトインプリメンテーションのオブジェクトリファレンスが異なるときに使います。サーバにリクエストが届いて、リクエストメッセージのうちインタフェースに依存しない部分をデコードした後、オブジェクトインプリメンテーションを見つける前に、オブジェクトリファレンスを差し替えることができます。

このフックはインタフェースに依存しません。

ユーザ定義関数を登録するには、Ob_SvrHook オブジェクトを作成し、作成したフックオブジェクトに関数を登録します。

フックオブジェクトには複数の関数が登録でき、関数は登録された順番に呼び出されます。また、フックオブジェクトも複数作成することができ、作られた順番にすべてのオブジェクトに登録されている関数が呼び出されます。

関数はフックオブジェクトのライフタイムの間だけ登録されています。

このフックに登録できるユーザ定義関数インタフェースはつぎのとおりです。

```
void func(CORBA::Object_ptr&, CORBA::Environment&);
```

登録の仕方を例で示します。

```
void change_object(CORBA::Object_ptr& o, CORBA::Environment& env)
{
        // 新たにオブジェクトをセットする前に、
        // 渡って来たオブジェクトをリリースする
        CORBA::release(o);

        o = ... // オブジェクトをセットする
}

main()
{
        ...

        Ob_SvrHook hook;
        hook <<= change_object;
        ...

}
```

クライアントとの接続が切れたときに呼ばれるフック

サーバアプリケーションがクライアントからのコネクションの切断を検出するときにこのフック関数を使用します。切断時のフック関数は以下の形式で定義します。

```
void myhook(Ob_SvrCon* con, CORBA::Environment&) {
        // ここに切断されたときの処理を記述します。}
```

切断時のフック関数は Ob_SvrSocketClosedHook 型のオブジェクトを作成することで登録されます。Ob_SvrSocketClosedHook のコンストラクタの引数は上記のフック関数へのポインタです。登録は検出したい切断が発生するまでに行っておきます。

main 関数で登録するときの例を示します。

例) main 関数で切断時のフック関数を登録する。

```
main() {
        CORBA::ORB_var orb = CORBA::ORB_init(...);
        ...
```

```
        Ob_SvrSocketClosedHook hobj(myhook);
}
```

次に、クライアントから呼ばれるオペレーション内で、そのオペレーションを呼び出したクライアントが切断されたときのフック関数の設定を行います。フックを呼び出す設定には、Ob_SvrCon::need_closed_hook 関数を使います。この設定を行った場合だけ、クライアントとの接続が切断されたときに、上記フック関数が呼ばれるようになります。need_closed_hook 関数を呼ばない場合あるいは、0を引数として呼び出した場合は、フック関数は呼ばれません。

```
AAA::op1(..., CORBA::Environment& env) {
        Ob_SvrCon* con = env.__connection();

        if (con) {
                // フック関数を呼ぶか呼ばないかのフラグ設定
                con->need_closed_hook(1);
        }
        ...

}
```

インタフェースリポジトリの利用方法

インタフェースリポジトリは CORBA オブジェクトのインタフェース情報を管理します。インタフェースリポジトリにはインタフェース情報の登録、削除、および参照という機能があります。

動的起動インタフェース(DII)を使った呼び出しを行うとき、ターゲットとなるオブジェクトのインタフェース情報が必要になります。このようなとき、インタフェースリポジトリから情報を取り出します。

WebOTX Object Broker では instif, rmif コマンドを使うことでインタフェース情報の登録と削除を簡単に行うことができます。したがって、以降の説明では参照する方法だけを説明します。

リポジトリオブジェクトの取得

インタフェースリポジトリは、リポジトリオブジェクトを頂点とする階層構造をしています。インタフェースリポジトリを利用するには、まず、リポジトリオブジェクトのオブジェクトリファレンスを取得する必要があります。

```
CORBA::ORB_ptr orbobj = CORBA::ORB_init(...);

// リポジトリオブジェクトのオブジェクトリファレンスを得る
CORBA::Object_var ir_in_obj =
        orbobj->resolve_initial_references("InterfaceRepository", env);
```

ここで得られたオブジェクトリファレンスは CORBA::Object 型なのでこのままでは使えません。
CORBA::Repository::_narrow()を使って型を変換します。

```
CORBA::Repository_var repository = CORBA::Repository::_narrow(ir_in_obj);
```

インタフェース情報の取得

インタフェース情報の取得方法には、あるオブジェクトが内包しているオブジェクトの一覧を返すものや、内包しているオブジェクトから手がかりになる文字列を使って検索するものなどさまざまなものがあります。

ここでは、これらのうち CORBA::Container::describe_contents()および
CORBA::InterfaceDef::describe_interface()を使った例を示します。それ以外のオペレーションはリファレンスマニュアルの「インタフェースリポジトリ」を参照してください。

下記の IDL 定義のみが instif コマンドによりインタフェースリポジトリに登録されているものとします。

```
interface myintf {
        void op(in short i);
};
```

このデータをインタフェースリポジトリから取り出す例を以下に示します。なお、例外処理や複数のインタフェースやオペレーションが含まれていた場合の考慮などは省略します。

```
// C++

#include <orb.h>
#include <stdio.h>

// 引数モード別
static char* opemode[] = {
	"in", "out", "inout", 0
};

// 型別
static char* tckind[] = {
	"null", "void", "short", "long", "ushort", "ulong",
	"float", "double", "boolean", "char", "octet", "any",
	"TypeCode", "Principal", "object", "struct", "union",
	"enum", "string", "sequence", "array", "typedef",
	"exception", "longlong", "ulonglong", "longdouble",
	"wchar", "wstring", "fixed", 0
};

void main(int argc, char** argv)
{
	CORBA::Environment env;

	// 初期化
	CORBA::ORB_var orbobj = CORBA::ORB_init(argc, argv, "", env);

	// リポジトリオブジェクトの取り出し
	CORBA::Object_var ir_in_obj =
		orbobj->resolve_initial_references("InterfaceRepository", env);

	CORBA::Repository_var repository =
		CORBA::Repository::_narrow(ir_in_obj);

	// リポジトリオブジェクトから InterfaceDef オブジェクトだけを取り出す
	// CORBA::dk_Interface	: 取り出すのはインタフェースのみ
	// 1			: 継承されたオブジェクトは含まない
	// -1L			: 数量制限なし
	CORBA::Container::DescriptionSeq_ptr descseq =
		repository->describe_contents(CORBA::dk_Interface, 1, -1L, env);

	// シーケンスの最初のオブジェクトを narrow する
	CORBA::InterfaceDef_var intfobj =

	CORBA::InterfaceDef::_narrow((*descseq)[(CORBA::ULong)0].contained_object);

	// インタフェース情報を取り出す
	CORBA::InterfaceDef::FullInterfaceDescription_ptr intfdesc =
		intfobj->describe_interface(env);

	// インタフェース名
	printf("interface %s {¥n", intfdesc->name);

	// オペレーションのシーケンスから 1 つ取り出す
	CORBA::OperationDescription opedesc = intfdesc->operations[(CORBA::ULong)0];

	// オペレーションのモード
	if(opedesc.mode == CORBA::OP_ONEWAY)
		printf("¥toneway ");
	else
		printf("¥t ");
```

```
        // オペレーションの戻り値の型
        printf("%s ", tckind[opedesc.result->kind(env)]);

        // オペレーション名
        printf("%s(", opedesc.name);

        // パラメータのシーケンスから１つ取り出す
        CORBA::ParameterDescription paradesc = opedesc.parameters[(CORBA::ULong)0];

        // オペレーションの引数のモード
        printf("%s ", opemode[paradesc.mode]);

        // オペレーションの引数
        printf("%s %s);¥n", tckind[paradesc.type->kind(env)], paradesc.name);
        printf("};¥n");
    }
```

CORBA::Container::describe_contents()は、そのオブジェクトが内包しているオブジェクトの一覧を CORBA::Container::DescriptionSeq へのポインタで返します。上記の例では InterfaceDef に限定して検索をしています。

CORBA::Container::DescriptionSeq は CORBA::Container::Description 構造体のシーケンスです。CORBA::Container::Description 構造体の定義は以下のとおりです。

```
    struct CORBA::Container::Description {
        CORBA::Contained contained_object; // 内包されたオブジェクト
        CORBA::DefinitionKind kind;    // 内包されたオブジェクトの種類
        CORBA::Any value;              // 内包されたオブジェクトが持つ
                                               // 型情報
    };
```

contained_object には InterfaceDef オブジェクトが入っています。そこで、CORBA::InterfaceDef::describe_interface()を使って、その CORBA::InterfaceDef オブジェクトが内包しているオペレーションのリストを取り出します。オペレーションのリストは CORBA::InterfaceDef::FullInterfaceDescription へのポインタとして返されます。

CORBA::InterfaceDef::FullInterfaceDescription は構造体です。下記にその定義を示します。

```
    struct CORBA::InterfaceDef::FullInterfaceDescription {
        Obi_String name;                    // インタフェース名
        Obi_String id;                              // リポジトリ ID
        Obi_String defined_in;                      // インタフェースを包含してい
    る
                                                    // Container のリポジトリ ID
        Obi_String version;                         // バージョン
        CORBA::OpDescriptionSeq operations;     // オペレーションのシーケンス
        CORBA::AttrDescriptionSeq attributes;   // 属性のシーケンス
        CORBA::RepositoryIdSeq base_interfaces; // 基底インタフェースの
                                                    // シーケンス
        CORBA::TypeCode type;                       // タイプコード
    };
```

さらに、operations から目的のオペレーションの型情報を取り出します。

operations は CORBA::OperationDescription 構造体のシーケンスです。CORBA::OperationDescription 構造体の定義を以下に示します。

```
    struct CORBA::OperationDescription {
        Obi_String name;                // オペレーション名
        Obi_String id;                          // リポジトリ ID
        Obi_String defined_in;                  // オペレーションを包含している
                                                // Container のリポジトリ ID
        Obi_String version;                     // バージョン
```

```
        CORBA::TypeCode result;                    // 戻り値の型を表すタイプコード
        CORBA::OperationMode mode;       // オペレーションのモード
                                                   // CORBA::OP_NORMAL:双方向呼び出し
                                                   // CORBA::OP_ONEWAY:単方向呼び出し
        CORBA::ContextIdSeq context;               // コンテキスト名のシーケンス
        CORBA::ParDescriptionSeq parameters; // パラメータのシーケンス
        CORBA::ExcDescriptionSeq exceptions; // ユーザ定義例外のシーケンス
```

parameters からパラメータ情報を取り出します。

parameters は CORBA::ParameterDescription 構造体のシーケンスです。CORBA::ParameterDescription 構造体の定義を以下に示します。

```
  struct CORBA::ParameterDescription {
        CORBA::Identifier name;        // 引数の名前
        CORBA::TypeCode type;          // 引数の型を表すタイプコード
        CORBA::IDLType type_def;   // 引数の型を表すオブジェクト
        CORBA::ParameterMode mode; // 引数のモード
                                       // CORBA::PARAM_IN    :in 引数
                                       // CORBA::PARAM_OUT   :out 引数
                                       // CORBA::PARAM_INOUT:inout 引数
  };
```

CORBA::OperationDescription 構造体のメンバ exceptions がヌル・ポインタでなければ、exceptions からユーザ定義例外情報を取り出します。

exceptions は CORBA::ExceptionDescription 構造体のシーケンスです。CORBA::ExceptionDescription 構造体の定義を以下に示します。

```
  struct CORBA::ExceptionDescription {
        CORBA::Identifier name;              // 例外名
        CORBA::RepositoryId id;              // リポジトリ ID
        CORBA::RepositoryId defined_in;      // この例外を返すオペレーションの
                                             // リポジトリ ID
        CORBA::VersionSpec version; // バージョン
        CORBA::TypeCode type;                // この例外を示すタイプコード
  };
```

これらのインタフェース情報により、DII による呼び出しに必要なパラメータを組み立てることができます。

Dynamic Invocation Interface (DII)

IDL コンパイラが生成したスタブを使うときには DII を意識する必要はありません。ここでは、直接 DII を使う方法を説明します。

DII を使うと、静的なオペレーション情報がクライアント側にリンクされていなくても、オペレーションを動的に呼び出すことができます。DII はアプリケーション作成時に呼び出すインタフェースが決まっていないような特殊なアプリケーションを作成する場合にだけ使います。このようなアプリケーションの例としては、インタフェースの情報を動的に与えてオブジェクトのテストする汎用のテストツールなどが考えられます。

基本的な呼び出し

DII を使って呼び出す手順は複数あります。大別すると、引数リストなどを先に作っておいて CORBA::Request オブジェクトを作成する方法と CORBA::Request オブジェクトを先に作っておく方法に別れます。

先に引数リストを作成するには CORBA::ORB::()か CORBA::ORB::create_operation_list()を使います。前者は CORBA::NVList::add()などを使って引数を 1 つずつ追加していかなければいけません。後者はインタフェースリポジトリに登録されているインタフェース情報をもとに自動的に追加されます。

CORBA::Request オブジェクトを作成するには、CORBA::Object::_request()または CORBA::Object::_create_request()を使います。CORBA::Object::_create_request()は引数リストにヌル・ポインタを設定しておいて、後から引数を追加することもできます。

作成した CORBA::Request に引数を追加するには、CORBA::Request::arguments()で引数リストを取り出す方法と、CORBA::Request::add_in_arg()などを使う方法があります。以下の例では後者を使います。引数リストを直接操作する方法は「引数リストを表すインターフェース」を参照してください。

以下ではもっとも基本的な呼び出し手順を示します。

DII によるオペレーションの呼び出しを行うには、CORBA::Request オブジェクトにオペレーションの情報(引数や戻り値の型など)をセットする必要があります。

CORBA::Request オブジェクトを作成するには、呼び出すオブジェクトのオブジェクトリファレンスが必要です。オブジェクトリファレンスの取得方法は静的な呼び出しと同じです。不特定のオブジェクトを呼び出すクライアントを作成する場合、名前サービスの CosNaming::Naming_Context::resolve()を使うのが一般的です。

また、不特定のオブジェクトを呼び出すクライアントを作成する場合は、インタフェースの定義を取得する必要があります。インタフェース定義を取得するには、CORBA::Object::_get_interface()を使います。

通常、CORBA::Object::_get_interface()は静的にリンクされているインタフェースを返します。不特定のオブジェクトを呼び出すクライアントを作成するときは、あらかじめ CORBA::ORB::_use_IR()によってインタフェースリポジトリの参照を許可しておきます。これにより、CORBA::Object::_get_interface()はあらかじめインタフェースリポジトリに登録されていたインタフェースを返すようになります。CORBA::ORB::_use_IR()は ObjectSpinner 独自のメソッドなので、他社の ORB に移植するときは注意が必要です。

```
CORBA::Environment env;

...

CORBA::Object_var obj = ...; // 何らかの手段でオブジェクトリファレンスを取得

// インタフェースリポジトリを使った検索を許可する
orb->__use_IR(1, env);

// インタフェースを定義しているオブジェクトの取得
CORBA::InterfaceDef_var intfobj = obj->_get_interface(env);

// インタフェース情報を取り出す
CORBA::InterfaceDef::FullInterfaceDescription_var intfdesc =
       intfobj->describe_interface(env);
```

intfdesc にはオブジェクトリファレンスが指し示すオブジェクトのすべてのインタフェース情報が入っています。この中から、目的のオペレーションを選びます。

```
CORBA::ULong count = intfdesc->operations.length();
CORBA::ULong i;
CORBA::OperationDescription opedesc;

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

       // オペレーションのシーケンスから 1 つ取り出す
       opedesc = intfdesc->operations[i];

       // オペレーション名が一致するものを探す(仮に"myop"を探すことにします)
       if(strcmp(opedesc.name, "myop") == 0)
                break;
}
```

名前が一致するオペレーションを取得できたら、CORBA::Requestオブジェクトを作成します。引数には、オペレーション名を渡します。

```
CORBA::Request_var req = obj->_request(opedesc.name, env);
```

作成した CORBA::Request オブジェクトに戻り値の型を CORBA::TypeCode_ptr 型で指定します。

```
req->set_return_type(opedesc.result, env);
```

オペレーションの引数を登録します。引数には in, out, inout の 3 種類があり、それぞれ別の関数が用意されています。引数は add_XXX_arg()(XXX は引数の種類によって異なる)を呼び出すたびに引数リストの後ろへ追加されます。add_XXX_arg()関数は、CORBA::Any への参照を返します。そして、返されたCORBA::Anyに引数を挿入します。各型の any への挿入の方法は、「C++マッピング」の「any」を参照してください

```
count = opedesc.parameters.length();

CORBA::ParameterDescription paradesc;
CORBA::Any value;

for(i=0; i<count; i++) {
        // パラメータのシーケンスから１つ取り出す
        paradesc = opedesc.parameters[i];

        // パラメータの設定
        switch(paradesc.mode){
        case CORBA::PARAM_IN:
                  switch(paradesc.type->kind()){
                  case CORBA::tk_short:
                            CORBA::Short in_param = ...;
                            req->add_in_arg(paradesc.name, env) <<= in_param;
                            break;
                  ...
                  }
                  break;
        case CORBA::PARAM_INOUT:
                  switch(paradesc.type->kind()){
                  case CORBA::tk_short:
                            CORBA::Short inout_param = ...;
                            req->add_inout_arg(paradesc.name, env)
                                                      <<= inout_param;
                            break;
                  ...
                  }
                  break;
        default:
                  switch(paradesc.type->kind()){
                  case CORBA::tk_short:
                            CORBA::Short out_param = 0; // dummy
                            req->add_out_arg(paradesc.name, env) <<= out_param;
                            break;
                  ...
                  }
                  break;
        }
}
```

引数の組み立てが終わったら、サーバの呼び出しを行います。サーバの呼び出し方には単方向のものと双方向の 2 種類があります。これらは IDL 定義をしたときに決まっています。呼び出しをするときには、以下のようにこれらの違いを調べ、呼び出し方にあった方法を使う必要があります。

```
if(opedesc.mode == CORBA::OP_ONEWAY)
        req->send_oneway(env);       // 単方向呼び出しのとき
else{
        CORBA::Status status = req->invoke(env);       // 双方向呼び出しのとき
}
```

双方向呼び出しで、かつ、戻り値を持っていた場合は、戻り値を取得する必要があります。戻り値は、CORBA::Any 型で取得します。any から各型へ取り出す方法は、「C++マッピング」の「any」を参照してください。

```
if(opedesc.result->kind() != CORBA::tk_void){

        switch(opedesc.result->kind()) {
        case CORBA::tk_short:
                  CORBA::Short result;
                  req->return_value(env) >>= result;
                  break;
```

```
            ...
            }
    }
```

リクエストの発行

- 同期オペレーション

応答を要求する呼び出しです。サーバへリクエストを送信し、応答が返るまで待ちます。

- oneway オペレーション

単方向の呼び出しです。サーバへリクエストを送信するだけで、応答を期待しません。IDL 定義で oneway としたオペレーションの呼び出しに使います。

- 遅延同期オペレーション

送信後、応答が返るのを待たずにクライアント側へ処理が戻ります。応答の取得は任意のタイミングでポーリングまたは待ち合わせをします。

- 複数リクエストの発行

複数のリクエストを同時に発行する方法です。oneway かどうかで使用する関数が異なります。

同期オペレーション

「基本的な呼び出し」の例のとおり、CORBA::Request::invoke()を使います。CORBA::Request::invoke()はサーバにリクエストを送信し、応答が返るまで待ちます。oneway のオペレーションには使用できません。

oneway オペレーション

「基本的な呼び出し」の例に示したとおり、CORBA::Request::send_oneway()を使います。CORBA::Request::send_oneway()はサーバにリクエストを送信するだけで応答を期待しません。したがって、応答を必要とするメソッドの呼び出しには利用できません。メソッドの呼び出し方針が oneway かどうかは IDL 定義をした時点で決まってしまいます。

遅延同期オペレーション

遅延同期オペレーションは、送信と受信を分ける方法です。遅延同期オペレーションでは、CORBA::Request::send_deferred()による送信オペレーションの発行後すぐに制御が戻ります。そして、任意の時点で、CORBA::Request::poll_response()を呼び出してポーリングするか CORBA::Request::get_response()を呼び出して待ち合わせをします。CORBA::Request::poll_response()は単に応答メッセージが受信可能かどうかを調べるだけです。応答メッセージを取得するには CORBA::Request::get_response()を呼び出す必要があります。

複数リクエストの発行

リクエストのシーケンスを作成しておき、複数のリクエストを順番に発行することができます。サーバからの応答を必要とする呼び出しの場合には、CORBA::ORB::send_multiple_requests_deferred を使います。oneway の呼び出しの場合は ORBA::ORB::send_multiple_requests_oneway()を使います。

個々のリクエストについて応答メッセージを調べるには遅延同期オペレーションのときと同様に CORBA::Request::poll_response()と CORBA::Request::get_response()を使います。また、複数のメッセージのうちのどれか 1 つ以上という場合は CORBA::ORB::poll_next_response()と CORBA::ORB::get_next_response()を使います。

Dynamic Skeleton Interface (DSI)

IDL コンパイラが生成したスケルトンを使うときには DSI を意識する必要はありません。ここでは、直接 DSI を使う方法を説明します。

DSI を使うと、静的なオペレーション情報がサーバ側にリンクされていなくとも、オペレーションを動的に呼び出すことができます。ちょうどクライアント側で使われる DII(Dynamic Invocation Interface)の逆向きの機能を実現しています。

DII は CORBA::Request オブジェクトにオペレーションの情報(引数や戻り型など)をセットし、CORBA::Request::invoke 関数を呼び出すことにより、オブジェクト呼び出しの入口を共通化しています。

DSI も、PortableServer::DynamicImplementation::invoke 関数を共通の入口としています。ただし、オペレーションの情報は CORBA::ServerRequest オブジェクトに入っているということが DII とは異なります。

DSI に必要なクラス

DSI を利用するためには PortableServer::DynamicImplementation クラスを継承したクラスを作成します。このクラスがリクエストの処理を実装する DSI サーバント(DSI の実装オブジェクト)になります。DSI サーバントは、つぎの各メンバ関数をオーバライドする必要があります。

```
namespace PortableServer {
	class DynamicImplementation : public virtual PortableServer::ServantBase {
		// 実装オブジェクトのオペレーションを呼び出す
		virtual void invoke(CORBA::ServerRequest_ptr, CORBA::Environment&);

		virtual CORBA::RepositoryId _primary_interface(
			const PortableServer::ObjectId&,
			PortableServer::POA_ptr,
			CORBA::Environment&
		);
	};
};
```

PortableServer::DynamicImplementation の派生クラスの実装

このクラスでは、各オペレーションのほかに、invoke 関数と_primary_interface 関数の実装が必要です。次の例を使って説明します。

```
// IDL

interface MyInterface {
	MyType2 MyOperation(in MyType arg);
};

// C++

class MyDynamicImpl : public PortableServer::DynamicImplementation {
public:
	MyDynamicImpl() { // 必要ならばメンバの領域の割り当て、初期化を行います }
	~MyDynamicImpl() { // 必要ならばメンバの領域の解放を行います }

	virtual void invoke(
		CORBA::ServerRequest_ptr,
		CORBA::Environment&
	);

	virtual CORBA::RepositoryId _primary_interface(
		const PortableServer::ObjectId&,
		PortableServer::POA_ptr,
		CORBA::Environment&
	);

	MyType2 MyOperation(MyType&, CORBA::Environment&); // 呼び出されるオペレーシ
ョン
};
```

invoke 関数

invoke 関数は、オペレーションが呼び出されるとき、ランタイムライブラリから呼ばれる、ユーザコードへの入口となります。invoke 関数の処理は自由に記述することができます。

オペレーションの情報は sr から取り出すことができます。呼び出しが行われたオブジェクトのオブジェクトリファレンスは、PortableServer::DynamicImplementation::_this メソッドを使って取得できます。

env には、invoke 関数内で起きた例外をセットして呼び出し側に通知します。(オプション)

ここでは、実際に C++オブジェクトのメソッドを呼び出すことを目的とした invoke 関数の実装例を説明します。ただし、エラー処理は省略してあります。

```
// IDL

interface MyInterface {
        MyType2 MyOperation(in MyType arg);
};

// C++

void
MyDynamicImpl::invoke(CORBA::ServerRequest_ptr sr, CORBA::Environment&) {
        try {
                // どのオペレーションが呼ばれたか調べる
                const char* op_name = sr->operation();

                if (strcmp(op_name, "MyOperation") == 0) {
                        MyType arg;
                        MyType2 ret;

                        // 引数のデコードに必要な情報をセットする
                        CORBA::NVList_ptr param;

                        CORBA::NamedValue_ptr np = param->add(CORBA::ARG_IN);
                        CORBA::Any* ap = np->value();
                        *ap <<= arg;

                        // 引数をデコードする
                        sr->arguments(param);


                        // IN 引数を取り出す
                        arg = *(MyType*)param->item(0)->value()->value();

                        // オペレーションを呼び出す
                        ret = MyOperation(arg);

                        // 戻り値を sr にセットする
                        CORBA::Any* result = new CORBA::Any;
                        *result <<= ret;
                        sr->set_result(*result);
                }
        } catch (CORBA::NO_MEMORY& e) { // CORBA::NO_MEMORY 例外のとき
                // エラー処理または例外を sr にセットする
                // CORBA::NO_MEMORY 例外を sr にセットする場合
                CORBA::Any* exc = new CORBA::Any(CORBA::_tc_NO_MEMORY, new
CORBA::NO_MEMORY(e), 1);

                sr->set_exception(*exc);
        } catch (CORBA::Exception& e) { // その他の例外のとき
                // エラー処理または例外を sr にセットする
        }
}
```

次に、DSI サーバントのインスタンスが複数あり、インスタンスにより処理を切り換える場合の例を以下に示します。

```
// IDL

interface MyInterface {
```

```
        MyType2 MyOperation(in MyType arg);
};

// C++

class MyDynamicImpl : public PortableServer::DynamicImplementation {
public:
        ...

        virtual void invoke(
                CORBA::ServerRequest_ptr,
                CORBA::Environment&
        );

        MyType2 MyOperation1(MyType&, CORBA::Environment&); // 呼び出されるオペレーション

        MyType2 MyOperation2(MyType&, CORBA::Environment&); // 呼び出されるオペレーション

        ...

};

main(...) {
        CORBA::ORB_ptr orb = CORBA::ORB_init(...);
        PortableServer::POA_ptr rootPOA = ... // ルート POA を取得

        CORBA::PolicyList plist;

        plist.length(1);

        plist[0] = rootPOA->create_id_assignment_policy(PortableServer::USER_ID);

        PortableServer::POA_ptr poa = rootPOA->create_POA("myPOA",
PortableServer::POAManager::_nil(), plist);

        // 文字列をオブジェクト ID に変換
        PortableServer::ObjectId_var oid1 =
                PortableServer::string_to_ObjectId("myimpl1");
        PortableServer::ObjectId_var oid2 =
                PortableServer::string_to_ObjectId("myimpl2");

        MyDynamicImpl* impl1 = ...  // MyOperation1 を呼び出すインスタンスを作成
        MyDynamicImpl* impl2 = ...  // MyOperation2 を呼び出すインスタンスを作成

        // サーバントを活性化
        poa->activate_object_with_id(oid1, impl1);
        poa->activate_object_with_id(oid2, impl2);

        ...

}

void
MyDynamicImpl::invoke(CORBA::ServerRequest_ptr sr, CORBA::Environment&) {
        try {
                CORBA::ORB_var orb = CORBA::ORB_init(...);

                CORBA::Object_var curobj =
orb->resolve_initial_references("POACurrent");
```

```
                PortableServer::Current_var cur =
PortableServer::Current::_narrow(curobj);

                CORBA::Object_var obj = _this();

                PortableServer::POA_var poa = cur->get_POA();

                PortableServer::ObjectId_var oid = poa->reference_to_id(obj);
                CORBA::String_var impl_id =
ProtableServer::ObjectId_to_string(*oid);

                // どのオペレーションが呼ばれたか調べる
                const char* op_name = sr->operation();

                if (strcmp(op_name, "MyOperation") == 0) {
                        MyType arg;
                        MyType2 ret;

                        // 引数のデコードに必要な情報をセットする
                        CORBA::NVList_ptr param;
                        CORBA::NamedValue_ptr np = param->add(CORBA::ARG_IN);
                        CORBA::Any* ap = np->value();
                        *ap <<= arg;

                        // 引数をデコードする
                        sr->arguments(param);

                        // IN 引数を取り出す
                        arg = *(MyType*)param->item(0)->value()->value();

                        // impl_id により呼び出すオペレーションを切り換える
                        if (strcmp(impl_id, "myimpl1") == 0) {
                                ret = MyOperation1(arg);
                        } else {
                                ret = MyOperation2(arg);
                        }

                        // 戻り値を sr にセットする
                        CORBA::Any* result = new CORBA::Any;
                        *result <<= ret;
                        sr->set_result(*result);
                }
        } catch (CORBA::Exception& e) { // その他の例外のとき
                // エラー処理または例外を sr にセットする
        }
}
```

_primary_interface 関数

_primary_interface 関数は、DSI サーバントに対応づけられたオブジェクトの呼び出し時に、DSI サーバントのあつかうインタフェースを取り出すために ORB から呼ばれます。引数には、呼び出しのきっかけとなったオブジェクトのオブジェクト ID とそのオブジェクトに対応づけられた POA が渡されます。DSI サーバントが複数のインタフェースをあつかう場合、引数により適切なインタフェースを返すように記述します。

ここでは、複数のインタフェースをあつかう DSI サーバントの_primary_interface 関数の実装例を説明します。ただし、エラー処理は省略してあります。

```
// IDL

interface MyInterface {
        MyType2 MyOperation(in MyType arg);
};
```

```
interface MyInterface2 {
        MyType2 MyOperation2(in MyType arg);
};

// C++

main(...) {
        CORBA::ORB_ptr orb = CORBA::ORB_init(...);
        PortableServer::POA_ptr rootPOA = ... // ルート POA を取得

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

        plist[0] = rootPOA->create_id_assignment_policy(PortableServer::USER_ID);

        PortableServer::POA_ptr poa =
                rootPOA->create_POA("myPOA", PortableServer::POAManager::_nil(),
plist);

        PortableServer::ObjectId_var oid1 =
                PortableServer::string_to_ObjectId("myintf1");
        PortableServer::ObjectId_var oid2 =
                PortableServer::string_to_ObjectId("myintf2");

        MyDynamicImpl* impl1 = ...  // MyInterface をあつかうインスタンスを作成
        MyDynamicImpl* impl2 = ...  // MyInterface2 をあつかうインスタンスを作成

        // サーバントを活性化
        poa->activate_object_with_id(oid1, impl1);
        poa->activate_object_with_id(oid2, impl2);

        ...

}

CORBA::RepositoryId MyDynamicImpl::_primary_interface(const
PortableServer::ObjectId& oid, PortableServer::POA_ptr poa, CORBA::Environment&) {
        // オブジェクト ID を文字列に変換
        CORBA::String_var str = PortableServer::ObjectId_to_string(oid);

        // 呼び出されたオブジェクトに対応したサーバントを調べる
        if (strcmp(str, "intf1") == 0) { // サーバントが impl1 のとき
            return CORBA::string_dup("IDL:MyInterface:1.0");
        } else { // それ以外のとき
            return CORBA::string_dup("IDL:MyInterface2:1.0");
        }
}
```

WebOTX Object Broker C++独自の注意事項

名前付け規則

大文字小文字を問わず、Ob および Obi で始まるシンボルは WebOTX Object Broker が使用しますので、アプリケーションプログラムではこれらのシンボルを定義しないようにしてください。

CORBA 仕様のクラスに WebOTX Object Broker 独自のメンバを追加する場合メンバ名は'_'(2 つの'_')で始まっています。

すなわち、Ob および Obi で始まるクラス/関数および'_'で始まるメンバ関数は WebOTX Object Broker 独自ですので、これらを使った部分に関しては移植性がありません。なお、Obi は WebOTX Object Broker 実装上内部的に使用するクラス/関数に使用していますので、これらの仕様に関しては断りなく変更されることがあ

ります。したがって、WebOTX Object Broker の将来の版でも利用できるとは限りません。Obi で始まるクラス/関数はアプリケーションプログラム内でご使用になりませんようにお願いします。

CORBA2.2 C++マッピングのベンダ選択部

CORBA では C++コンパイラの仕様の違いを考慮し、ORB ベンダに対していくつかの選択肢を設けています。WebOTX Object Broker では次の選択をしています。

例外マッピング:

例外のマッピングとしては(1)C++の例外を使う方法、(2)例外用の引数を使う方法が選択可能となっています。

WebOTX Object Broker では、C++の例外を使う方法を採用しています。

モジュールマッピング:

モジュールのマッピングとしては、(1)C++の namespace を使う方法、(2)入れ子クラスを使う方法、(3)'_'による名前の連結とが選択可能です。

WebOTX Object Broker では C++の namespace を使う方法を採用しています。

クライアントでのソケット管理

WebOTX Object Broker の既定値の設定では同一クライアントプロセスから同一サーバプロセス間では 1 つのコネクションしか使いません。クライアントプログラムが複数のスレッドで動作しているときで、複数のスレッドが同一サーバプロセスに対し同時に要求を出した場合、その 1 つのコネクションを各スレッドで共有して利用します。

共有を行いたくない場合は、

```
orbobj->__reuse_socket(0, env);
```

として共有を解除します。この呼び出しを行うと、ORB の通信を行うたびにコネクションを作ります。

このため繰り返し呼び出しをする場合などは性能が低下する可能性があります。

ソケットの通信先はホスト、ポートの組で管理されます。文字列での管理ではなく、そのホスト名から検索可能な(複数の)IP アドレス単位で管理していますので、正式名、ニックネームといったホスト名の使い分けをしても同一ソケットが使われますし、マルチホームホストなどで、複数の IP アドレスを持つホストの場合で、違う IP アドレスへのコネクションがある場合でもホスト名を指定した場合は共有されます。ただし、マルチホームホストでドット表記の IP アドレスが複数指定してある場合には共有されません。

ソケット共有を行うと、サーバプロセス終了の前後に呼び出しを行った場合、終了前に作ったコネクションを終了後に使うため、サーバが再起動しているにもかかわらず通信がエラーになることがあります。このようなときは、再度通信を行うと正常に通信できます。ソケットを共有しない設定にすることも有効です(詳しい説明は Object Broker C++のオンラインマニュアルを参照してください)。

_narrow の実装

与えられたオブジェクトリファレンスに対応した実装オブジェクトが存在するメモリ空間内で_narrow を呼び出すと、実装オブジェクトへのポインタが_duplicate されて返されます。
そうでなければ、与えられたオブジェクトリファレンスに対応した Proxy オブジェクトが新しく作られ、返されます。
このときの Proxy オブジェクトの型は、オブジェクトリファレンスがサポートしているインタフェースの ID によって決まります。インタフェース ID は、オブジェクトリファレンスを生成するときに指定する、InterfaceDef に対応しています。
次の例で説明します。

```
// IDL

interface A {};
interface B {};
interface C : A, B {};
interface D : C {};
```

C の InterfaceDef でオブジェクトリファレンスを生成し、各_narrow に渡すと、次に示す結果となります。

・A::_narrow(ap)は ap に対応した CProxy を新しく作り、A_ptr 型として返します

・B::_narrow(ap)は ap に対応した CProxy を新しく作り、B_ptr 型として返します

・C::_narrow(ap)は ap に対応した CProxy を新しく作り、C_ptr 型として返します

・D::_narrow(ap)は nil オブジェクトリファレンスを返します

またサーバとクライアントでリンクしている cmn.C の数や種類が異なると次のことが起こりえます。

サーバはつぎの A, B のインターフェースをサポートしているとします。

```
// a.idl
interface A {};

// b.idl
#include "a.idl"
interface B : A {
        string op2();
};

// c.idl
#include "a.idl"
interface C {
        A op();
};
```

サーバ内の'C の実装クラス'::op()オペレーションは、次のように、B オブジェクトリファレンスを返すとします。

```
A_ptr 'C の実装クラス'::op(CORBA::Environment& env)
{
        B_ptr o = ...;
        return A::_duplicate(o, env);
}
```

実装オブジェクトが B であれば、クライアント側のスタブオブジェクトは BProxy となりますので、クライアント側にも bcmn.o が必要になります。

any を受け取るプログラムでの anonymous な型の扱い

たとえば、サーバが次のような any を受け取るオペレーションをもつインターフェースをサポートしているとします。

```
// serv.idl

interface A {
        void func(in any arg);
};
```

そして、次のような型の IDL 定義から生成されたスタブをリンクしているクライアントが、サーバを呼び出したとします。

```
// clnt.idl
struct st {
        long l1;
        long l2;
};


// C++ client code
A_ptr o = ...;
CORBA::Environment env;
st arg;

arg.l1 = 1234;
arg.l2 = 5678;

CORBA::Any aarg;

aarg <<= arg;
```

```
o->func(aarg, env);
```

clntcmn.o をリンクして作ったサーバの func オペレーションは、次のことができます。

```
void 'A の実装クラス'::func(const CORBA::Any& arg, CORBA::Environment&)
{
        st* p;
        arg >>= p;
        CORBA::Long a = p->l1;
        CORBA::Long b = p->l2;
        ...

}
```

しかし、clntcmn.o をリンクしないで作ったサーバは、st 型を扱うことができないので、上記の方法で構造体のメンバをアクセスすることができません。このように、直接扱うことのできない型のデータが入っている any を anonymous any と呼びます。

WebOTX Object Broker では、anonymous any の value には、CDR(CORBA2.2 の Common Data Representation (CDR)の節を参照)に準じた方法でエンコードした値が入っています。そして、各要素の値は Ob_MesBuf クラスを使って取り出すことができます。

```
void 'A の実装クラス'::func(const CORBA::Any& arg, CORBA::Environment&)
{
        // arg には st が入っていることを前提としています
        Ob_MesBuf mb;

        mb.from_any_value(arg.value());

        CORBA::Long a, b;

        mb >>= a;
        mb >>= b;
        ...

}
```

上記の例は、構造体なので、mb から各要素を順番に取り出しています。

また、逆に anonymous なデータ型を any に設定するときのために、次の2つの関数が規定されています。

・CORBA::Any(CORBA::TypeCode_ptr, void*, CORBA::Boolean)コンストラクタ

・void CORBA::Any::replace(CORBA::TypeCode_ptr, void*)関数

第一引数には設定するデータ型を表わす TypeCode を渡しますが、IDL 定義されていない型にはタイプコード定数がないので、プログラムの中で動的に作ります。たとえば、上記の st 型のタイプコードは次のように作成します。

```
// C++
CORBA::StructMemberSeq members;
members.length(2);

members[0].name = CORBA::string_dup("l1"); // l1 メンバのメンバ名
members[0].type = CORBA::_tc_Long; // l1 メンバのタイプコード
members[1].name = CORBA::string_dup("l2"); // l2 メンバのメンバ名
members[1].type = CORBA::_tc_Long; // l2 メンバのタイプコード

CORBA::Object_var orbobj = ...; // CORBA::ORB_init より

CORBA::Environment env;

CORBA::TypeCode_var tc =  orbobj->create_struct_tc("IDL:st:1.0", "st", members,
env);
```

第 2 引数には anonymous なデータ型のポインタを渡します。WebOTX Object Broker では、anonymous なデータは Ob_MesBuf オブジェクトを使って作成します。たとえば、上記の st 型のときには次のように作成しま

す。

```
// C++

CORBA::Long l1, l2;

l1 = 1234;
l2 = 5678;

Ob_MesBuf mb;

mb <<= l1;
mb <<= l2;
```

このようにそれぞれ作成したタイプコードと anonymous データを次のように any に渡します。

```
CORBA::Any any(tc, mb.to_any_value(), 1);
```

あるいは

```
any.replace(tc, mb.to_any_value(), 1);
```

これらの関数の第二引数に Ob_MesBuf オブジェクトで作成したデータ以外のものを渡すと、コンストラクタは値をセットしません。replace 関数は例外を返します。

なおコンストラクタは Ob_MesBuf オブジェクトで作成したデータ以外のものを渡されたときにはログファイルにメッセージを書き出します。

上記の例では、構造体を使って説明しましたが、他の型の Ob_MesBuf オブジェクトへの入れ方は取り出し方と同じです。

性能チューニング

WebOTX Object Broker では１つ１つの IIOP メッセージ全体を一旦メモリ上に置いて送受信を行っています。このとき、アプリケーションプログラムによってはメモリのフラグメンテーションのため、メモリがまだあるにもかかわらず、大きなメッセージ用の連続領域が取れないことがあります。そこで、WebOTX Object Broker では、小さなメモリを組み合わせて、仮想的な連続領域を構築しています。またソケットの I/O はこのメモリ領域の大きさ単位で行っているため、アプリケーションによっては性能に影響する場合があります。この大きさを変えることにより性能チューニングが可能な場合があります。大きさは設定値 MesBufSize で指定します。また、メモリ領域の初期割り当て個数を MesBufNum で指定することができます(詳しくは運用ガイドの「1. 環境設定について」をご参照ください)。

Ob_MesBuf クラスの使い方

CORBA2.0 以降で規定されている IIOP の CDR にしたがった方法で、バイト配列を作成したり、そこからデータを読み出したりするときに Ob_MesBuf クラスを使うと便利です。

Ob_MesBuf クラスでは主に次のことができます。

・IDL 既定義型のマーシャリング

・IDL 既定義型のアンマーシャリング

・IDL ユーザ定義型のマーシャリング

・IDL ユーザ定義型のアンマーシャリング

・encapsulation

・マーシャリングされたデータの大きさを調べる

Ob_MesBuf クラスの使い方

Ob_MesBuf クラスを使った IDL 既定義型のマーシャリングは、次のように記述します。

```
// C++
Ob_MesBuf mb; // エンコード用オブジェクトを生成
CORBA::Long l = 1000;
mb <<= l; // l をエンコードする
```

IDL 既定義型のアンマーシャリング

Ob_MesBuf クラスを使った IDL 既定義型のアンマーシャリングは、次のように記述します。

```
// C++
Ob_MesBuf mb(128); // デコード用オブジェクトを生成
CORBA::Long l;
mb >>= l; // CORBA::Long 型をデコードする
```

IDL ユーザ定義型のマーシャリング

Ob_MesBuf クラスを使ってユーザ定義型をエンコードするには、その型を IDL 定義ファイルに定義し、cmn ファイルをリンクして使います。IDL 基本型用エンコードルーチンはライブラリで提供されています。

また、Ob_MesBuf オブジェクトを使うときには、エンコード用に使うか、デコード用に使うかを、あらかじめ指定する必要があります。エンコード用に指定する方法は 2 通りあります。

コンストラクタで指定する方法

初期化処理関数と終了処理関数を使う方法

それぞれの方法を説明します。

コンストラクタで指定する方法

コンストラクタでエンコード用に指定する方法を次の encode_sample1()関数の例を使って説明します。

```
// IDL

struct st {
        long l;
        string s;
};

enum en { ee1, ee2, ee3 };


// C++

void encode_sample1()
{
        st data1;
        en data2;

        data1.l = 1234;
        data1.s = CORBA::string_dup("abcdefg");
        data2 = ee2;

        Ob_MesBuf mb; // 1. エンコード用コンストラクタ

        // IDL 定義した型は、Ob_MesBuf 型に対して operator<<=が使えるようになる
        mb <<= data1; // 2.
        mb <<= data2;
        ...

}
```

初期化処理関数と終了処理関数を使う方法

初期化処理関数と終了処理関数を使ってエンコード用に指定する方法を、次の encode_sample2()関数の例を使って説明します。

```
// IDL

struct st {
        long l;
        string s;
};
```

```
enum en { ee1, ee2, ee3 };


// C++

void encode_sample2(Ob_MesBuf& mb)
{
        st data1;
        en data2;

        data1.l = 1234;
        data1.s = CORBA::string_dup("abcdefg");
        data2 = ee2;

        CORBA::ULong length;

        // 1. ここからエンコード用として使用することを宣言します
        mb.begin_encode_message(&length);

        mb <<= data1;
        mb <<= data2;

        // 2. これでエンコード用としての使用を終了することを宣言します
        mb.end_encode_message();

        ...

}
```

mb が内部的に管理するバッファのカレントポインタからエンコード用として使用することを宣言します。このときの引数の領域には、end_encode_message()関数を呼び出したとき、使用バッファ長が書き込まれます。この領域は必ず指定してください。

end_encode_message()関数は begin_encode_message()関数に対する終了処理関数です。
begin_encode_message()関数の引数で渡ってきた領域に、begin_encode_message()から
end_encode_message()までに使われたバッファ長を書き込みます。

IDL ユーザ定義型のアンマーシャリング

Ob_MesBuf クラスを使ってユーザ定義型をデコードするには、その型を IDL 定義ファイルに定義し、cmn ファイルをリンクして使います。IDL 基本型用のデコードルーチンはライブラリで提供されています。

また、Ob_MesBuf オブジェクトを使うときには、エンコード用に使うか、デコード用に使うかを、あらかじめ指定する必要があります。デコード用に指定する方法は 2 通りあります。

コンストラクタで指定する方法

初期化処理関数と終了処理関数を使う方法

それぞれの方法を説明します。

コンストラクタで指定する方法

コンストラクタでデコード用に指定する方法を次の decode_sample1()関数の例を使って説明します。

```
// IDL

struct st {
        long l;
        string s;
};

enum en { ee1, ee2, ee3 };
```

```
// C++

void decode_sample1()
{
        char* buffer = ....; // エンコードされたデータの入っているバッファ

        // バッファ長は 1024 バイトだとする

        Ob_MesBuf mb(1024); // 1. デコード用コンストラクタ
        mb.buf_copy(buffer, 1024); // 2. buffer からデータをコピーする
        mb.reset(); // 3. カレントポインタを先頭に戻します

        st data1;
        en data2;

        // IDL 定義した型は、Ob_MesBuf 型に対して operator>>=が使えるようになる
        mb >>= data1; // 4.
        mb >>= data2;

        ...
}
```

Ob_MesBuf のインスタンスを生成します。デコード用のインスタンスはバッファ長を指定するコンストラクタを使って生成します。

エンコードされたデータをバッファから mb にコピーします。

バッファの先頭からデコードしたいので、mb が内部的に管理しているバッファのカレントポインタを先頭にします。

IDL 定義された型は Ob_MesBuf に対する operator>>=()が出力されますので、例のように簡単に記述することができます。

初期化処理関数と終了処理関数を使う方法

初期化処理関数と終了処理関数を使ってデコード用に指定する方法を、次の decode_sample2()関数の例を使って説明します。

```
// IDL

struct st {
        long l;
        string s;
};

enum en { ee1, ee2, ee3 };


// C++

void decode_sample2(Ob_MesBuf& mb)
{
        // mb にはすでにデコードしたいデータが入っているとする
        mb.begin_decode_message(100, 0); // 1.

        st data1;
        en data2;

        mb >>= data1;
        mb >>= data2;

        mb.end_decode_message(); // 2.

        ...
```

```
}
```

ここからデコード用として使うことを宣言します。このときの第一引数にはデコードするバイト数を指定しますが、この値は内部的には使っていません。第二引数にはデコードを実行するマシンのバイトオーダを指定します。これを動的に求めるには次の関数が便利です。

```
CORBA::Boolean my_byteorder()
{
        const CORBA::Boolean big_endian = 0;
        const CORBA::Boolean little_endian = 1;

        long l = 1;

        if ((*(char *)&l) == l) {
                return little_endian;
        } else {
                return big_endian;
        }
}
```

encapsulation

CDR に encapsulation という手法があります。これは CDR のデータの先頭にバイトオーダを付け、データを octet のシーケンスにカプセル化するためのものです。

encapsulation の形式でエンコードする

encapsulation の形式でエンコードする方法を encode_sample3()の例を使って説明します。

```
// IDL

struct st {
        long l;
        string s;
};

enum en { ee1, ee2, ee3 };


// C++

void encode_sample3()
{
        st data1;
        en data2;

        data1.l = 1234;
        data1.s = CORBA::string_dup("abcdefg");

        data2 = ee2;

        Ob_MesBuf mb;
        CORBA::ULong length;

        // 1. ここから encapsulation することを宣言します
        mb.begin_encode_encapsulation(&length);

        mb <<= data1;
        mb <<= data2;

        // 2. これで encapsulation を終了することを宣言します
        mb.end_encode_encapsulation();

        // encapsulation したものを sequence<octet>型にコピーしたいとします
```

```
        // IDL 定義で typedef sequence<octet> encap;と宣言したとします
        encap encap_data;

        // 3. encap_data の要素の領域を length 分確保します
        encap_data.length(length);

        // 4. mb から encap_data にコピーします
        mb.buf_copy_to(&encap_data[0], length);

        ...


}
```

mb を encapsulation 用として使うことを宣言します。このときの引数の領域には、end_encode_encapsulation()関数内で使用バッファ長が書き込まれます。この領域は必ず指定してください。また、encapsulation に必要なバイトオーダの情報は、実行したマシンの値が自動的に入ります。

end_encode_encapsulation()関数は begin_encode_encapsulation()関数に対する終了処理関数です。begin_encode_encapsulation()関数の引数で渡ってきた領域に、encapsulation に使われたバッファ長を書き込みます。

encapsulation したデータを sequence<octet>型のインスタンスにコピーしたいときは、まず encap_data のバッファ長を length 分確保します。

mb から encap_data に encapsulation したデータをコピーします。このとき、buf_copy_to()関数は、mb が内部的に管理しているバッファの最初から length 分コピーされることに注意してください。

encapsulation の形式でデコードする

encapsulation の形式でエンコードされたデータをデコードする方法を decode_sample3()の例を使って説明します。

```
// IDL

struct st {
        long l;
        string s;
};

enum en { ee1, ee2, ee3 };


// C++

void decode_sample3(Ob_MesBuf& mb)
{
        // mb にはすでにデコードしたいデータが入っているとする
        mb.begin_decode_encapsulation(100); // 1.

        st data1;
        en data2;

        mb >>= data1;
        mb >>= data2;

        mb.end_decode_message(); // 2.

        ...

}
```

ここから encapsulation されているデータをデコードすることを宣言します。このときの引数は encapsulation されているデータサイズを指定しますが、この値は内部的には使っていません。

end_decode_ensapsulation()関数は begin_decode_encapsulation 関数に対する終了処理関数です。

マーシャリングされたデータの大きさを調べる

マーシャリングされたデータの大きさを調べるには used_size()関数を使います。

以下に、マーシャリングされたデータを CORBA::Octet 型配列で取り出す例を示します。

```
Ob_MesBuf mb; // エンコード用オブジェクトの生成
mb <<= (CORBA::Long)100; // データのエンコード
CORBA::ULong len = mb.used_size(); // Byte 長を得る
CORBA::Octet* buffer = new CORBA::Octet[len];
mb.buf_copy_to(buffer, len); // buffer にエンコードデータをコピーする
```

SSL の利用方法

利用する SSL 製品の指定

Object Broker C++では、SSL 通信で利用する製品として、SecureWare/セキュリティパックと OpenSSL が選択可能です。　利用する製品は、設定の SSLProvider で指定します。指定方法は運用編 OpenSSL 関連のオプション設定を参照してください。

証明書/鍵の指定

SSL 通信で使用される証明書と鍵の指定は、SSLProvider が SSL-C の場合は、鍵 ID で指定します。

SSLProvider が OpenSSL の場合は、ファイルで指定します。指定方法は運用編 OpenSSL 関連のオプション設定を参照してください。

サポートしている証明書のファイルの形式は、PEM 形式のみです。　証明書ファイルにprivate key が含まれていない場合、プライベート鍵も指定する必要があります。プライベート鍵も指定は、運用編 OpenSSL 関連のオプション設定を参照してください。

SSL ポート番号の指定方法

ソースコードに記述する場合

SSL ポート番号をソースコードに記述する場合、以下のようにします。

```
#define SSL_PORT_NUMBER ...

int main(int argc, char** argv)
{
    ...
    orb = CORBA::ORB_init(argc, argv, "ObjectSpinner");
    orb->__use_ssl(SSL_PORT_NUMBER);
    ...
    Ob_ssl_initialize();
    ...
}
```

起動時の引数で指定する場合

SSL ポート番号を起動時の引数で指定する場合、以下のようにします。

<apName> -ORBSSLPort=port_num

環境変数で指定する場合

SSL ポート番号を環境変数で指定する場合、以下の設定名を使用します。

設定方法の詳細に関しては、運用ガイドの「1.1. オペレーティングシステム別の設定方法」を参照してください。

"ImplName"SSLPort　　SSL ポート番号を指定します。

※ "ImplName"はインプリメンテーション名を表します。

※ポート番号には 1 以上の値を使用してください。

※システムによっては、一定の範囲の値が予約されている場合があります。

鍵 ID の指定方法

ソースコードに記述する場合

鍵 ID をソースコードに記述する場合、以下のようにします。

鍵 ID が固定の場合に有効です。

サーバの鍵 ID を指定する場合

```
#define KEY_ID "..."

int main(int argc, char** argv)
{
    ...
    orb = CORBA::ORB_init(argc, argv, "ObjectSpinner");
    orb->__server_cert_key(KEY_ID);
    ...
    Ob_ssl_initialize();
    ...
}
```

クライアントの鍵 ID を指定する場合

```
#define KEY_ID "..."

int main(int argc, char** argv)
{
    ...
    orb = CORBA::ORB_init(argc, argv, "ObjectSpinner");
    orb->__client_cert_key(KEY_ID);
    ...
    Ob_ssl_initialize();
    ...
}
```

起動時の引数で指定する場合

鍵 ID を起動時の引数で指定する場合、以下のようにします。

サーバの鍵 ID を指定する場合

<apName> -ORBServerCertKey=KeyID

クライアントの鍵 ID を指定する場合

<apName> -ORBClientCertKey=KeyID

環境変数で指定する場合

鍵 ID を環境変数で指定する場合、以下の設定名を使用します。

設定方法の詳細に関しては、運用ガイドの「1.1. オペレーティングシステム別の設定方法」を参照してください。

"ImplName"ServerCertKey　　サーバの鍵 ID を指定します

ClientCertKey　　クライアントの鍵 ID を指定します

※ "ImplName"はインプリメンテーション名を表します。

クライアント証明書の要求

ソースコードに記述する場合

クライアント証明書要求をソースコードに記述する場合、以下のようにします。

```
int main(int argc, char** argv)
```

```
{
    ...
    orb = CORBA::ORB_init(argc, argv, "ObjectSpinner");
    orb->__cert_request((CORBA::Boolean)1);
    ...
    Ob_ssl_initialize();
    ...
}
```

起動時の引数で指定する場合

クライアント証明書要求を起動時の引数で指定する場合、以下のようにします。

<apName> -ORBCertRequest

環境変数で指定する場合

クライアント証明書要求を環境変数で指定する場合、以下の設定名を使用します。

設定方法の詳細に関しては、運用ガイドの「1.1. オペレーティングシステム別の設定方法」を参照してください。

"ImplName"CertRequest　　"on"を指定するとクライアント証明書を要求します

アクセプトモードの選択

ソースコードに記述する場合

接続待ちポートの選択は、以下のようにします。

SSL ポートへの接続のみ受付ける場合

```
int main(int argc, char** argv)
{
    ...
    orb = CORBA::ORB_init(argc, argv, "ObjectSpinner");
    orb->__accept_mode(Ob_AcceptSSLPortOnly);
    ...
    Ob_ssl_initialize();
    ...
}
```

SSL ポートと非 SSL ポートの両方への接続を受付ける場合

```
int main(int argc, char** argv)
{
    ...
    orb = CORBA::ORB_init(argc, argv, "ObjectSpinner");
    orb->__accept_mode(Ob_AcceptBothPort);
    ...
    Ob_ssl_initialize();
    ...
}
```

起動時の引数で指定する場合

アクセプトモードを起動時の引数で指定することは出来ません。

環境変数で指定する場合

アクセプトモードを環境変数で指定することはできません。

SSL を使用できないクライアント接続

SSL を使用した接続しか受付けないサーバに、SSL を使用できないクライアントが接続した場合、クライアントに以下の例外が返ります。

CORBA_NO_RESPONSE(1141)

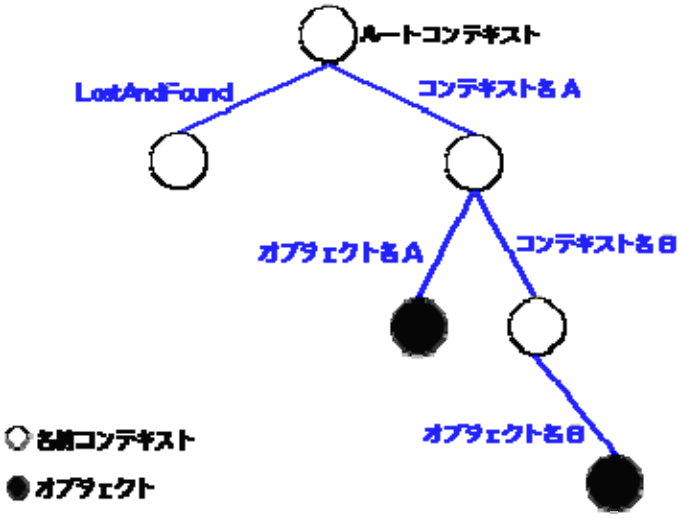
CORBA_NO_PERMISSION(1161)

WebOTX Object Broker が提供するサービスの使い方

名前サービスの利用方法

名前サービスの利用方法について説明します。

名前サービスは名前コンテキスト(Naming Context)というオブジェクトを使って、オブジェクトに階層的な名前をつける機能を提供します。名前サービスの名前階層はファイルシステムにたとえることができます。ファイルシステムのディレクトリに相当するオブジェクトが名前コンテキストです。名前コンテキストのうち、ファイルシステムのルートディレクトリに相当するものを特にルートコンテキストと呼びます。名前サービスでは、ファイルシステムと同様に名前コンテキストの下に名前コンテキストを置くことができます。名前空間の構造を図に示します。



名前空間の構造

ルートコンテキストの取得

名前の階層はルートコンテキストから始まりますので、名前サービスを利用するにはルートコンテキストを取得しなければなりません。

```
// C++
CORBA::ORB_ptr orbobj = CORBA::ORB_init(...);

// 名前サービスのルートコンテキストを得る
CORBA::Object_ptr nmsv_in_obj =
        orbobj->resolve_initial_references("NameService", env);
```

ここで得られたオブジェクトリファレンスは org.omg.CORBA.Object 型なのでこのままでは使えません。

```
// C++
CosNaming::NamingContext_ptr nmsv_root_ctx =
        CosNaming::NamingContext::_narrow(nmsv_in_obj);
```

名前の登録

任意のオブジェクトに対する名前の登録は bind または rebind により行います。

階層を持った名前も扱うことができるため、WebOTX Object Broker C++では sequence<CosNaming::NameComponent>である CosNaming::Name 型を使って名前をつけます。各階層をあらわす NameComponent 型には id と kind フィールドがあり(どちらも string 型)どちらのフィールドも使い方はアプリケーションで自由に決めてかまいません。以下は、上記 nmsv_root_ctx に id: "myobject"、kind: ""で名前をつける例です。

```
// C++
    CosNaming::Name myname;
    myname.length(1);

    myname[0].id = (const char*)"myobject";
    myname[0].kind = (const char*)"";

    CORBA::Object_ptr myobj;

    nmsv_root_ctx->bind(myname, myobj, env);
```

名前コンテキストの登録は bind_context または rebind_context により行います。使い方は bind や rebind とほとんど同じです。

```
// C++
    CosNaming::Name myname;
    myname.length(1);

    myname[0].id = (const char*)"mycontext";
    myname[0].kind = (const char*)"";

    // 名前コンテキストの生成
    CosNaming::NamingContext_var myctx = nmsv_root_ctx->new_context(env);

    // 名前の登録
    nmsv_root_ctx->bind_context(myname, myctx, env);
```

上記の例では名前コンテキストの生成と名前の登録を別々に行っていますが、この 2 つを同時に行う bind_new_context というオペレーションもあります。

```
// C++
    CosNaming::Name myname;
    myname.length(1);

    myname[0].id = (const char*)"mycontext";
    myname[0].kind = (const char*)"";

    // 名前コンテキストの生成と名前の登録
    CosNaming::NamingContext_var myctx =
       nmsv_root_ctx->bind_new_context(myname, env);
```

rebind や rebind_context は、すでに同じ名前が存在したときに新しいオブジェクトと入れ替わってしまい、もとのオブジェクトとの関連付けは失われてしまうので注意が必要です。bind, bind_context では、同じ名前が存在した場合は例外が発生します。上記の例では名前階層が 1 段だけでしたが、複数段を指定することが可能です。複数段を指定するときは、途中の名前コンテキストは実在していなければなりません。また、名前サービスではファイルシステムの"../"に相当する指定はできません。

例) nmsv_root_ctx の配下に"mycontext"と命名された名前コンテキストがあるものとします。

```
// C++
    CosNaming::Name n;

    n.length(2);

    // 名前の作成
    n[0].id = (const char *)"mycontext";
    n[0].kind = (const char *)"";
    n[1].id = (const char *)"new_context";
    n[1].kind = (const char *)"";

    CosNaming::NamingContext_var myctx2 =
       nmsv_root_ctx->bind_new_context(n, env);
```

以下に名前コンテキスト ctx1, ctx2 が同一のオブジェクト obj を指す例を示します。

例)

```
// C++
// 変数の定義
CosNaming::NamingContext_var   root1, root2, ctx1, ctx2;
CosNaming::Name       n11, n12, n21, n22;
CORBA::Object_var     obj;
CORBA::Environment    env;

// 基点になる名前コンテキストとターゲットになるオブジェクトの初期化
root1 = ...;
root2 = ...;
obj = ...;

// 名前コンテキストの作成
n11.length(1);
n11[0].id = (const char *)"alpha";
n11[0].kind = (const char *)"";
ctx1 = root1->bind_new_context(n11, env);

n12.length(1);
n12[0].id = (const char *)"beta";
n12[0].kind = (const char *)"";
ctx2 = root2->bind_new_context(n12, env);

// 2 つの名前コンテキストが同一のオブジェクトを bind する
n21.length(1);
n21[0].id = (const char *)"nameX";
n21[0].kind = (const char *)"";
ctx1->bind(n21, obj, env);

n22.length(1);
n22[0].id = (const char *)"name1";
n22[0].kind = (const char *)"";
ctx2->bind(n22, obj, env);
```

何らかの手段でオブジェクトリファレンスを獲得することにより、別のサーバで動作している名前サービスのコンテキストを bind_context で異なる名前サービスにつなぐこともできます。この操作はリモートファイルシステムのマウントにたとえることができます。別の名前サービスの一部をあたかも同一の名前サービスの名前空間の一部のように扱うことができます。

名前によるオブジェクトリファレンスの参照

名前によるオブジェクトリファレンスの参照には resolve を使います。

```
CosNaming::Name myname;
    myname.length(1);

    myname[0].id = (const char*)"myobject";
    myname[0].kind = (const char*)"";

    CORBA::Object_ptr myobj = nmsv_root_ctx->resolve(myname, env);
```

階層の作成

名前コンテキストの階層を作るには、すでに「名前の登録」で説明したとおり、new_context で名前コンテキストを作成して bind_context を呼び出すか、bind_new_context を呼び出します。bind_new_context は ew_context を呼び出した後で、bind_context を呼び出したのと同様です。new_context によって生成された名前コンテキストは、名前サーバを再起動してもデータが失われないように、ルートコンテキストの直下にある LostAndFound という特別な名前コンテキストに一時的に置かれています。

LostAndFound に登録された名前コンテキストは、bind_context で名前をつけて他のコンテキストに登録されたときに、LostAndFound から自動的に削除されます。

前述したとおり、LostAndFound は特別な名前コンテキストであり、外部からの変更は受け付けません。

LostAndFound に登録される名前は、名前サービスが自動的に生成し、削除します。したがって、LostAndFound に登録されている名前は意識する必要はありません。

名前コンテキストの一覧

ある名前コンテキストの一覧を取得するには、list を使います。ORB では、名前サーバと通信を行って一覧を取得します。一度に大量の要求をすると、通信を行う時間がかかり、なかなか処理が完了しないことがあります。なかなか処理が終了しないと、クライアントの応答性に問題がでます。したがって、list は一度に転送する最大要素数を指定できるようになっています。一度に転送できなかったときは、残りの要素の転送用に BindingIterator オブジェクトが生成されます。BindingIterator オブジェクトに対して next_n または next_one オペレーションを繰り返すことで、すべての要素を取得することができます。BindingIterator オブジェクトが返されたときは、不要になった時点で必ず org.omg.CosNaming.BindingIterator.destroy オペレーションを実行してオブジェクトを削除しなければなりません。削除しなかった BindingIterator オブジェクトは永久に残ります。

例)

```
// C++
CORBA::Environment env;

CosNaming::NamingContext_var ctx1 = ...

// ctx1 に 10 個の名前を登録しておく。

...

CosNaming::BindingList_var bl;

CosNaming::BindingIterator_var bi;

// 1 つだけ bl に取り出して、残りは bi から参照することにした
ctx1->list(1L, bl, bi, env);

printf("id : %s¥nkind : %s¥n",
   bl[0].binding_name[0].id, bl[0].binding_name[0].kind);

// bi から 1 つだけ取り出す
CosNaming::Binding_var b;

bi->next_one(b, env);

printf("id : %s¥nkind : %s¥n",
   b->binding_name[0].id, b->binding_name[0].kind);

// bi から 5 つまとめて取り出す
bi->next_n(5L, bl, env);

for(int i = 0; i < 5; i++){

printf("id : %s¥nkind : %s¥n",
   bl[i].binding_name[0].id, bl[i].binding_name[0].kind);

}

// bi の削除
bi->destroy(env);
```

名前の削除

名前の削除には、名前コンテキストであるか通常のオブジェクトであるかに関わらず unbind を使います。

CORBA オブジェクトの削除と名前サービスに登録されている情報は連動しているわけではありません。

名前サービスは、名前とオブジェクトリファレンスのペアを単に記憶しているだけですので、unbind を呼び出しても、名前とオブジェクトリファレンスの関係が絶たれるだけで、登録されているオブジェクト自体が削除されるわけではありません。また、登録されている CORBA オブジェクトが削除されたとしても名前サービスに

は登録された名前とオブジェクトリファレンスのペアは残ったままになります。

名前コンテキストの unbind の呼び出しは、そのコンテクストにオブジェクトやコンテクストが存在する場合にも成功します。unbind によって名前を失った名前コンテキストは、ルートコンテキスト直下にある LostAndFound というコンテキストに再登録されます。LostAndFound への再登録には、名前サービスによって自動的に割り当てた名前が使われます。名前コンテキスト以外のオブジェクトの unbind では LostAndFound につながれることはありません。

名前コンテキストオブジェクトの削除には名前コンテキストオブジェクトの destroy オペレーションを使います。名前コンテキストの destroy では、削除しようとする名前コンテキストの配下に名前が登録されていてはいけません。もし、削除しようとする名前コンテキストの配下に名前がつながっているときは NotEmpty 例外が発生します。

注意:

名前の付いていない名前コンテキストは LostAndFound に自動的につながれます。しかし、LostAndFound に自動的につながれた名前に対しては unbind を呼び出す必要はありません。destroy の内部で自動的に削除されます。必要なくなった名前コンテキストは削除するべきです。

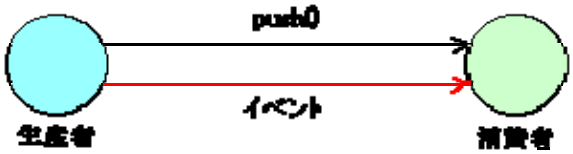
名前コンテキストの unbind と destroy は、どちらを先に行ってもかまいません。しかし、配下に名前がつながっている可能性がある場合、unbind を先に行ってしまうと配下につながっている名前の検索や削除が困難になってしまいます。したがって、配下につながっている名前は、事前に unbind を使って削除しておくようにしてください。

イベントサービスの利用方法

イベントサービスはイベントと呼ぶデータをひとかたまりとして、イベントの送信者(以後生産者と呼びます)からイベントの受信者(以後消費者と呼びます)へ送るサービスです。WebOTX Object Broker では生産者と消費者の仲介を行うイベントチャネルサーバとして提供されています。

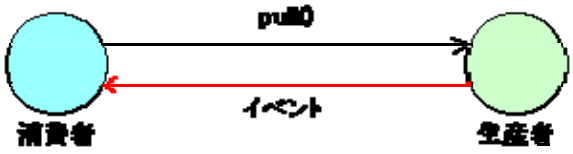
プッシュモデルとプルモデル

プッシュモデルとは消費者が CORBA オブジェクトとして実装され、生産者は消費者オブジェクトの push オペレーションを呼び出すことによりイベントを送るモデルです。



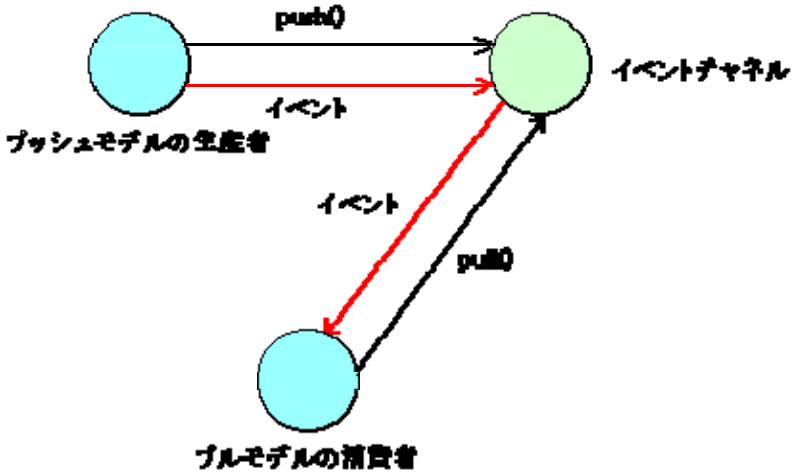
プッシュモデル

プルモデルとは生産者が CORBA オブジェクトとして実装され、消費者が生産者オブジェクトの pull オペレーションを呼び出すことによりイベントを受け取るモデルです。



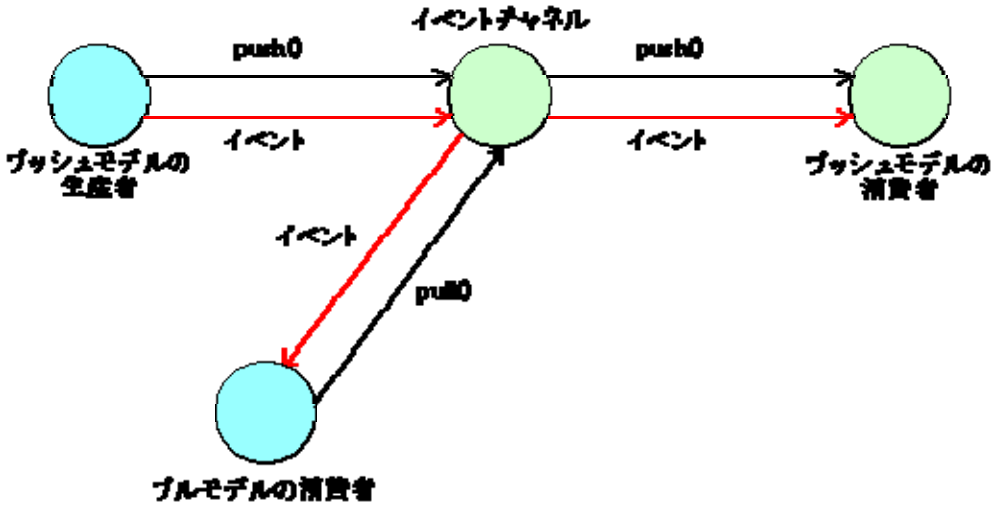
プルモデル

WebOTX Object Broker では、各生産者、消費者はイベントチャネルサーバを介して通信を行いますので、相手のモデルを気にすることなく通信することができます。たとえば、プッシュモデルの生産者とプルモデルの消費者との間でイベント通信を行なうことができます。



イベントチャネル

また、複数の生産者、複数の消費者が 1 つのイベントチャネルを介してイベントを送ることができます。1 つの生産者が生産したイベントは、イベントチャネル内のバッファに一時的に貯えられ、その時点で接続しているすべての消費者に対して送られます。イベントはすべての消費者が受信した段階でイベントチャネル内のバッファから取り除かれます。



複雑な使い方

注意

生産者、および消費者は明示的に disconnect_X_Y(X: push or pull, Y: supplier or consumer)オペレーションを呼ぶまで接続しているものとして扱われます。disconnect_X_Y を呼ばずに、消費者を終了すると、イベントを取得していない消費者がいるものとして扱われ、やがてバッファあふれが起こり新規のイベントが送られなくなりますのでご注意ください。イベントチャネル内の

バッファの大きさは設定で変更できます。

イベントチャネルサーバの使い方

WebOTX Object Broker のイベントチャネルサーバは、Windows 版では eventsv.exe、UNIX 版(HP-UX 版, Solaris 版, Linux 版)では eventsv という名前です。これらを利用するためには次の手順が必要です。

オブジェクト生成

WebOTX Object Broker のイベントチャネルサーバを利用するためには、org.omg.CosEventChannelAdmin.EventChannel をサポートするイベントチャネルオブジェクトを生成する必要があります。しかし、WebOTX Object Broker JavaTM はイベントチャネルオブジェクトを作成することができません。したがって、C++を使ったアプリケーションによって登録する必要があります。以下に C++を使ってオブジェクトを生成する方法を示します。

イベントチャネルサーバは自動起動で利用します。

自動起動引数はデフォルトの位置で渡します。つまり、CORBA::ORB::_server_argument_index(CORBA::ULong)は既定値(0)のままにします。

活性化方針は CORBA_SharedServer を指定します。

スレッド処理方針は pool あるいは permethod を指定します。

```
// C++によるコーディング例

CORBA_ORB_ptr orb = CORBA_ORB_init(...);

orb->__implementation_name("eventchimpl");

orb->__server_path_name("サーバ上での eventsv.exe または eventsv のパス名");

CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");

PortableServer::POA_var rootPOA = PortableServer::POA::_narrow(obj);

// LifespanPolicy として PERSISTENT をもつ POA を作成
CORBA::PolicyList plist;

plist.length(1);

plist[0] = rootPOA->create_lifespan_policy(PortableServer::PERSISTENT);

PortableServer::POA_var poa =
    rootPOA->create_POA("MyPOA", PortableServer::POAManager::_nil(), plist);

CORBA::Object_var eventch_obj =
    poa->create_reference("IDL:omg.org/CosEventChannelAdmin/EventChannel:1.0");

char *objstr = orb->object_to_string(eventch_obj);

// これを Java アプリケーションに渡す
```

生産者としての使い方

(1) プッシュモデルの生産者

プッシュモデルの生産者は push オペレーションを実行するクライアントとして実装します。次の手順でイベントチャネルとの接続を行います。

1. イベントチャネルオブジェクトのオブジェクトリファレンス(org.omg.CosEventChannelAdmin.EventChannel インタフェース)を取得します。
2. イベントチャネルオブジェクトの for_suppliers オペレーション(C++)生産者管理オブジェクトを取得します。
3. 生産者管理オブジェクトの obtain_push_consumer オペレーション(C++)でプロクシプッシュ消費者オブジェクトを取得します。
4. 3.で取得したプロクシプッシュ消費者オブジェクトに対してこの生産者のプッシュ生産者オブジェク

トを引数として connect_push_supplier オペレーション(C++)を呼び出します。このとき、プッシュ生産者オブジェクトは nil オブジェクトでもかまいません。

5. 以後、3.で取得したプロクシプッシュ消費者オブジェクトの push オペレーション(C++)を呼び出して生産したイベントを送ります。
6. 終了時には 3.で取得したプロクシプッシュ消費者オブジェクトの disconnect_push_consumer オペレーション(C++)を実行します。

(2) プルモデルの生産者

プルモデルの生産者は org.omg.CosEventComm.PullSupplier インタフェースをサポートするオブジェクトとして実装します。次の手順でイベントチャネルとの接続を行います。

1. イベントチャネルオブジェクトのオブジェクトリファレンス(org.omg.CosEventChannelAdmin.EventChannel インタフェース)を取得します。
2. イベントチャネルオブジェクトの for_suppliers オペレーション(C++)で生産者管理オブジェクトを取得します。
3. 生産者管理オブジェクトの obtain_pull_consumer オペレーション(C++)でプロクシプル消費者オブジェクトを取得します。
4. 3.で取得したプロクシプル消費者オブジェクトに対してこの生産者のプル生産者オブジェクトを引数として connect_pull_supplier オペレーション(C++)を呼び出します。
5. 以後、イベントチャネルは pull(C++)または try_pull オペレーション(C++)により、このプル生産者からイベントを受け取ります。これらのオペレーションの実装コードで生産したイベントを返してください。
6. 終了時には 3.で取得したプロクシプル消費者オブジェクトの disconnect_pull_consumer オペレーション(C++)を実行します。

プルモデルの生産者を使うと、各生産者に対応して、イベントチャネルサーバプロセス内に専用のスレッドが 1 つ生成されます。

消費者としての使い方

(1) プッシュモデルの消費者

プッシュモデルの消費者は org.omg.CosEventComm.PushConsumer インタフェースをサポートするオブジェクトとして実装します。次の手順でイベントチャネルとの接続を行います。

1. イベントチャネルオブジェクトのオブジェクトリファレンス(org.omg.CosEventChannelAdmin.EventChannel インタフェース)を取得します。
2. イベントチャネルオブジェクトの for_consumers オペレーション(C++)で消費者管理オブジェクトを取得します。
3. 消費者管理オブジェクトの obtain_push_supplier オペレーション(C++)でプロクシプッシュ生産者オブジェクトを取得します。
4. 3.で取得したプロクシプッシュ生産者オブジェクトに対してこの消費者のプッシュ消費者オブジェクトを引数として connect_push_consumer オペレーション(C++)を呼び出します。
5. 以後、イベントチャネルは push オペレーション(C++)により、このプッシュ消費者に対してイベントを送ります。push オペレーションの実装コードでイベントを受け取ったときの処理を行ってください。
6. 終了時には 3.で取得したプロクシプッシュ生産者オブジェクトの disconnect_push_supplier オペレーション(C++)を実行します。

プッシュモデルの消費者を利用すると、各消費者に対応して、イベントチャネルサーバプロセス内に 1 つ、専用のスレッドが生成されます。

(2) プルモデルの消費者

プルモデルの消費者は pull または try_pull オペレーションを実行するクライアントとして実装します。次の手順でイベントチャネルとの接続を行います。

1. イベントチャネルオブジェクトの for_consumers オペレーション(C++)で消費者管理オブジェクトを取得します。
2. イベントチャネルオブジェクトの for_consumers オペレーション(C++)で消費者管理オブジェクトを取

得します。

3. 消費者管理オブジェクトの obtain_pull_supplier オペレーション(C++)でプロクシプル生産者オブジェクトを取得します。
4. 3.で取得したプロクシプル生産者オブジェクトに対してこの消費者のプル消費者オブジェクトを引数として connect_pull_consumer オペレーション(C++)を呼び出します。このとき、プル消費者オブジェクトは nil オブジェクトでもかまいません
5. 以後、3.で取得したプロクシプル生産者オブジェクトの pull(C++)または try_pull オペレーション(C++)を呼び出してイベントを受け取ります。pull オペレーションではイベントが発生するまで、この関数から戻りません。通常設定されている、呼びだしタイムアウトが設定してある場合で、タイムアウトが発生すると、イベントが 1 つ失われます。この場合、try_pull を使うようにしてください。

終了時には 3.で取得したプロクシプル生産者オブジェクトの disconnect_pull_supplier オペレーション(C++)を実行します。

7.2.7.Object Broker Java の機能

本項の記述は、Object Broker Java にのみ当てはまります。

7.2.7.1. Object Broker Java のプロパティ設定

Object Broker Java に対して属性を設定することができます。設定した属性は ORB 初期化時に読み込まれます。

設定可能なプロパティの一覧は、運用編「4.10.3 Object Broker Java™ における ORB のプロパティ定義」を参照してください。

Java アプリケーションでのプロパティ設定方法は以下のとおりです。優先度が高い順に記載しています。

org.omg.CORBA.ORBClass および org.omg.CORBA.ORBSingletonClass は Java システムプロパティに指定する方法で設定してください。それ以外の設定方法では有効になりません。

・ ORB.init メソッドの第 1 パラメータに渡す

次のように、アプリケーション起動時の引数に指定します。ORB.init(String[], java.util.Properties)の第 1 パラメータに、main 関数のパラメータをそのまま渡すようにプログラミングされている必要があります。

```
> java <AP のクラス名> -CorbalocAskWithMT true -CodeSetEncoding
OSF_SJIS1=MS932
```

・ プロパティ定義ファイルを指定する

次のように、テキストファイルに「-<プロパティ名>=<設定値>」を記述します。各行の先頭にはハイフンが必要です。

```
-CorbalocAskWithMT=true
-CodeSetEncoding=OSF_SJIS1=MS932
```

作成したプロパティ定義ファイルの名前を次のように指定します。

```
> java <APのクラス名> -PropertyFile <プロパティ定義ファイル名>
```

・ ORB.init メソッドの第 2 パラメータに渡す

java.util.Properties オブジェクトにプロパティを設定し、ORB.init(String[], java.util.Properties)の第 2 パラメータに渡します。

```
      :
      :
java.util.Properties props = new java.util.Properties();
props.setProperty("CorbalocAskWithMT", "true");
props.setProperty("CodeSetEncoding", "OSF_SJIS1=MS932");

ORB orb = ORB.init(args, props);
      :
      :
```

・ Java システムプロパティに指定する

次のように、アプリケーション起動時の Java システムプロパティに指定します。ORB 初期化より前の位置であれば、プログラム中に java.lang.System.setProperty メソッドで記述することもできます。

```
> java -Djp.co.nec.orb.CorbalocAskWithMT=true <APのクラス名>
```

・ 統合運用管理ツール、運用管理コマンドで設定する

アプリケーションを起動するときの設定は必要ありません。

統合運用管理ツール、運用管理コマンドでの指定は、Standard-J/Standard/Enterprise の各エディションで、EJB もしくは CORBA のサーバ上でのみ有効になります。

・ ユーザのホームディレクトリ(${user.home})の orb.properties に指定する

次のように記述したファイルをユーザのホームディレクトリに orb.properties という名前で作成します。そのユーザ権限で動作するすべてのアプリケーションで設定が有効になります。アプリケーションを起動するときの設定は必要ありません。

```
CorbalocAskWithMT=true
CodeSetEncoding=OSF_SJIS1=MS932
```

・ Java のホームディレクトリ(${java.home})配下の lib ディレクトリの orb.properties に指定する

次のように記述したファイルを Java のホームディレクトリ配下の lib ディレクトリに orb.properties という名前で作成します。その Java ランタイムで動作するすべてのアプリケーションで設定が有効になります。アプリケーションを起動するときの設定は必要ありません。

```
CorbalocAskWithMT=true
CodeSetEncoding=OSF_SJIS1=MS932
```

・ API で指定する

ORB 初期化より前の位置で、プログラム中に jp.co.nec.orb.Config クラスの各メソッドを呼び出して設定します。

```
        :
        :
jp.co.nec.orb.Config.corbalocAskWithMT();
jp.co.nec.orb.Config.setCodeSetEncoding("OSF_SJIS1", "MS932");
        :
ORB orb = ORB.init(args, props);     // ORB 初期化
        :
        :
```

Java アプレットの場合の設定方法は以下のとおりです。優先度が高い順に記載しています。

org.omg.CORBA.ORBClass および org.omg.CORBA.ORBSingletonClass は HTML の PARAM タグに記述する方法で設定してください。それ以外の設定方法では有効になりません。

・ HTML の PARAM タグに記述する

次のように、HTML の PARAM タグに記述します。ORB.init(java.applet.Applet, java.util.Properties)の第 1 パラメータに、アプレット自身のオブジェクトを渡すようにプログラミングされている必要があります。

```
<HTML>
        :
<param name="jp.co.nec.orb.CorbalocAskWithMT" value="true">
<param name="jp.co.nec.orb.CodeSetEncoding" value="OSF_SJIS1=MS932">
        :
</HTML>
```

・ ORB.init の第 2 パラメータに渡す

java.util.Properties にプロパティを設定し、ORB.init(String[], java.util.Properties)の第 2 パラメータに渡します。

```
        :
        :
java.util.Properties props = new java.util.Properties();
props.setProperty("CorbalocAskWithMT", "true");
props.setProperty("CodeSetEncoding", "OSF_SJIS1=MS932");
        :
ORB orb = ORB.init(applet, props);
        :
        :
```

・ API で指定する

ORB 初期化より前の位置で、プログラム中に jp.co.nec.orb.Config クラスの各メソッドを呼び出して設定します。

```
        :
        :
jp.co.nec.orb.Config.corbalocAskWithMT();
jp.co.nec.orb.Config.setCodeSetEncoding("OSF_SJIS1", "MS932");
        :
ORB orb = ORB.init(applet, props);
        :
        :
```

7.2.7.2. 呼び出しタイムアウトの設定

オブジェクト呼び出しのタイムアウトは、次に示す単位で設定することができます。以下の順に優先度は低くなります。

・ プロセス内一律

・ インタフェース単位

・ オブジェクト単位

設定しない場合の既定値は 30 秒です。

タイムアウト時間の既定値は、WebOTX Ver6.1 から 30 秒に変更になりました。WebOTX Ver5 およびそれ以前のバージョンでは、無制限(タイムアウトしない)が既定値です。

オペレーションを呼び出してから応答が返るまでの間に、設定した時間が過ぎてしまったときは、org.omg.CORBA.NO_RESPONSE システム例外(マイナーコード 5130)が返されます。

1. プロセス内一律のタイムアウト設定

1-1. プロパティによる設定

クライアントプログラムの RequestTimeout プロパティに秒単位で設定します。

設定方法は、本節の「Object Broker Java のプロパティ設定」の項を参照してください。

1-2. API による設定

クライアントプログラム上で jp.co.nec.orb.Config.setTimeout(int)を呼び出して設定します。パラメータには、タイムアウト時間をミリ秒単位で設定します。

```
// タイムアウトを 10 秒に設定
jp.co.nec.orb.Config.setTimeout(10000);
```

2. インタフェース単位のタイムアウト設定

2-1. API による設定

クライアントプログラム上で jp.co.nec.orb.Config.setTimeout(String, int)を呼び出して設定します。パラメータには、タイムアウト時間をミリ秒単位で設定します。

```
// Hello インタフェースのタイムアウトを 10 秒に設定
jp.co.nec.orb.Config.setTimeout("Hello", 10000);
```

3. オブジェクト単位のタイムアウト設定

3-1. API による設定

クライアントプログラム上で jp.co.nec.orb.Config.setTimeout(org.omg.CORBA.Object, int)を呼び出して設定します。パラメータには、タイムアウト時間をミリ秒単位で設定します。

```
// CORBA オブジェクトのタイムアウトを 10 秒に設定
org.omg.CORBA.Object obj = ....
jp.co.nec.orb.Config.setTimeout(obj, 10000);
```

7.2.7.3. フック

Object Broker Java では、あらかじめ登録されたユーザ定義メソッドをオペレーション呼び出しの処理中にコールバックする機能を提供しています。この機能をフックと呼びます。

フックは CORBA 仕様では定義されていない、Object Broker 独自の機能です。

Object Broker Java のフックには、以下の 2 種類のレベルがあります。

・ パラメータレベル

・ メッセージレベル

また、これらのフックを次に示すレベルで登録することができます。

・ システムレベル

プロセス内で呼び出しが実行されると、登録したフックが呼び出されます。

・ インタフェースレベル

指定されたインタフェース名の呼び出しが実行されると、登録したフックが呼び出されま

す。

・オブジェクトレベル

指定されたオブジェクトの呼び出しが実行されると、登録したフックが呼び出されます。

フックの登録

フックは、jp.co.nec.orb.HookObj クラスを継承して、メソッドをオーバーライドすることで実装します。フックの登録は、フックの実装クラス(HookObj)の次のメソッドを呼び出します。登録後に呼び出しが実行されると、オーバーライドしたメソッドがコールバックされます。

・システムレベル

```
public final void Register();
```

・インタフェースレベル

```
public final void Register(String ifname);
```

・オブジェクトレベル(サーバ側)

```
public final void Register(org.omg.PortableServer.Servant execobj);
```

・オブジェクトレベル(クライアント側)

```
public final void Register(org.omg.CORBA.Object execobj);
```

フックの削除

フックの削除は、フックの実装クラス(HookObj)の次のメソッドを呼び出します。

・システムレベル

```
public final void Cancel();
```

・インタフェースレベル

```
public final void Cancel(String ifname);
```

・オブジェクトレベル(サーバ側)

```
public final void Cancel(org.omg.PortableServer.Servant execobj);
```

・オブジェクトレベル(クライアント側)

```
public final void Cancel(org.omg.CORBA.Object execobj);
```

インタフェースレベルの登録、削除で指定するインタフェース名はリポジトリ ID を指定してください。リポジトリ ID は Helper クラスの id()メソッドで取得することができます。

パラメータレベルのフック

パラメータレベルのフックには、次の 4 ヶ所のコールバックポイントがあります。各コールバックメソッドがオーバーライドされていれば、各ポイントで呼び出されます。

A. クライアントアプリケーションのオブジェクト呼び出し直後

```
public int ClientBeforeSend(Request req, ObjectImpl execobj)
```

B. サーバアプリケーションのオブジェクト呼び出し直前

```
public int ServerAfterReceive(ServerRequest req, Servant execobj)
```

C. サーバアプリケーションのオブジェクトの戻り直後

```
public int ServerBeforeSend(ServerRequest req, Servant execobj)
```

D. クライアントアプリケーションのオブジェクトの戻り直前

```
public int ClientAfterReceive(Request req, ObjectImpl execobj)
```

パラメータレベルのフックは、スタブ/スケルトンが DSI/DII で生成された場合のみ使用することができます。スタブ/スケルトンを DSI/DII で生成するには、IDL コンパイラに -notstream を指定する必要があります。DSI/DII でない場合、フックを登録してもエラーになりませんが、コールバックは呼び出されません。

ServerAfterReceive では、ServerRequest クラスのメソッドで最初に値を取得する時に、引数、コンテキストの順で値を取得しなければなりません。

メッセージレベルのフック

メッセージレベルのフックには、次の 4 ヶ所のコールバックポイントがあります。各コールバックメソッドがオーバーライドされていれば、各ポイントで呼び出されます。

E. クライアントアプリケーションのオブジェクト呼び出し直後

```
public int ClientBeforeSend(MessageBuffer buffer)
```

F. サーバアプリケーションのオブジェクト呼び出し直前

```
public int ServerAfterReceive(MessageBuffer buffer)
```

G. サーバアプリケーションのオブジェクトの戻り直後

```
public int ServerBeforeSend(MessageBuffer buffer)
```

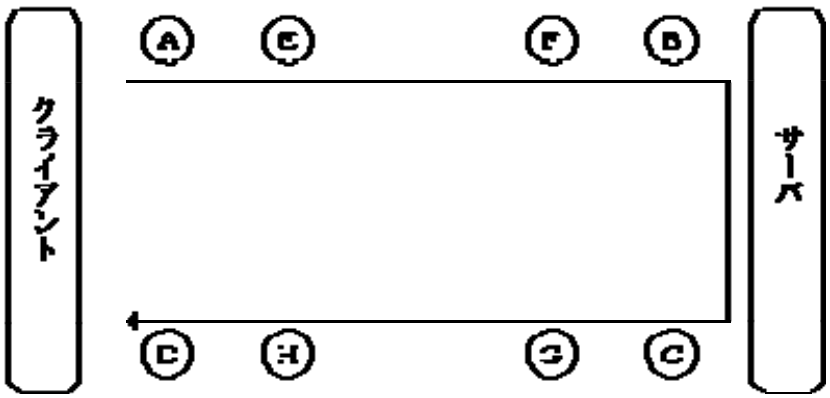
H. クライアントアプリケーションのオブジェクトの戻り直前

```
public int ClientAfterReceive(MessageBuffer buffer)
```

MessageBuffer クラスは、リクエスト、リプライメッセージを取り扱うためのクラスで、このクラスのメソッドを用いてメッセージの値の取得や、新しいメッセージの登録を行うことができます。

ServerAfterReceive、ClientAfterReceive では、引数の MessageBuffer クラスが保持するバッファのサイズはメッセージサイズに関係なく一定です。メッセージがフラグメントで分割されていてメッセージサイズがバッファサイズより長い場合は、メッセージの先頭からバッファサイズ分までが渡されます。

次の図は各メソッドが実行される順番を示しています。矢印がメッセージの流れ、アルファベットが各メソッドを表しています。



動的スケルトンインタフェース(DSI)の場合、サーバ側のフックは使用できません。

各レベルで登録できるメソッドは、以下のとおりです。

	パラメータレベル				メッセージレベル			
	A	B	C	D	E	F	G	H
システムレベル	○	○	○	○	○	○	○	○
インタフェースレベル	－	○	○	－	－	－	－	－
オブジェクトレベル	○	○	○	○	○	－	－	○

戻り値によるフック実行の制御

フックは、オブジェクトレベル、インタフェースレベル、システムレベルの順に実行されます。また、同じレベルで登録されたフックは、登録された順番で実行されます。メソッドの戻り値によって、処理の中断やスキップを行うことができます。フックを作成するときには、戻り値に注意しなければなりません。

戻り値による制御は次の通りです。

- 戻り値が 0 である場合　次のフックを実行します。
- 戻り値が 1 である場合　同じレベルのフックを実行せず、次のレベルに移ります。
- 戻り値が 2 である場合　以降のフック処理を実行しません。
- 戻り値が 3 である場合　パラメータレベルのフック処理のクライアント折り返し、サーバ折り返しを実行します。
- 上記以外である場合　フック処理を中断し、エラーログに出力します。

戻り値で 3 を指定できるのはパラメータレベルのフックの A, B の場合だけです。これ以外のメソッドで戻り値に 3 を指定すると、フック処理を中断します。

7.2.7.4. インタフェースリポジトリ

インタフェースリポジトリは CORBA オブジェクトのインタフェース情報を管理します。インタフェースリポジトリにはインタフェース情報の登録、削除、および参照という機能があります。

動的起動インタフェース(DII)を使った呼び出しを行うとき、ターゲットとなるオブジェクトのインタフェース情報が必要になります。このようなとき、インタフェースリポジトリから情報を取り出します。

WebOTX Object Broker では instif, rmif コマンドを使うことでインタフェース情報の登録と削除を簡単に行うことができます。したがって、ここでは参照する方法を説明します。

リポジトリオブジェクトの取得

インタフェースリポジトリは、リポジトリオブジェクトを頂点とする階層構造をしています。インタフェースリポジトリを利用するには、まず、リポジトリオブジェクトのオブジェクトリファレンスを取得する必要があります。

```
import org.omg.CORBA.*;
public class MyProg {
    public static void main(String args[]) {
        ORB orbobj = ORB.init(args, null);

        org.omg.CORBA.Object ir_in_obj =
            orbobj.resolve_initial_references("InterfaceRepository");
        Repository repository =
            RepositoryHelper.narrow(ir_in_obj);
                :
                :
    }
}
```

インタフェース情報の取得

あるオブジェクトが内包しているオブジェクトの一覧を返すものや、内包しているオブジェクトから手がかりになる文字列を使って検索するものなどさまざまな取得方法があります。

ここでは、org.omg.CORBA.Container.describe_contents と org.omg.CORBA.InterfaceDef.describe_interface の例を示します。それ以外のオペレーションは API リファレンス編の「インタフェースリポジトリ」を参照してください。

次の IDL 定義が instif コマンドによりインタフェースリポジトリに登録されているものとします。

```
interface myintf {
    void op(in short i);
};
```

このデータをインタフェースリポジトリから取り出す例を以下に示します。なお、例外処理や複数のインタフェースやオペレーションが含まれていた場合の考慮などは省略します。

```
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class client {
    public static void main(String args[]) {
        // 引数モード別
        final String[] opemode = {
            "in", "out", "inout"
        };

        // 型別
        final String[] tckind = {
            "null", "void", "short", "long", "ushort", "ulong",
            "float", "double", "boolean", "char", "octet",
            "any", "TypeCode", "Principal", "object", "struct",
            "union", "enum", "string", "sequence", "array",
            "typedef", "exception", "longlong", "ulonglong",
            "longdouble", "wchar", "wstring", "fixed", "value",
            "value_box", "native", "abstract", 0
        };

        try {
            // 初期化
            ORB orbobj = ORB.init(args, null);

            // リポジトリオブジェクトの取り出し
            org.omg.CORBA.Object ir_in_obj =
                orbobj.resolve_initial_references(
                        "InterfaceRepository");
            Repository repository =
```

```
                    RepositoryHelper.narrow(ir_in_obj);

                // リポジトリオブジェクトから
                // InterfaceDef オブジェクトだけを取り出す
                // CORBA_dk_Interface   : 取り出すのはインタフェースのみ
                // true         : 継承されたオブジェクトは含まない
                // -1           : 数量制限なし
                org.omg.CORBA.ContainerPackage.Description descseq[] =
                    repository.describe_contents(
                        DefinitionKind.dk_Interface, true, -1);

                // シーケンスの最初のオブジェクトを narrow する
                InterfaceDef intfobj =
                    InterfaceDefHelper.narrow(
                        descseq[0].contained_object);

                // インタフェース情報を取り出す
                org.omg.CORBA.InterfaceDefPackage.FullInterfaceDescriptio
n
                    intfdesc = intfobj.describe_interface();

                // インタフェース名
                System.out.println("interface " + intfdesc.name + "{");

                // オペレーションのシーケンスから１つ取り出す
                OperationDescription opedesc = intfdesc.operations[0];

                // オペレーションのモード
                if(opedesc.mode == OperationMode.OP_ONEWAY)
                    System.out.println("    oneway ");

                // パラメータのシーケンスから１つ取り出す
                ParameterDescription paradesc = opedesc.parameters[0];

                // 戻り値の型, オペレーション名, 引数のモード, 引数を表示
                System.out.println("    "
                    + tckind[opedesc.result.kind().value()] + " "
                    + opedesc.name + "("
                    + opemode[paradesc.mode.value()] + " "
                    + tckind[paradesc.type.kind().value()]  + " "
                    + paradesc.name + ")");

                System.out.println("};");
            } catch(Exception e) {
                System.out.println("ERROR : " + e);
            }
        }
    }
```

org.omg.CORBA.Container.describe_contents()は、そのオブジェクトが内包しているオブジェクトの一覧を org.omg.CORBA.ContainerPackage.Description の配列で返します。上記の例では InterfaceDef に限定して検索をしています。

org.omg.CORBA.ContainerPackage.Description クラスの定義は以下のとおりです。

```
package org.omg.CORBA.ContainerPackage;

final public class Description {
    // instance variables
    public org.omg.CORBA.Contained contained_object;
    public org.omg.CORBA.DefinitionKind kind;
    public org.omg.CORBA.Any value;

    // constructors
    public Description(
        org.omg.CORBA.Contained _contained_object,
        org.omg.CORBA.DefinitionKind _kind, org.omg.CORBA.Any _value
    ) {
        contained_object = _contained_object;
        kind = _kind; value = _value;
    }

    public Description() {}

}
```

contained_object には InterfaceDef オブジェクトが入っています。そこで、org.omg.CORBA.InterfaceDef.describe_interface()を使って、その org.omg.CORBA.InterfaceDef オブジェクトが内包しているオペレーションのリストを取り出します。オペレーションのリストは org.omg.CORBA.InterfaceDefPackage.FullInterfaceDescription として返されます。

org.omg.CORBA.InterfaceDefPackage.FullInterfaceDescription クラスの定義は以下のとおりです。

```
package org.omg.CORBA.InterfaceDefPackage;

final public class FullInterfaceDescription {
    // instance variables
    public java.lang.String name;
    public java.lang.String id;
    public java.lang.String defined_in;
    public java.lang.String version;
    public org.omg.CORBA.OperationDescription[] operations;
    public org.omg.CORBA.AttributeDescription[] attributes;
    public java.lang.String[] base_interfaces;
    public org.omg.CORBA.TypeCode type;
    public boolean is_abstract;

    // constructors
    public FullInterfaceDescription(
        java.lang.String _name,
        java.lang.String _id,
        java.lang.String _defined_in,
        java.lang.String _version,
        org.omg.CORBA.OperationDescription[] _operations,
        org.omg.CORBA.AttributeDescription[] _attributes,
        java.lang.String[] _base_interfaces,
        org.omg.CORBA.TypeCode _type,
        boolean _is_abstract
    ) {
        name = _name;
        id = _id;
        defined_in = _defined_in;
        version = _version; operations = _operations;
        attributes = _attributes;
        base_interfaces = _base_interfaces; type = _type;
        is_abstract = _is_abstract;
    }
```

```
    public FullInterfaceDescription() {}
}
```

さらに、operations から目的のオペレーションの型情報を取り出します。

operations は org.omg.CORBA.OperationDescription クラスの配列です。
org.omg.CORBA.OperationDescription クラスの定義を以下に示します。

```
package org.omg.CORBA;

final public class OperationDescription {
    /* instance variables*/
    public java.lang.String name;
    public java.lang.String id;
    public java.lang.String defined_in;
    public java.lang.String version;
    public org.omg.CORBA.TypeCode result;
    public OperationMode mode;
    public java.lang.String[] contexts;
    public ParameterDescription[] parameters;
    public ExceptionDescription[] exceptions;
    /* constructors*/
    public OperationDescription(
        java.lang.String _name,
        java.lang.String _id,
        java.lang.String _defined_in,
        java.lang.String _version,
        org.omg.CORBA.TypeCode _result,
        OperationMode _mode,
        java.lang.String[] _contexts,
        ParameterDescription[] _parameters,
        ExceptionDescription[] _exceptions
    ) {
        name = _name;
        id = _id;
        defined_in = _defined_in;
        version = _version;
        result = _result;
        mode = _mode;
        contexts = _contexts;
        parameters = _parameters;
        exceptions = _exceptions;
    }

    public OperationDescription() {}
}
```

parameters からパラメータ情報を取り出します。

parameters は org.omg.CORBA.ParameterDescription クラスの配列です。
org.omg.CORBA.ParameterDescription クラスの定義を以下に示します。

```
// 引数の種別
// PARAM_IN : in パラメータ
// PARAM_OUT : out パラメータ
// PARAM_INOUT : inout パラメータ
package org.omg.CORBA;
final public class ParameterMode {
    private int _v;
    public static final int _PARAM_IN = 0;
    public static final int _PARAM_OUT = 1;
    public static final int _PARAM_INOUT = 2;
    public static final ParameterMode PARAM_IN = new
ParameterMode(_PARAM_IN);
    public static final ParameterMode PARAM_OUT = new
ParameterMode(_PARAM_OUT);
    public static final ParameterMode PARAM_INOUT = new
ParameterMode(_PARAM_INOUT);
    public int value() { return _v; }
    private ParameterMode(int v) { _v = v; }
    ...
}

// ParameterDescription
package org.omg.CORBA;
final public class ParameterDescription {
    // instance variables public java.lang.String name;
    public org.omg.CORBA.TypeCode type; public IDLType type_def; public
ParameterMode
    mode;
    // constructors
    public ParameterDescription(
        java.lang.String _name,
        org.omg.CORBA.TypeCode _type,
        IDLType _type_def,
        ParameterMode _mode
    ) {
        name = _name;
        type = _type;
        type_def = _type_def;
        mode = _mode;
    }
    public ParameterDescription() {}
}
```

org.omg.CORBA.OperationDescription クラスのメンバ exceptions が null 以外ならば、exceptions からユーザ定義例外情報を取り出します。

exceptions は org.omg.CORBA.ExceptionDescription クラスの配列です。
org.omg.CORBA.ExceptionDescription クラスの定義を以下に示します。

```
package org.omg.CORBA;

final public class ExceptionDescription {
    /* instance variables*/
    public java.lang.String name;
    public java.lang.String id;
    public java.lang.String defined_in;
    public java.lang.String version;
    public org.omg.CORBA.TypeCode type;
    /* constructors*/
    public ExceptionDescription(
        java.lang.String _name,
        java.lang.String _id,
        java.lang.String _defined_in,
        java.lang.String _version,
        org.omg.CORBA.TypeCode _type
    ) {
        name = _name;
        id = _id;
        defined_in = _defined_in;
        version = _version;
        type = _type;
    }

    public ExceptionDescription() {}
}
```

これらのインタフェース情報により、DII による呼び出しに必要なパラメータを組み立てることができます。

7.2.7.5. Dynamic Invocation Interface (DII)

DII を使うと、静的なオペレーション情報がクライアント側にリンクされていなくても、オペレーションを動的に呼び出すことができます。DII はアプリケーション作成時に呼び出すインタフェースが決まっていないような特殊なアプリケーションを作成する場合にだけ使います。このようなアプリケーションの例としては、インタフェースの情報を動的に与えてオブジェクトをテストする汎用のテストツールなどが考えられます。

基本的な呼び出し

DII を使って呼び出す手順として、引数リストなどを先に作っておき org.omg.CORBA.Request オブジェクトを作成する方法と、org.omg.CORBA.Request オブジェクトを先に作っておく方法があります。

先に引数リストを作成するには org.omg.CORBA.ORB.create_list か org.omg.CORBA.ORB.create_operation_list を使います。前者は org.omg.CORBA.NVList.add などを使って引数を 1 つずつ追加していかなければいけません。後者はインタフェースリポジトリに登録されているインタフェース情報をもとに自動的に追加されます。

org.omg.CORBA.Request オブジェクトを作成するには、org.omg.CORBA._request または org.omg.CORBA._create_request を使います。org.omg.CORBA._create_request は引数リストに null を設定しておいて、後から引数を追加することもできます。

作成した org.omg.CORBA.Request に引数を追加するには、org.omg.CORBA.Request.arguments で引数リストを取り出す方法と、org.omg.CORBA.Request.add_in_arg などを使う方法があります。以下の例では後者を使います。引数リストを直接操作する方法は「引数リストを表すインタフェース」を参照してください。

以下ではもっとも基本的な呼び出し手順を示します。

DII によるオペレーションの呼び出しを行うには、org.omg.CORBA.Request オブジェクトにオペレーションの

情報(引数や戻り値の型など)をセットする必要があります。

org.omg.CORBA.Request オブジェクトを作成するには、呼び出すオブジェクトのオブジェクトリファレンスが必要です。オブジェクトリファレンスの取得方法は静的な呼び出しと同じです。不特定のオブジェクトを呼び出すクライアントを作成する場合、名前サービスの org.omg.CosNaming.NamingContext.resolve を使うのが一般的です。

また、不特定のオブジェクトを呼び出すクライアントを作成する場合は、インタフェースの定義を取得する必要があります。WebOTX Object Broker Java™ ではインタフェース定義を取得するには、org.omg.CORBA._get_interface_def を使います。org.omg.CORBA._get_interface_def()はあらかじめインタフェースリポジトリに登録されていたインタフェースを返します。

```
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class client {
    public static void main(String args[]) {
        try {
            // 何らかの手段でオブジェクトリファレンスを取得
            org.omg.CORBA.Object obj = ...;

            // インタフェースを定義しているオブジェクトの取得
            InterfaceDef intfobj =
                org.omg.CORBA.InterfaceDefHelper.narrow(obj._get_interface())
;
            // インタフェース情報を取り出す
            org.omg.CORBA.InterfaceDefPackage.FullInterfaceDescription
                intfdesc = intfobj.describe_interface();
                    :
                    :
```

intfdesc にはオブジェクトリファレンスが指し示すオブジェクトのすべてのインタフェース情報が入っています。この中から、目的のオペレーションを選びます。

```
int count = intfdesc.operations.length;
OperationDescription opedesc = new OperationDescription();

for (int i = 0; i < count; i++) {
    // オペレーションのシーケンスから 1 つ取り出す
    opedesc = intfdesc.operations[i];

    // オペレーション名が一致するものを探す(仮に"myop"を探すことにします)
    if (opedesc.name.equals("myop")) {
        break;
    }
}
```

名前が一致するオペレーションを取得できたら、org.omg.CORBA.Request オブジェクトを作成します。引数には、オペレーション名を渡します。

```
Request req = obj._request(opedesc.name);
```

作成した org.omg.CORBA.Request オブジェクトに戻り値の型を指定します。

```
req.set_return_type(opedesc.result);
```

オペレーションの引数を登録します。引数には in, out, inout の 3 種類があり、それぞれ別のメソッド(add_XXX_arg)が用意されています。引数はメソッドを呼び出すたびに引数リストの後ろへ追加されます。add_XXX_arg は、org.omg.CORBA.Any を返します。返された org.omg.CORBA.Any に引数を挿入します。

```
count = opedesc.parameters.length;

ParameterDescription paradesc;
Any value;

for (int i = 0; i < count; i++) {
    // パラメータのシーケンスから 1 つ取り出す
    paradesc = opedesc.parameters[i];

    // パラメータの設定
    switch (paradesc.mode.value()){
    case ParameterMode._PARAM_IN:
        switch (paradesc.type.kind().value()) {
        case TCKind._tk_short:
            short in_param = ...;
            req.add_named_in_arg(paradesc.name).insert_short(in_param);
            break;
        ...
        }
        break;
    case ParameterMode._PARAM_INOUT:
        switch (paradesc.type.kind().value()) {
        case TCKind._tk_short:
            short inout_param = ...;
            req.add_named_inout_arg(paradesc.name).insert_short(inout_param);
            break;
        ...
        }
        break;
    default:
        req.add_named_out_arg(paradesc.name);
        break;
    }
}
```

引数の組み立てが終わったら、サーバの呼び出しを行います。サーバの呼び出し方法には単方向と双方向の 2 種類があります。これらは IDL 定義をしたときに決まります。

```
if (opedesc.mode == OperationMode.OP_ONEWAY) {
    req.send_oneway();  // 単方向呼び出しのとき
}
else {
    req.invoke();       // 双方向呼び出しのとき
}
```

双方向呼び出しで、かつ、戻り値を持っていた場合は、戻り値を取得する必要があります。戻り値は、org.omg.CORBA.Any 型で取得します。

```
if (opedesc.result.kind() != TCKind.tk_void) {
    switch(opedesc.result.kind().value()) {
    case TCKind._tk_short:
        short result = req.return_value().extract_short();
        break;
    ...
    }
}
```

リクエストの発行

リクエストの発行には、4 つの方法があります。

- 同期オペレーション

  応答を要求する呼び出しです。サーバへリクエストを送信し、応答が返るまで待ちあわせます。

  前述の「基本的な呼び出し」の例のとおり、org.omg.CORBA.Request.invoke()を使用します。

- oneway オペレーション

  単方向の呼び出しです。サーバへリクエストを送信するだけで、応答を期待しません。IDL で oneway と定義したオペレーションの呼び出しに使います。

  前述の「基本的な呼び出し」の例のとおり、org.omg.CORBA.Request.send_oneway()を使用します。

- 遅延同期オペレーション

  送信と受信を分ける呼び出しです。送信後、応答が返るのを待たずにクライアント側へ処理が戻ります。応答の取得は任意のタイミングでポーリングまたは待ち合わせをします。

  遅延同期オペレーションでは、org.omg.CORBA.Request.send_deferred()による送信の発行後、すぐに制御が戻ります。任意の時点で、org.omg.CORBA.Request.poll_response()を呼び出してポーリングするか org.omg.CORBA.Request.get_response()を呼び出して待ち合わせます。org.omg.CORBA.Request.poll_response()は単に応答メッセージが受信可能かどうかを調べるだけです。応答メッセージを取得するには org.omg.CORBA.Request.get_response()を呼び出す必要があります。

- 複数リクエストの発行

  複数のリクエストを同時に発行する呼び出しです。リクエストのシーケンスを作成しておき、複数のリクエストを順番に発行することができます。oneway かどうかで使用する関数が異なります。

  サーバからの応答を必要とする呼び出しの場合には、org.omg.CORBA.ORB.send_multiple_requests_deferred()を使います。oneway の呼び出しの場合は org.omg.CORBA.ORB.send_multiple_requests_oneway()を使います。

  個々のリクエストについて応答メッセージを調べるには遅延同期オペレーションのときと同様に org.omg.CORBA.Request.poll_response()と org.omg.CORBA.Request.get_response()を使います。また、複数のメッセージのうちのどれか 1 つ以上という場合は org.omg.CORBA.ORB.poll_next_response()と org.omg.CORBA.ORB.get_next_response()を使います。

7.2.7.6. Dynamic Skeleton Interface (DSI)

DSI を使うと、静的なオペレーション情報がサーバ側にリンクされていなくとも、オペレーションを動的に呼び出すことができます。

DII は org.omg.CORBA.Request オブジェクトにオペレーションの情報(引数や戻り型など)をセットし、org.omg.CORBA.Request.invoke()を呼び出すことにより、オブジェクト呼び出しの入口を共通化しています。

DSI は、org.omg.PortableServer.DynamicImplementation.invoke()を共通の入口としています。オペレーションの情報は org.omg.CORBA.ServerRequest オブジェクトに入っているという点が DII と異なります。

DSI に必要なクラス

DSIを利用するためには org.omg.PortableServer.DynamicImplementation クラスを継承したクラスを作成します。このクラスがリクエストの処理を実装する DSI サーバント(DSI の実装オブジェクト)になります。DSI サーバントは、次のメソッドをオーバライドする必要があります。

```
package org.omg.PortableServer;
abstract public class Servant {
     ・・・
    // methods for which the skeleton or application
    // programmer must provide an an implementation
    abstract public String[] _all_interfaces(POA poa, byte[] objectId);
     ・・・
}
```

```
package org.omg.PortableServer;
abstract public class DynamicImplementation extends Servant {
     abstract public void invoke(org.omg.CORBA.ServerRequest request);
}
```

このクラスでは、各オペレーションのほかに invoke メソッドと_all_interfaces メソッドの実装が必要です。

・ Invode メソッド

invoke メソッドは、オペレーションが呼び出されるとき、ランタイムライブラリから呼ばれる、ユーザコードへの入口となります。invoke メソッドの処理は自由に記述することができます。

オペレーションの情報は request から取り出すことができます。呼び出されたオブジェクトのオブジェクトリファレンスは、org.omg.PortableServer.DynamicImplementation の_this_object メソッドを使って取得できます。

後述の例では、DSI サーバントを複数生成して、インスタンスにより処理を切り換えています。ただし、エラー処理は省略してあります。

・ _all_interface メソッド

DSI サーバントがあつかうインタフェースを取り出すために ORB から呼び出されます。ORB は、IOR を生成するためや、クライアントからの_is_a オペレーションの要求に応答する際に呼び出します。引数には、呼び出しのきっかけとなったオブジェクトのオブジェクト ID とそのオブジェクトに対応づけられた POA が渡されます。DSI サーバントが実装するインタフェースのリポジトリ ID の配列を返却しなければなりません。継承する上位の interface も含めた配列の最初の要素には最下位の interface を指定してください。DSI サーバントが複数のインタフェースをあつかう場合、引数により適切なインタフェースを返すように記述します。

以下に、例を示します。

```
interface MyInterface {
    long MyOperation(in long arg);
};
```

```
public class MyDynamicImpl extends org.omg.PortableServer.DynamicImplementation
{
    public MyDynamicImpl() {}
    private int MyOperation1(int a) {
        return a+1;
    }
    private int MyOperation2(int a) {
        return a+2;
    }

    String[] _ids = {"IDL:MyInterface:1.0"};
    public String[] _all_interfaces(org.omg.PortableServer.POA poa,
                                    byte[] objectId) {
        return _ids;
    }

    public void invoke(org.omg.CORBA.ServerRequest request) {
        try {
            org.omg.CORBA.ORB orb = _orb();
            org.omg.CORBA.Any rslt = orb.create_any();
            // どのオペレーションが呼ばれたか調べる
            String op_name = request.operation();
            if ("MyOperation".equals(op_name)) {
                // 引数のデコードに必要な情報をセットする
                org.omg.CORBA.NVList arg = orb.create_list(1);
                org.omg.CORBA.Any param =
                    arg.add(org.omg.CORBA.ARG_IN.value).value();
                param.type(orb.get_primitive_tc(org.omg.CORBA.TCKind.tk_long)
);
                // 引数をデコードする
                request.arguments(arg);
                // IN 引数を取り出す
                int a = param.extract_long();
                // オペレーションを呼び出す
                org.omg.CORBA.IntHolder ret = new org.omg.CORBA.IntHolder();
                // オブジェクト ID によりオペレーションを切り換える
                if ("myimpl1".equals(new String(_object_id())))
                    ret.value = MyOperation1(a);
                else
                    ret.value = MyOperation2(a);
                // 戻り値を request にセットする
                rslt.type(orb.get_primitive_tc(org.omg.CORBA.TCKind.tk_long))
;
                rslt.insert_long(ret.value);
                request.set_result(rslt);
                return;
            }
            // エラー処理または例外を request にセットする
            // org.omg.CORBA.BAD_OPERATION 例外を request にセットする場合
            org.omg.CORBA.Any exp = _orb().create_any();
            org.omg.CORBA.BAD_OPERATIONHelper.insert(
                exp, new org.omg.CORBA.BAD_OPERATION(
                    0, org.omg.CORBA.CompletionStatus.COMPLETED_MAYBE));
        } catch (Exception e) {
            // エラー処理または例外を request にセットする
            e.printStackTrace();
        }
    }

    public static void main(String[]args) {
        try {
```

```
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
            org.omg.CORBA.Object o =
orb.resolve_initial_references("RootPOA");
            org.omg.PortableServer.POA rootPOA =
                org.omg.PortableServer.POAHelper.narrow(o);
            org.omg.CORBA.Policy policies[] = new org.omg.CORBA.Policy[1];
            policies[0] = rootPOA.create_id_assignment_policy(
                org.omg.PortableServer.IdAssignmentPolicyValue.USER_ID);
            org.omg.PortableServer.POA myPOA =
                rootPOA.create_POA("myPOA", null, policies);
            // 文字列をオブジェクト ID に変換
            byte[] oid1 = new String("myimpl1").getBytes();
            byte[] oid2 = new String("myimpl2").getBytes();
            // MyOperation1,MyOperation2 を呼び出すインスタンスを作成
            MyDynamicImpl impl1 = new MyDynamicImpl();
            MyDynamicImpl impl2 = new MyDynamicImpl();
            // サーバントを活性化
            myPOA.activate_object_with_id(oid1, impl1);
            myPOA.activate_object_with_id(oid2, impl2);
            // オブジェクトリファレンスを名前サーバに登録
            org.omg.CORBA.Object obj1 = myPOA.servant_to_reference(impl1);
            org.omg.CORBA.Object obj2 = myPOA.servant_to_reference(impl2);
            org.omg.CORBA.Object nsv =
                orb.resolve_initial_references("NameService");
            org.omg.CosNaming.NamingContext ns =
                org.omg.CosNaming.NamingContextHelper.narrow(nsv);
            org.omg.CosNaming.NameComponent ncseq1[] =
                {new org.omg.CosNaming.NameComponent("implobj1", "")};
            org.omg.CosNaming.NameComponent ncseq2[] =
                {new org.omg.CosNaming.NameComponent("implobj2", "")};
            ns.rebind(ncseq1, obj1);
            ns.rebind(ncseq2, obj2);
            // POA マネージャの活性化
            myPOA.the_POAManager().activate();
            orb.run();
        } catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

7.2.7.7. サーバ処理のスレッドポリシー選択

Object Broker Java は、サーバ処理を実行するスレッドの動作を指定するスレッドポリシーの機能を提供しています。次のポリシーから選択することができます。

- PerClientThread

  クライアント単位にスレッドを生成し、クライアントからの要求をそのスレッド上でシリアルに実行する。

- PooledThread

  あらかじめ複数のスレッドをプールし、クライアントからの要求をプールされたスレッドに分配してパラレルに実行する。

スレッドポリシーは、サーバプログラムの ThreadPolicy プロパティに PerClientThread もしくは PooledThread を指定して設定します。設定しない場合の既定値は PerClientThread です。

また、PooledThread の場合のプール数を PooledThreadNumber プロパティで設定することができます。設定しない場合の既定値は 10 です。

設定方法は、本節の「Object Broker Java のプロパティ設定」の項を参照してください。

スレッドポリシーの設定は、POA の ThreadPolicy が ORB_CTRL_MODEL である場合のみ有効です。ポリシーに SINGLE_THREAD_MODEL が設定されている場合は、PooledThread であっても、単一のスレッドで順次処理されます。

PooledThread が選択されているとき、oneway オペレーション、遅延同期オペレーション、複数リクエスト発行を使用すると、必ずしもクライアントからの要求順序通りにサーバがで処理されません（順序性が保証されません）。

7.2.7.8. 文字コードセット

CORBA 呼び出しで文字列データを通信する場合に、サーバとクライアントとの間で、文字コードの意識が合っていることが重要です。

文字コードセットは、ネイティブコードセットとコンバージョンコードセットの 2 つに分けられます。

ネイティブコードセット	ORB がプログラム内部で取り扱う文字コードを示します。通常は変更することはできません。
コンバージョンコードセット	ネイティブコードセットに変換することが可能な文字コードを示します。

Object Broker Java では、以下のように設定しています。

	char 型、string 型	wchar 型、wstring 型
ネイティブコードセット	UCS2L1	UCS2L1
コンバージョンコードセット	OSF_SJIS1 JIS_eucJP ISO8859-1 ISO646	UTF16

コードセットのネゴシエーション機能

CORBA 仕様では、クライアントとサーバの間で文字および文字列のデータを受渡しするときに、どの文字コードセットを使用するかについて意識を合わせるためのネゴシエーション機能が規定されています。

コードセットのネゴシエーション機能では、おもに次の処理をおこないます。

1. サーバ側の ORB は、通信に使用することができるコードセットをオブジェクトリファレンスに設定します。
2. クライアント側の ORB は、オブジェクトリファレンスに設定されたコードセットが使用可能であれば、そのコードセットで通信を行います。

コードセットのネゴシエーション機能を使用するためには、IIOP および GIOP のバージョンが 1.1 以上である必要があります（Object Broker は 1.1 もしくは 1.2 で動作しますので、通常は指定する必要はありません）。IIOP および GIOP のバージョンが 1.0 である場合は、char 型、string 型は ISO8859-1 を、wchar 型、wstring 型は UCS2L1 を使用して通信します。

コンバージョンコードセットの変更

コンバージョンコードセットは、ConversionCodeSets プロパティ、ConversionCodeSetsW プロパティを設定することによって変更することができます。

既定値の設定で通信をおこなうことが可能ですので、通常は設定を変更する必要はありません。

char 型、string 型のコンバージョンコードセットで指定可能なコードセットは下表のとおりです。**太字**は指定を変更しなかった場合の既定値です。UCS2L1 はネイティブコードセットに設定されています。

コードセット名	コードセット番号	説明	エンコーディング
ISO8859-1	0x00010001	ISO 8859-1:1987; Latin Alphabet No. 1	――――
ISO8859-2	0x00010002	ISO 8859-2:1987; Latin Alphabet No. 2	8859_2
ISO8859-3	0x00010003	ISO 8859-3:1988; Latin Alphabet No. 3	8859_3
ISO8859-4	0x00010004	ISO 8859-4:1988; Latin Alphabet No. 4	8859_3
ISO8859-5	0x00010005	ISO/IEC 8859-5:1988; Latin-Cyrillic Alphabet	8859_5
ISO8859-6	0x00010006	ISO 8859-6:1987; Latin-Arabic Alphabet	8859_6
ISO8859-7	0x00010007	ISO 8859-7:1987; Latin-Greek Alphabet	8859_7
ISO8859-8	0x00010008	ISO 8859-8:1988; Latin-Hebrew Alphabet	8859_8
ISO8859-9	0x00010009	ISO/IEC 8859-9:1989; Latin Alphabet No. 5	8859_9
ISO646	0x00010020	ISO 646:1991 IRV (International Reference Version)	――――
UCS2L1	0x00010100	ISO/IEC 10646-1:1993; UCS-2, Level 1	――――
JIS0201	0x00030001	JIS X0201:1976; Japanese phonetic characters	JIS
JIS0208:1978	0x00030004	JIS X0208:1978 Japanese Kanji Graphic Characters	JIS
JIS0208:1983	0x00030005	JIS X0208:1983 Japanese Kanji Graphic Characters	JIS
JIS0208	0x00030006	JIS X0208:1990 Japanese Kanji Graphic Characters	JIS
JIS0212	0x0003000a	JIS X0212:1990; Supplementary Japanese Kanji Graphic Chars	JIS
JIS_eucJP	0x00030010	JIS eucJP:1993; Japanese EUC	EUCJIS
OSF_UJIS	0x05000010	OSF Japanese UJIS	EUCJIS
OSF_SJIS1	0x05000011	OSF Japanese SJIS-1	SJIS
OSF_SJIS2	0x05000012	OSF Japanese SJIS-2	SJIS
UTF8	0x05010001	X/Open UTF-8; UCS Transformation Format 8 (UTF-8)	――――
JVC_eucJP	0x05020001	JVC_eucJP	EUCJIS
JVC_SJIS	0x05020002	JVC_SJIS	SJIS
Cp437	0x100201b5	IBM-437 (CCSID 00437); PC USA	Cp437
Cp850	0x10020352	IBM-850 (CCSID 00850); Multilingual IBM PC Data-MLP 222	Cp850
Cp852	0x10020354	IBM-852 (CCSID 00852); Multilingual Latin-2	Cp852
Cp855	0x10020357	IBM-855 (CCSID 00855); Cyrillic PC Data	Cp855
Cp857	0x10020359	IBM-857 (CCSID 00857); Turkish Latin-5 PC Data	Cp857
Cp861	0x1002035d	IBM-861 (CCSID 00861); PC Data Iceland	Cp861
Cp862	0x1002035e	IBM-862 (CCSID 00862); PC Data Hebrew	Cp862
Cp863	0x1002035f	IBM-863 (CCSID 00863); PC Data Canadian French	Cp863
Cp864	0x10020360	IBM-864 (CCSID 00864); Arabic PC Data	Cp864
Cp866	0x10020362	IBM-866 (CCSID 00866); PC Data Cyrillic 2	Cp866
Cp869	0x10020365	IBM-869 (CCSID 00869); Greek PC Data	Cp869

Cp874	0x1002036a	IBM-874 (CCSID 00874); Thai PC Display Extended SBCS	Cp874
Cp932	0x100203a4	IBM-932 (CCSID 00932); Japanese PC Data Mixed	MS932
Cp1250	0x100204e2	IBM-1250 (CCSID 01250); MS Windows Latin-2	Cp1250
Cp1251	0x100204e3	IBM-1251 (CCSID 01251); MS Windows Cyrillic	Cp1251
Cp1252	0x100204e4	IBM-1252 (CCSID 01252); MS Windows Latin-1	Cp1252
Cp1253	0x100204e5	IBM-1253 (CCSID 01253); MS Windows Greek	Cp1253
Cp1254	0x100204e6	IBM-1254 (CCSID 01254); MS Windows Turkey	Cp1254
Cp1255	0x100204e7	IBM-1255 (CCSID 01255); MS Windows Hebrew	Cp1255
Cp1256	0x100204e8	IBM-1256 (CCSID 01256); MS Windows Arabic	Cp1256
Cp1257	0x100204e9	IBM-1257 (CCSID 01257); MS Windows Baltic	Cp1257

wchar 型、wstring 型で指定可能なコードセットは以下のとおりです。**太字**は指定を変更しなかった場合の既定値です。UCS2L1 はネイティブコードセットに設定されています。

コードセット名	コードセット番号	説明	エンコーディング
UCS2L1	0x00010100	ISO/IEC 10646-1:1993; UCS-2, Level 1	――――
UTF16	0x00010109	ISO/IEC 10646-1:1993; UTF-16, UCS Transformation Format 16-bit form	

コードセットとエンコーディング

Java VM は、内部的な文字コードとして Unicode を用います。したがって、通信で使用されている文字コードが Unicode ではない場合、文字コードを変換する必要があります。

通信で使用されるコードセットが UCS2L1, ISO8859-1, ISO646, UTF-8, UTF-16 の場合は、Object Broker Java 独自の機能によってコード変換をおこないます。それ以外のコードセットが通信に使用された場合は、Java の機能によってコード変換をおこないます。

それぞれの文字コードセットに対応するエンコーディングは上表のとおりです。また、CodeSetEncoding プロパティを設定することによって文字コードセットに対応するエンコーディング名を変更することができます。

文字化けの発生について

サーバとクライアントのうち、一方が C++、もう一方が Java である場合、通常はコードセットのネゴシエーションによって通信に使用されるコードセットに OSF_SJIS1（シフト JIS）が選択されます。

このとき、既定値では、シフト JIS と Unicode の変換のためのエンコーディングに SJIS を使用しますが、下表に示す文字で文字化けが発生する場合があります。文字化けが発生する文字が以下に該当する場合は、CodeSetEncoding プロパティに「OSF_SJIS1=MS932」と設定してください。

文字	シフト JIS でのコード
～	0x8160
‖	0x8161
―	0x817C
￠	0x8191
￡	0x8192
￢	0x81CA
NEC 特殊文字	0x8740 ～ 0x87FC
IBM 拡張文字（NEC 選定）	0xED40 ～ 0xEEFC
ユーザ定義文字（外字領域）	0xF040 ～ 0xF9FC
IBM 拡張文字	0xFA40 ～ 0xFC4B

7.2.7.9. コードベース

valuetype の実装クラスおよび ValueFactory の実装クラスを Web サーバからダウンロードする機能について説明します。

コードベースダウンロード機能

CORBA では、valuetype に関する情報をリモートの環境から取得するための **SendingContextRunTime** サービスが規定されています。

valuetype の実装クラスおよび ValueFactory の実装クラスをローカルな環境に持っていないとき、HTTP を利用して Web サーバからダウンロードする機能をコードベースダウンロード機能と呼びます。実装クラスをダウンロードするための URL のことを**コードベース**と呼びます。

通信する相手の ORB からコードベースを取得し、valuetype の実装クラスおよび ValueFactory の実装クラスをダウンロードして、valuetype のインスタンスを作成する処理は Object Broker Java が自動的におこないますので、アプリケーションで意識してプログラミングする必要はありません。

コードベースダウンロード機能を使用するためには、valuetype の送信側の ORB で、この機能を有効にするための設定が必要です。

- インタフェースの引数に valuetype を使用し、in 属性を持つ場合は、クライアント側の ORB に対して設定が必要です。
- インタフェースの引数に valuetype を使用し、out 属性を持つ場合は、サーバ側の ORB に対して設定が必要です。
- インタフェースの引数に valuetype を使用し、inout 属性を持つ場合は、クライアント側とサーバ側の両方の ORB に対して設定が必要です。
- インタフェースの戻り値として valuetype を使用する場合は、サーバ側の ORB に対して設定が必要です。

コードベースダウンロード機能を有効にするための設定には以下の 3 つがあります。

- UseCodeBase プロパティに true を設定する。
- コードベースプロパティファイルを作成し、クラスファイルをダウンロードすることができる URL を記述する。
- アプリケーションのプログラムでオブジェクト ID（クライアントでは ORB)とコードベースプロパティ名を登録する。

コードベースの設定単位

コードベースの設定は、1 つのプロセスの中で複数の設定を使い分けることができます。

サーバ側は、オブジェクト ID 毎に設定します。コードベースの設定は、そのオブジェクト ID に対応するサーバントにのみ有効になります。オブジェクト ID は POA.activate_object()の戻り値として取得できます。

クライアント側は、ORB 毎に設定します。コードベースの設定は、その ORB に対応するリクエストにのみ有効です。ORB は ORB.init()の戻り値として取得できます。

アプリケーションのプログラムでオブジェクト ID(クライアントでは ORB)と、設定を区別するための**コードベースプロパティ名**を登録する必要があります。

登録は Config.setCodeBaseIdent()を呼び出すことによっておこないます。登録した時点でコードベースプロパティ名に対応するコードベースプロパティファイルに記述された URL からダウンロードが可能になります。

異なるオブジェクト ID(クライアントでは ORB)に対して同じコードベースプロパティ名を登録すれば、同じ設定を使用することができます。

プログラム内でコードベースプロパティ名の登録をしないオブジェクト ID(クライアントでは ORB)については、プロセスで共通の設定を使用することができます。

コードベースプロパティファイル

コードベースプロパティファイルは、valuetype の実装クラスおよび ValueFactory の実装クラスをダウンロードするために、そのクラスファイルが置かれている場所を示す URL を記述するファイルです。

ただし、コードベースプロパティファイルには ValueFactory の実装クラスファイルの URL のみを記述します。ValueFactory の実装クラスで valuetype の実装クラスを使用していれば、そのクラスファイルも同時にダウンロードされますので、valuetype の実装クラスファイルの URL を記述する必要はありません。

・ クラスファイルを直接ダウンロードする場合

クラスファイルが格納されているディレクトリを参照できる URL を記述してください。

ValueFactory の実装クラスが DefaultFactory である場合は、クラスファイル名を URL の記述から省略することができます。

例） http://host1/java/class

・ JAR 形式でダウンロードする場合

クラスファイルを JAR 形式にまとめた場合、JAR ファイル名を含む形式で、次の例のように記述してください。

例） jar:http://www.foo.co.jp/java/class/calc.jar!/

コードベースプロパティファイルは、テキストファイルとして作成します。行単位に複数のアイテムを空白もしくはタブで区切って記述します。構文は以下のとおりです。

```
* <URL> ... <URL>
<Repository-ID> <URL> ... <URL>
    :
    :
<Implementation-class> <URL> ... <URL>
    :
    :
```

'*' で始まる行は、デフォルトの URL を記述します。<Repository-ID> での指定も <Implementation-class>での指定もない場合、デフォルトの URL から検索します。

いずれの指定でも URL を複数書くことができます。記述順に検索してダウンロードできた場合はそのクラスを使用します。

コードベースプロパティ名に対応するコードベースプロパティファイルのファイル名は、以下のようになります。

<コードベースプロパティ名>.cbp

ファイルを格納するディレクトリはカレントディレクトリ（アプリケーションを実行したディレクトリ）になります。このディレクトリは CodeBasePropertyFileDir プロパティで変更できます。

プロセスで共通のコードベースプロパティファイルのファイル名は、以下のようになります。

カレントディレクトリ/codebase.conf

このファイル名は CodeBasePropertyFile プロパティでディレクトリを含めた形で変更できます。(この指定は CodeBasePropertyFileDir プロパティより優先されます)

Object Broker Java では、コードベースダウンロード機能を使用する場合、アプリケーションでセキュリティマネージャが設定されていなければ、RMISecurityManager を設定して使用します(アプレットではブラウザによってセキュリティマネージャが設定されます)。

そのため、アクセス権の設定をポリシーファイルに記述する必要があります。

アプリケーションの場合:

```
grant {
    permission java.util.PropertyPermission "*", "read, write";
    permission java.io.FilePermission "${user.home}¥¥orb.properties", "read";
    permission java.io.FilePermission "${java.home}¥¥lib¥¥orb.properties", "read";
    permission java.lang.RuntimePermission "getClassLoader";
    permission java.io.FilePermission "<コードベースプロパティファイル名>", "read";
    permission java.net.SocketPermission "<Web サーバの IP アドレス>[:<Web サーバのポート番号
>]", "connect, resolve";
    permission java.net.SocketPermission "<サーバプロセスが動作するマシン名>[:<ポート番号
>]", "connect, resolve";
    permission java.net.SocketPermission "<サーバ AP が動作するマシン名>[:<ポート番号>]",
"connect, resolve";
    permission java.net.SocketPermission "<クライアント AP が動作するマシン名>[:<ポート番号
>]", "accept, resolve";
};
```

アプレットの場合:

```
grant {
    permission java.io.FilePermission "<コードベースプロパティファイル名>", "read";
    permission java.net.SocketPermission "<Web サーバの IP アドレス>[:<Web サーバのポート番号
>]", "connect, resolve";
};
```

7.2.7.10. クライアント側コネクションの強制切断

クライアントとサーバとの間で確立されたコネクションは要求が完了した後も再利用のために保持しています。システム構成によっては、利用頻度の低いコネクションが保持されたままになり、システム資源を無駄に消費しているともいえます。これを解消するために、以下のメソッドを提供します。このメソッドは jp.co.nec.orb.ConnectionManager クラスに定義されています。

- 指定したオブジェクトに対応するコネクションを切断

```
public static void close_client_connection(org.omg.CORBA.Object obj);
```

- すべてのコネクションを切断

```
public static void close_all_client_connections();
```

処理中の要求が存在する場合には、その完了を待ち合わせて切断します。他のスレッドからの要求が破棄されることはありません。その待ち合わせ中に新たな要求を受け付けた場合にもさらに待ち合わせします。

また、コネクションが確立されているか否かを確認するために、以下のメソッドを提供します。このメソッドは jp.co.nec.orb.ConnectionManager クラスに定義されています。

- オブジェクトに対応するコネクションの状態を確認

```
public static boolean is_connected(org.omg.CORBA.Object obj);
```

- すべてのコネクションの状態を確認

```
public static boolean is_connected_any();
```

7.2.7.11. アプレットプロキシ

Java アプレットでは、次のような制限があります。

- アプレットはサーバとして動作することができません。アプレットがサーバとして動作するためには、accept による TCP/IP 接続ができる必要があります。しかし、アプレットはセキュリティに関わる理由により accept による TCP/IP 接続を許していません。
- アプレットから呼び出しを実行できるオブジェクトは、Web サーバ上で動作するものに限られます。アプレットは、アプレットが転送されたホスト以外との接続が許可されていないためです。

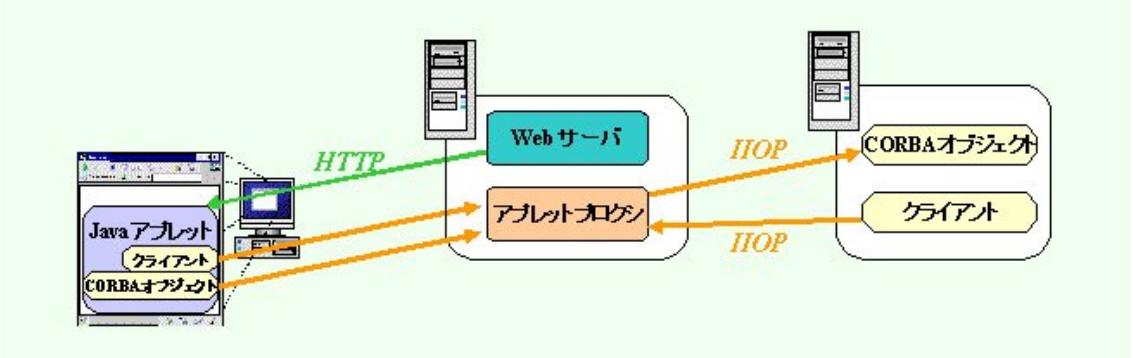
この制限を回避するために、アプレットプロキシという代理サーバを用意しています。Enterprise Edition もしくはオプション製品である Object Broker Java でのみ提供される機能です。プロキシプロトコルには SOCKS5 を使用しています。

アプレットプロキシは Web サーバと同じホストで動作させます。

たとえば、リアルタイム掲示板を考える場合、掲示板に書き込みが行われたときには、掲示板に接続中の各アプレットに対して画面の更新要求を出す必要があります。この場合、アプレットがサーバになる必要があります。前述の制限により、通常はアプレットがサーバとして動作することはできませんが、アプレットプロキシを使用することで可能になります。

また、次の図.のように、アプレットから Web サーバ以外のホストで動作しているオブジェクトに対して呼び出しを行うことも、アプレットプロキシを使うことで可能になります。

アプレットがサーバにならず、Web サーバホストで動作するオブジェクトへの呼び出しを行うだけであれば、アプレットプロキシを使用する必要はありません。



アプレットが Web サーバからダウンロードされたとき、そのアプレットに対してアプレットプロキシを使用するように設定されていれば、すべての IIOP 通信はアプレットプロキシ経由で行われます。 その場合、アプレット内のオブジェクトはサーバソケットを作成しません。

アプレットプロキシを使用するには、HTML ファイルの PARAM タグに

```
<PARAM NAME="AppletProxyHost" VALUE="ホスト名">
```

と指定する必要があります。ホスト名にはアプレットプロキシの動作しているホスト名を指定します。このホスト名は、アプレットがダウンロードされた Web サーバホスト名と同一のものを指定します(アプレットプロキシは Web サーバと同じホスト上で動作させなければなりません)。

アプレットプロキシを使用する時はサーバのポート番号の指定は無視されます。アプレットプロクシで SSL を使用する場合は、次のようにポート番号に 0 を指定してください。

```
<PARAM NAME="SSLPort" VALUE="0">
```

7.2.8.Object Broker が提供するサービス

インタオペラブル名前サービスの使用方法

インタオペラブル名前サービスとは、リモートホスト上の初期サービスのオブジェクトリファレンスを相互に取得/提供するための仕組みです。

インタオペラブル名前サービスには 2 つの仕様案が OMG に提出されていました。1 つは iioploc URL, iiopname URL で表される形式、もう 1 つは corbalocURL, corbaname URL で表される形式です。両者は細部を除いてよく似ています。旧製品 ObjectSpinner では、この 2 つの仕様をサポートしていました。

Object Broker は、この 2 つの仕様との互換性を保持したまま、CORBA2.4.2(ptc/01-02-01)で正式に取り込まれた仕様に準拠しています。CORBA2.4.2 では corbalocURL/corbanameURL の形式が採択され、rir プロトコルが追加されました。

オブジェクトリファレンスを文字列で表わす形式として、従来使われていた IOR 形式のほかに、iioploc(corbaloc) URL 形式と iiopname (corbaname) URL 形式が追加されました。

- iioploc(corbaloc) URL 形式

  オブジェクトキーに対応したオブジェクトリファレンスを返すサーバ(以降、corbaloc サーバ)を利用するための文字列表現です。iioplocURL はオブジェクトキーのほかに corbaloc サーバのホスト名やポート番号などの情報も含みます。

- iiopname(corbaname) URL 形式

  iioploc URL 形式で表わされる CosNaming::NamingContext オブジェクトに加え、名前サービスに登録されているオブジェクトも表わすことができます。

iioploc URL 形式と iiopname URL 形式

iioploc URL 形式と iiopname URL 形式のシンタックスを説明します。

```
<iioploc> = "iioploc://"[<addr_list>]["/"<key_string>]
<iiopname> = "iiopname://"[<addr_list>]["/"<string_name>]
<addr_list> = [<address>","]* <address>
<address> = [<version> <host> [":" <port>]]
<version> = <major> "." <minor> "@" | ""
<major> = IIOP の major バージョン番号
<minor> = IIOP の minor バージョン番号
<host> = ホスト名 | IP アドレス (未指定時は localhost)
<port> = corbaloc サーバのポート番号 (未指定時は 2809)
<key_string> = 文字列化したオブジェクトキー | ""
<string_name> = 文字列化した名前シーケンス
```

corbaloc URL 形式と corbaname URL 形式

corbaloc URL 形式と corbaname URL 形式のシンタックスを説明します。

```
<corbaloc> = "corbaloc:"<obj_addr_list>"/"<key_string>
<corbaname> = "corbaname:"<corbaloc_obj>["#"<string_name>]
<corbaloc_obj> = <obj_addr_list>["/"<key_string>]
<obj_addr_list> = [<obj_addr>","]* <obj_addr>
<obj_addr> = <prot_addr> | <future_prot_addr>
<prot_addr> = <rir_prot_addr> | <iiop_prot_addr>
<iiop_prot_addr> = <iiop_id><iiop_addr>
<iiop_id> = <iiop_default> | <iiop_prot_token>":" ["//"]
<iiop_default> = ":"
<iiop_prot_token> = "iiop"
<iiop_addr> = [<version> <host> [":" <port>]]
<version> = <major> "." <minor> "@" | ""
<major> = IIOP の major バージョン番号
<minor> = IIOP の minor バージョン番号
<host> = ホスト名 | IP アドレス (未指定時は localhost)
<port> = corbaloc サーバのポート番号 (未指定時は 2809)
<rir_prot_addr> = <rir_prot_token>":"
<rir_prot_token> = "rir"
<key_string> = 文字列化したオブジェクトキー | ""
<string_name> = 文字列化した名前シーケンス
```

初期サービスのオブジェクト指定

初期サービスのオブジェクトは、ORBInitRef プロパティで指定します。

「<ObjectID>=<ObjectURL>」の形式で指定してください。

- ObjectID

  "NameService"などの初期サービスのオブジェクト ID を指定します。

- ObjectURL

  オブジェクトを表わす URL を IOR 形式、iioploc 形式、iiopname 形式で指定します。

オブジェクト URL の省略部分の指定

オブジェクト URL の省略部分は、ORBDefaultInitRef プロパティで指定します。

org.omg.CORBA.ORB.resolve_initial_references()が、指定された内容を補完した URL が表すオブジェクトリファレンスを返します。

WebOTX Object Broker がサポートしているオブジェクトキー

iioploc URL や iiopname URL に指定するオブジェクトキーとして WebOTX Object Broker がサポートしているオブジェクトキーは以下のとおりです。

オブジェクトキー	オブジェクト
"NameService"	そのホスト上の名前サーバのルートコンテキスト
"InterfaceRepository"	そのホスト上の IR サーバの Repository オブジェクト

(注) iiopname URL では、オブジェクトキーを指定できません。"NameService"に固定されます。

7.2.9.アプリケーション設計/コーディング時の秘訣

この章では、CORBA を使用したアプリケーションを作成するときに気をつけなければならないこと、デバッグ時のためにあらかじめ注意しておいた方が良いことについて説明します。

インタフェースの設計

アプリケーションを設計するときは、ネットワークを使用した通信(つまり CORBA を使用したメソッド呼び出し)の数が最小限になるように設計してください。ネットワークの通信速度も高速になってはきていますが、プロセッサの速度と比較すると非常に遅いことにかわりがありません。CORBA を使用すると、ネットワークの低レベル API を意識せずにコーディングすることが可能ですが、そのために 逆にネットワークのオーバヘッドのことを忘れてしまうこともあるようです。アプリケーションでネットワークを使用した通信がどの程度行われるかは、IDL によるインタフェース定義の段階でほぼ確定してしまいます。CORBA を使用するということはネットワークによる通信を行うことだということを忘れないことが重要です。IDL の定義をアプリケーションの構築後に変更すると、後戻り工数が非常に大きくなります。IDL 定義の段階でネットワーク遅延を考慮したインタフェースを定義することが重要です。たとえば、サーバオブジェクトから情報を取り出したいとしたとき、次のような IDL 記述があるとします。

```
interface StateObject {

    attribute long state1;
    attribute long state2;
    attribute long state3;
    attribute long state4;

};
```

この IDL で、StateObject から、state1, state2, state3, state4 の全ての情報を得たいとすると次のようなプログラムになります。

```
//Java
long s1 = StateObjectObj.state1();
long s2 = StateObjectObj.state2();
long s3 = StateObjectObj.state3();
long s4 = StateObjectObj.state4();

//C++
long s1 = StateObjectObj->state1();
long s2 = StateObjectObj->state2();
long s3 = StateObjectObj->state3();
long s4 = StateObjectObj->state4();
```

これだと、4 回の通信が発生します。

これに対して、先の IDL 定義を次のようにしていたとします。

```
interface StateObject {

    struct State {
        long state1;
        long state2;
        long state3;
        long state4;
    };

    State getState();

};
```

この場合、先程のクライアントプログラムは次のように書くことができます。

```
//Java
State state = StateObjectObj.getState();
long s1 = state.state1;
long s2 = state.state2;
long s3 = state.state3;
```

```
long s4 = state.state4;

//C++
State state = StateObjectObj->getState();
long s1 = state.state1;
long s2 = state.state2;
long s3 = state.state3;
long s4 = state.state4;
```

これだと、通信は getStatus()の部分の 1 回しか起こりません。この例は非常に単純な例ですが、CORBA を使用した呼び出しがアプリケーション全体で最小限になるように、全体のオブジェクト構成を設計するように注意してください。

ロギングの組み込み

CORBA を用いたアプリケーションは、ネットワークを使用したプログラムであることを常に意識しておく必要があります。ネットワークを利用したアプリケーションでは、一つの CPU 上で動作する単独のアプリケーションでは起きないようなエラーが起こります。ネットワークを使用して、多数のプログラムが多数のホスト間で協調して動作するプログラムを書く場合、ある一つのホストが障害を起こして動作しなくなったとしてもアプリケーション全体が動作しなくなるというようなことは許されません。

また、複数のマシンで協調動作するプログラムのどこかで何らかのエラーが起こった時に、あらかじめプログラム中にエラー情報を保存しておくしくみなどが組み込まれていないと、何が起こっているのかを判断することが非常に困難になります。

プログラムを作成するときは、エラー処理に細心の気を配ってコーディングしてください。たとえば、次に示すような足し算をする関数では、通常の 1 プロセスで動作するプログラムならばエラー処理を考慮する必要はありません。

```
//Java
public class adder {
    long add(long l1, long l2) {
      return l1+l2;
    }
}

//C++
class adder {
  public:
    long add(long l1, long l2) {
      return l1+l2;
    }
}
```

呼び出し側では

```
//Java
long result = adderobj.add(1L, 2L);

//C++
long result = addreobj->add(1L, 2L);
```

で十分であり、エラーが起こる余地はありません。

しかし、CORBA を使用したネットワーク経由での呼び出しの場合は、リモートホストがダウン中の場合や、リモートホストは動作中であるがネットワークに障害があり通信不能である場合等、いろいろな要因を考慮しなければなりません。このようなケースでは、CORBA は例外を発生させます。プログラムでは、この例外を捕らえ、ロギングのメカニズムを導入して、記録しておくことが障害解析のために必要です。これにより、アプリケーション全体で何が起こったかの把握が可能になります。

プログラム作成の時には、ロギング機構を組み込み、エラー(例外)を無視せずにこれに記録するように注意してください。

アプリケーションで CORBA の例外をロギングする場合には、最低以下の情報を出力することをお勧めします。

- 時刻
- システム例外の種別
- 例外のマイナーコード

マイナーコードを記録しておくことにより、何が起こったかがわかります。システム例外の種別だけでは、具体的に何が起こったかがわからないことが多々あります。

また、注意すべき点として、システム例外やマイナーコードは、OMG の仕様で明確に規定されているわけではないことがあります。つまり、WebOTX Object Broker ではある例外を返すケースでも、別のベンダの ORB では別の種別の例外を返すケースがありえます。また、マイナーコードは完全にベンダ依存であり、個々の ORB ベンダがコードを定義することになっています。したがって、プログラム中で、システム例外 X が起こったときに処理 X を行い、システム例外 Y が起こった時に Y を行うという具合に、例外種別により、ロジックが違うようなコードを書くと、移植性の悪いプログラムになります。

ロギングの仕方は、アプリケーションの性質や使用する OS の持つ機能を考慮に入れて、アプリケーションで最適な方法を決定してください。一般的には、ファイルに記録しておく、OS がロギングの機能を提供している場合にはその機能を使用するなどが考えられます。

7.2.10.CORBA/IIOP による通信の仕組み

CORBA2.0 以上に準拠した ORB は、他社の ORB との通信に IIOP を使用することが義務づけられています。IIOP は CORBA2.0 で規定された通信のプロトコルです。WebOTX Object Broker は、他社製の ORB と通信する場合はもちろん、WebOTX Object Broker を使用したアプリケーション同志の通信についても IIOP を使用します。

プロトコルの詳細を理解する必要はありませんが、ある程度の知識は、性能上の問題が発生した時に役に立ちます。本章では、IIOP の動作の仕組みを説明し、どのようなコーディングをすると性能に影響するか等についての指針を示します。

IIOP はトランスポート層のプロトコルに TCP/IP を使用します。前提知識として、TCP/IP では、通信相手のアドレス情報は、IP アドレス(マシンを識別するアドレス)とそのマシン内でのポート番号のペアであらわされます。IP アドレスは、32 ビットの数値ですが、人間が分かりやすいような名前をホストにつけることもできます。この場合はオペレーティングシステムの hosts ファイルに、ホスト名と IP アドレスのペアを記録しておくことにより、IP アドレスの代わりにホスト名を指定することもできるようになります。また、ホスト名と IP アドレスのペアの管理を DNS と呼ばれるサーバで行い、管理を一元的に行う機能を使用する場合もあります。ただし、TCP/IP のアドレス情報としては、あくまで IP アドレスが必要となりますので、ホスト名を指定した場合には最終的に IP アドレスへ変換する作業が一連の処理のなかのどこかで行われることになります。したがって、DNS の検索が遅い場合には、結果としてオブジェクト呼び出しも遅くなってしまいます。

CORBA ではオブジェクト呼び出しの際にオブジェクトリファレンスと呼ばれるものをオブジェクトの識別子として使用します。

クライアントからオブジェクトを呼び出す時、次のようなコーディングをします。

```
// Java
obj.add(1L, 2L);

// C++
obj->add(1L, 2L);

※以降の図では、C++言語でのコーディング例は省略します。
```

obj はリモートプロセスで実装される CORBA オブジェクトのオブジェクトリファレンスです。

このコードが実行されると CORBA の仕組みでどう動作するかをみていきます。

1. サーバのアドレス情報の取得

   ORB が正しい相手にリクエストを転送するには、obj で表される CORBA オブジェクトが、ネットワーク上のどのマシンで動作する、どのプロセス上で動作しているのかという情報が必要になります。つまり、TCP/IP の用語でいうと、IP アドレスとポート番号を知る必要があります。実は、この二つの情報は obj のデータ構造にすでに含まれています(オブジェクトリファレンスにはこのほかにも様々な情報が含まれています)。したがって、ORB はこの情報を使用して、対象のプロセスにコネクトすることができます。obj のデータ構造に含まれるホストアドレスは IP アドレスでもホスト名でも

よいことになっています。

2. マーシャリング(marshaling)

   次に、引数を対象オブジェクトに渡す必要がありますので、ORB は引数の 1L、2L をネットワークでの転送ができるようにメモリの連続領域に詰め込んでいきます。この処理をマーシャリング(marshaling)と呼びます。

3. メッセージ送信

   マーシャリングが終了すると、ORB は先程確立したコネクションを使用して、IIOP_REQUEST という種別をヘッダにつけて、マーシャリングしたデータを送信します。IIOP_REQUEST というメッセージはクライアントからサーバに対してリクエストを発行する時に使用されるメッセージ種別です。

4. サーバでのアンマーシャリング(unmarshaling)/実装オブジェクトの呼び出し

   IIOP_REQUEST を受信したサーバは、受信したメッセージの内容を個々の引数に分解します。この処理をアンマーシャリング (unmarshaling)と呼びます。その後、プログラマが定義した実装オブジェクトをこの引数を使用して呼び出します。実装オブジェクトからリターンすると ORB には返り値や out 引数が返されます。ORB はこの情報をマーシャリングし、IIOP_REQUEST を受け取った TCP/IP コネクションを使用して、IIOP_REPLY 種別を持ったヘッダを添付して送信します。IIOP_REPLY というメッセージは、サーバからクライアントに対して IIOP_REQUEST に対する返事を返す時に使用されるメッセージ種別です。

5. クライアントでのアンマーシャリング/呼び出し元へのリターン

   IIOP_REPLY メッセージを受け取ったクライアントは、メッセージをアンマーシャルし、out 引数、返り値情報を呼び出し元に返します。

このケースでは、クライアントからサーバオブジェクトのメソッド呼び出し 1 回につき、TCP/IP を使用した通信は 2 回行われており、必要とされる通信回数としては最も効率のよいものとなります。

しかし、IIOP では次のようなシーケンスで呼び出しが起こることもあります。

上記 1-3 までは同じです。

4 において、IIOP_REQUEST を受け取ったサーバが、別のサーバに要求を転送させることができます。WebOTX Object Broker で具体的にはどのような場合に転送が起こるかは後程説明しますが、ここでは、サーバが自分で IIOP_REQUEST を処理することができない場合に、実際に処理が可能なサーバに要求を転送させる場合の動きとして理解してください。

サーバが IIOP_REQUEST を受け取ったがその処理をできない場合、クライアントに対して、「すいませんが、わたしはこのリクエストを処理できません。これこれのアドレスに接続すれば処理をしてもらえますので、こちらにもう一度要求をだしてください。」という内容の返事を返します。IIOP のプロトコルでいうと IIOP_REPLY メッセージに特殊なフラグ(LOCATION_FORWARD)をたて、接続先のアドレス情報を含めて返事を返します。

LOCATION_FORWARD のリプライを受け取ったクライアントは、先程送信したリクエストを LOCATION_FORWARD で示されたアドレスに再送します。

以降の処理は最初の説明と同様になります。

上記の動作は IIOP で規定されているシーケンスであり、CORBA2.0 準拠の ORB では常にサポートされているものです。ただし、1 つの呼び出しに対して、クライアントから 1 つ目のサーバに対する要求、その返事、2 つ目のサーバに対する要求、その返事というように、最低 4 回の通信が行われます。通常、一度 LOCATION_FORWARD を受け取ったクライアントはアドレス情報を記憶しておき、2 回目以降の同一オブジェクトリファレンスを使用した呼び出しでは、最初から実際に要求を処理できるサーバに要求を出すようになります。

```
// 足し算オブジェクトのリファレンスを取得。

obj = someobj.getAdderObj();

// 1 回目の呼び出し。 4 回の通信が起こる。

obj.add(...);

// 同一オブジェクトリファレンスによる 2 回目の呼び出し。
```

```
// 2 回の通信しか起こらない。

obj.add(...);
```

しかし、これを次のように記述すると効率が非常に悪くなります。

```
// 足し算オブジェクトのリファレンス取得。

obj = someobj.getAdderObj();

// 1 回目の呼び出し。4 回の通信が起こる。

obj.add(...);

// 同一の足し算オブジェクトのリファレンス再取得。

obj = someobj.getAdderObj();

// 2 回目の呼び出し。4 回の通信が起こる。

obj.add(...);
```

上記のような記述では、同じオブジェクトに対する 2 回目の要求でも 4 回の通信が起きてしまいます。

上記の例は非常に単純なので実際にこのようなプログラムを書くことはないと思いますが、プログラム内の様々な処理の中で、結果として上記のようなケースになってしまっている可能性がないかどうか気をつけてください。

それでは、LOCATION_FORWARD はどのようなケースで発生するかを説明します。必ずしもこのケースに限られるわけではなく、いろいろな場面で、また個々のベンダの実装方法により違いがありますが、WebOTX Object Broker JavaTM では、サーバの自動起動(Automatic Activation)を実行する時に使用されます。

オブジェクトリファレンスには、ホストアドレスとポート番号が埋め込まれていますが、自動起動可能なサーバのオブジェクトリファレンスのポート番号として、WebOTX Object Broker C++では OAD(Object Activation Daemon)のポート番号が埋め込まれています。

クライアントからの要求は、まず OAD に送信されます。OAD はリクエストメッセージ中の情報から、どのサーバを起動するかを判断しサーバを起動します。起動されたサーバは、これから使用する自分のポート番号を OAD に通知します。通知を受けた OAD は、クライアントに対して LOCATION_FORWARD メッセージを返します。この時、サーバから通知を受けた実際のポート番号も一緒に通知されます。LOCATION_FORWARD を受け取ったクライアントは、今度は実際のサーバオブジェクトに IIOP_REQUEST メッセージを送信するというように動作します。

IIOP には上記で説明した他にもメッセージ種別がありますが、オブジェクト呼び出しで何が起こるかを理解するには、上記の種別を理解すれば十分です。

7.2.11.Java マッピング

IDL 言語のそれぞれの構成要素を Java 言語で使用する規則を定めているのが Java 言語マッピングです。

名前付けのルール

ユーザが定義した型やインタフェースが Java の予約語と同じ名前のとき、名前が衝突しないように、IDL コンパイラが自動的に、定義名の前に"_"を付けて出力します。また Java マッピングにより規定される次の名前についても、名前の衝突が起きないように"_"をつけて出力します。

名前が以下の場合には"_"をつけた名前が使用されるようになります。

- 末尾に"Helper"、"Holder"、"Operations"、"Package"、"POA"、"POATie"が付いた名前
- 先頭に"POA_"が付いた名前
- Java 言語キーワードと同じ名前
- Java 定数と同じ名前
- java.langObject のメソッドと同じ名前

module

IDL の module は同じ名前で Java の package にマッピングされます。そして、module 内で定義されたすべての IDL タイプは、その package の中の Java クラスや interface としてマッピングされます。

基本型

基本型のマッピングは下表のとおりです。各型で例外名の書かれているものは、マーシャリング時にデータのサイズや内容が調べられ、規定どおりでなければそれぞれの例外が返ることを示します。

IDL での型	Java での型	Holder 型	例外
boolean	boolean	BooleanHolder	
char	char	CharHolder	CORBA::DATA_CONVERSION
wchar	char	CharHolder	CORBA::DATA_CONVERSION
octet	byte	ByteHolder	
string	java.lang.String	StringHolder	CORBA::BAD_PARAM CORBA::DATA_CONVERSION
wstring	java.lang.String	StringHolder	CORBA::BAD_PARAM CORBA::DATA_CONVERSION
short	short	ShortHolder	
unsigned short	short	ShortHolder	
long	int	IntHolder	
unsigned long	int	IntHolder	
long long	long	LongHolder	
unsigned long long	long	LongHolder	
float	float	FloatHolder	
double	double	DoubleHolder	
fixed	java.math.BigDecimal	FixedHolder	CORBA::DATA_CONVERSION

Helper クラス

すべてのユーザ定義型に対して Helper クラスが用意されます。名前はユーザ定義型の後ろに"Helper"がつきます。このクラスは次の用途に使われます。

- Any に値を出し入れするため（insert、extract メソッド）
- リポジトリ ID を得るため（id メソッド）
- TypeCode を得るため（type メソッド）
- 通信ストリームに対して読み書きするため（read、write メソッド）

Holder クラス

out および inout パラメータモードをサポートするためのクラスです。このクラスは、org.omg.CORBA パッケージに入っているすべての IDL 基本型や、すべての名前つきユーザ定義型(typedef は除く)に対して生成されます。ユーザ定義型に対する Holder クラス名はユーザ定義型名の後ろに"Holder"が付いたものです。

空のオブジェクトリファレンスや valuetype を表現するために、Java の null を用いることができます。ただし、空の文字列や配列を表現するためには、それぞれ長さ 0 の文字列や配列を用いなければなりません。

boolean

IDL boolean 定数の TRUE と FALSE は、Java の true と false にマッピングされます。

char、wchar

IDL char および IDL wchar は Java の char にマッピングされます。しかし、Java の char は Unicode を表現する 16 ビットですが、IDL の char は 8 ビットで表現します。IDL wchar は 16 ビットで表現されます。char が範囲外の値を表現していたなら、CORBA::DATA_CONVERSION 例外が返ります。

octet

IDL octet は 8 ビットで、Java の byte にマッピングされます。

string、wstring

IDL string および IDL wstring は固定長でも可変長でも java.lang.String にマッピングされます。IDL string は 8 ビットキャラクタの文字列を表します。IDL wstring は 16 ビットキャラクタの文字列を表します。型の違反があれば、CORBA::DATA_CONVERSION 例外が返ります。固定長文字列の長さチェックで違反があれば、CORBA::BAD_PARAM 例外が返ります。

fixed

IDL fixed は java.math.BigDecimal にマッピングされます。

定数

interface 内で定義された定数は、Java の interface 内のクラス変数にマッピングされます。

interface の外で定義された定数は、定数の値を持った value という名前のクラス変数を含む、定数と同じ名前の public interface にマッピングされます。

```
// IDL
module Example {
    interface Face {
        const long aLongerOne = -321;
    };
    const long aLongOne = -123;
};
```

```
// Generated Java
package Example;
public interface FaceOperations {
}

public interface Face extends FaceOperations, org.omg.CORBA.Object,
                              org.omg.CORBA.portable.IDLEntity {
    int aLongerOne = (int) (-321L);
}

public interface aLongOne {
    int value = (int) (-123L);
}
```

enum

IDL enum は enum 型と同じ名前の Java の class にマッピングされます。

```
// IDL
enum EnumType {a, b, c};
```

```
// Generated Java
public class EnumType implements org.omg.CORBA.portable.IDLEntity {
    public static final int _a = 0;
    public static final EnumType a = new EnumType(_a);
    public static final int _b = 1;
    public static final EnumType b = new EnumType(_b);
    public static final int _c = 2;
    public static final EnumType c = new EnumType(_c);
    public int value() {...}
    public static EnumType from_int(int value) {...}

    // コンストラクタ
    protected EnumType(int) {...}
};
```

struct

IDL struct は、それぞれのメンバ、デフォルトコンストラクタ、各メンバを引数にとるコンストラクタをもつ、同じ名前の Java の final class にマッピングされます。

union

IDL union は、次のメソッドをもつ Java の final class にマッピングされます。

- デフォルトコンストラクタ
- 弁別子のアクセスメソッド
- それぞれのメンバのアクセスメソッド
- それぞれのメンバの変更メソッド
- 複数の case ラベルを持つメンバのための変更メソッド
- デフォルト case ラベルを持つメンバのための変更メソッド
- （必要ならば）デフォルト変更メソッド

型名やメンバ名が弁別子名と衝突した場合は、弁別子のメソッド名は "_" を付けたものになります。

メンバアクセスメソッドと変更メソッドはメンバ名と同じ名前でオーバロードされます。値をセットする前にアクセスメソッドが呼ばれたときは、CORBA::BAD_OPERATION 例外が返ります。

デフォルト case ラベルのメンバの変更メソッドは、デフォルト値を弁別子にセットします。また、弁別子の値を明示的に指定できる変更メソッドも生成されます。

弁別子にセットできるすべてのラベルが case ラベルとして定義されているとき、さらにデフォルト case ラベルを定義することはできません。このときは IDL コンパイラがエラーを通知します。

デフォルト変更メソッド_default()は、デフォルト case ラベルが明示的に定義されなかったときや、弁別子にセットできるすべてのラベルが case ラベルとして定義されなかったときに生成されます。

sequence

IDL sequence は、同じ名前の Java の配列にマッピングされます。固定長シーケンスの場合、ストリームへの書込み時に長さがチェックされます。長さに違反していたら CORBA::MARSHAL 例外が返ります。

array

IDL array は IDL 固定長 sequence と同じ方法でマッピングされます。長さ違反チェックはマーシャリング時に行われ、違反していたら CORBA::MARSHAL 例外が返ります。

interface

interface クラス

IDL interface は同じ名前の Java の public interface にマッピングされます。Java の interface は org.omg.CORBA.Object インタフェースを基底インタフェースとします。

IDL operation は、対応する Java interface のメソッドにマッピングされます。

interface の Helper クラスには static の narrow メソッドが追加されます。これは、基底インタフェースの org.omg.CORBA.Object から IDL interface に対応する Java interface に変換するためのメソッドです。narrow の実行に失敗したら、CORBA::BAD_PARAM 例外が返ります。

nil オブジェクトリファレンスの値は Java の null を使います。

IDL attribute は Java の取り出しと書き替えの 2 つのメソッドにマッピングされます。これらは同じ名前でオーバロードされます。IDL readonly attribute は取り出しメソッドのみにマッピングされます。

IDL で interface の継承があるときは、直接 Java の interface の継承を使います。

abstract interface

IDL abstract interface は、オブジェクトリファレンスと valuetype のどちらも扱うことが可能です。通常の IDL interface と同様に、同じ名前の Java の public interface にマッピングされますが、org.omg.CORBA.Object インタフェースの明示的な継承はしていません。

Parameter Passing Modes

Java は、渡された引数の値を変更することができません。このため、呼び出し元から渡された値を変更するためのしくみとして、Holder クラスが存在します。IDL の in パラメタに対しては Java マッピングで規定されている Java の型を直接渡します。これに対して out 引数については、メソッドからの変更結果が Holder オブジェクトに挿入されて返ってきます。inout 引数については呼び出し元から Holder に引数の値を挿入してメソッド呼び出しを行い、メソッドからの返却値が Holder に設定された状態で返ってきます。

exception

IDL exception のマッピングは struct とほとんど同じです。Java のクラスにマッピングされ、そのクラスは、メンバに対応したインスタンス変数と、デフォルトコンストラクタ、メンバの初期値を指定できるコンストラクタを持ちます。

CORBA システム例外(SystemException)は、java.lang.RuntimeException を継承します。

ユーザ例外(UserException)は、java.lang.Exception を継承します。SystemException は Java のコード中に throws 句や catch 節が必要ありませんが、UserException に関しては明示的に throws 句もしくは catch 節を記述する必要があります。

SystemException

CORBA システム例外は、org.omg.CORBA.SystemException から継承した Java の public final クラスにマッピングされます。それぞれの IDL 標準例外は下表のとおりです。

IDL の例外	Java クラス名
CORBA::UNKNOWN	org.omg.CORBA.UNKNOWN
CORBA::BAD_PARAM	org.omg.CORBA.BAD_PARAM
CORBA::NO_MEMORY	org.omg.CORBA.NO_MEMORY
CORBA::IMP_LIMIT	org.omg.CORBA.IMP_LIMIT
CORBA::COMM_FAILURE	org.omg.CORBA.COMM_FAILURE
CORBA::INV_OBJREF	org.omg.CORBA.INV_OBJREF
CORBA::NO_PERMISSION	org.omg.CORBA.NO_PERMISSION
CORBA::INTERNAL	org.omg.CORBA.INTERNAL
CORBA::MARSHAL	org.omg.CORBA.MARSHAL
CORBA::INITIALIZE	org.omg.CORBA.INITIALIZE
CORBA::NO_IMPLEMENT	org.omg.CORBA.NO_IMPLEMENT
CORBA::BAD_TYPECODE	org.omg.CORBA.BAD_TYPECODE
CORBA::BAD_OPERATION	org.omg.CORBA.BAD_OPERATION
CORBA::NO_RESOURCES	org.omg.CORBA.NO_RESOURCES
CORBA::NO_RESPONSE	org.omg.CORBA.NO_RESPONSE
CORBA::PERSIST_STORE	org.omg.CORBA.PERSIST_STORE
CORBA::BAD_INV_ORDER	org.omg.CORBA.BAD_INV_ORDER
CORBA::TRANSIENT	org.omg.CORBA.TRANSIENT
CORBA::FREE_MEM	org.omg.CORBA.FREE_MEM
CORBA::INV_IDENT	org.omg.CORBA.INV_IDENT
CORBA::INV_FLAG	org.omg.CORBA.INV_FLAG
CORBA::INTF_REPOS	org.omg.CORBA.INTF_REPOS
CORBA::BAD_CONTEXT	org.omg.CORBA.BAD_CONTEXT
CORBA::OBJ_ADAPTER	org.omg.CORBA.OBJ_ADAPTER
CORBA::DATA_CONVERSION	org.omg.CORBA.DATA_CONVERSION
CORBA::OBJECT_NOT_EXIST	org.omg.CORBA.OBJECT_NOT_EXIST
CORBA::TRANSACTION_REQUIRED	org.omg.CORBA.TRANSACTION_REQUIRED
CORBA::TRANSACTION_ROLLEDBACK	org.omg.CORBA.TRANSACTION_ROLLEDBACK
CORBA::INVALID_TRANSACTION	org.omg.CORBA.INVALID_TRANSACTION
CORBA::INV_POLICY	org.omg.CORBA.INV_POLICY
CORBA::CODESET_INCOMPATIBLE	org.omg.CORBA.CODESET_INCOMPATIBLE

UserException

ユーザ例外は、org.omg.CORBA.UserException から継承した Java の final class にマッピングされます。クラスの内容は IDL struct に対して生成されるものと同様です。Helper クラスと Holder クラスも生成されます。生成された Java クラスは完全なコンストラクタを持ち、String 型のパラメータはベースの UserException クラスのコンストラクタを呼ぶ時の id に連結されます。

例外がネストした IDL スコープ内で定義された場合は、Java クラス名はそのスコープ内で定義されます。そうでなければ、Java クラス名は例外が定義されている IDL module に対応した Java package 内に定義されます。

Any

IDL any 型は Java クラス org.omg.CORBA.Any にマッピングされます。このクラスはあらかじめ定義されているすべての型のインスタンスを出し入れするメソッドを持っています。もし取り出しオペレーションで型がマッチしなければ、CORBA::BAD_OPERATION 例外が返ります。ユーザ定義型のための出し入れメソッドは、それぞれの型の Helper クラスで実装されます。

挿入オペレーションは値をセットして、TypeCode を新たにセットします。

引数付きの type(org.omg.CORBA.TypeCode)メソッドによって TypeCode をセットすると、現在挿入されている値はクリアされ、Any の内部の型として指定された TypeCode が設定されます。このオペレーションは、IDL out parameter をセットするときに使います。値をセットする前に値を取り出そうとしたときは、CORBA::BAD_OPERATION 例外が返ります。

Any のインスタンスは ORB.create_any()で作成します。

Nested Type

IDL ではインタフェースの中に型を定義できますが Java では許されていません。Java のクラスにマッピングされる型がインタフェース内で定義されたときは、少し特殊なマッピングになります。まずインタフェース名に"Package"という名前をつけたパッケージを定義し、そのパッケージ内で、インタフェース内で定義されたクラスが定義されます。

typedef

IDL 基本型の typedef は、オリジナルの型にマッピングされます。つまり、基本型に対する typedef が使用されているところでは、すべて typedef のもとになったオリジナルの型を使用します。

配列とシーケンス以外の IDL 型の typedef は、オリジナルの型（基本型もしくは typedef されていないユーザ定義型）にマッピングされます。

Helper クラスがすべての typedef 型に対して生成されます。Holder クラスは、配列とシーケンスの typedef に対してのみ生成されます。

valuetype

concrete valuetype

IDL valuetype は、同じ名前の Java の abstract クラスと、<valuetype 名>ValueFactory という名前の Java インタフェースにマッピングされます。また、Helper クラスと Holder クラスが生成されます。

valuetype がマッピングされている abstract クラスは、定義されたステートに対応するインスタンス変数を持ちます。IDL で public と指定されたフィールドは public のインスタンス変数に、private と指定されたフィールドは protected のインスタンス変数にマッピングされます。

オペレーションと attribute は、この valuetype が継承、サポートしている valuetype やインタフェース内で定義されたのものも含めて、abstract メソッドとして展開されます。valuetype の実装者は、この Java クラスを継承した実装クラスを定義して、これらの abstract メソッドの実装コードを記述します。

この Java クラスは、IDL の valuetype 定義に custom 指定があるかどうかによって、org.omg.CORBA.portable.CustomValue か org.omg.CORBA.portable.StreamableValue のどちらかを継承し、ValueBase インタフェースの実装を提供します。custom 指定がない valuetype の Java クラスは、org.omg.CORBA.portable.Streamable インタフェースの実装を提供します。custom 指定がある valuetype の場合、org.omg.CORBA.portable.CustomValue の実装は、valuetype の実装者が行います。他の valuetype を継承する場合は、継承する valuetype にマッピングされた Java クラスが継承されます。

ValueFactory インタフェースは、org.omg.CORBA.portable.ValueFactory を継承して、IDL で factory 宣言されたそれぞれのファクトリに対応するメソッドを含んでいます。メソッド名はファクトリ名と同じで、ファクトリの引数は IDL operation のパラメータと同じ方法で展開されます。valuetype の実装者は、生成された ValueFactory インタフェースのメソッドを実装したファクトリクラスを提供する必要があります。ファクトリが IDL で宣言されていないときは、ファクトリインタフェースは生成されません。この場合、valuetype の実装者は、read_value()のメソッドを実装するために org.omg.CORBA.portable.ValueFactory の実装クラスを作成します。

Object Broker Java では、ValueFactory の実装クラスの雛形を、<valuetype 名>DefaultFactory という名前で生成します。すでに、この名前のファイルが存在する場合は上書きされません。この雛形ファイルは、ValueFactory インタフェースを実装する Java クラスで、valuetype の実装クラス名を<valuetype 名>Impl として記述しています。valuetype の実装者は、この雛形ファイルの valuetype の実装クラス名部分を実装に合わせて書き換えることによって、ValueFactory の実装クラスを作成することができます。

abstract valuetype

IDL abstract valuetype は同じ名前の Java の public interface にマッピングされます。

abstract valuetype 内に記述した IDL operation および IDL attribute に対応するメソッドは、abstract メソッドとして展開されます。

abstract valuetype 自体を直接実装することはできません。abstract valuetype を実装するには、これを継承する concrete valuetype を定義し、実装します。

Helper、Holder クラスは concrete valuetype と同様に展開されます。

boxed valuetype

boxed valuetype には、Java プリミティブ型にマッピングされるものと、同じ名前の Java クラスにマッピングされるものがあります。Holder クラスは他の valuetype と同様に生成されます。Helper クラスも生成されますが、boxed valuetype 固有の形式で展開されます。

boxed valuetype の型がプリミティブ型(float, long, char, wchar, boolean, octet など)の場合は、boxed valuetype と同じ名前で生成される Java クラスにマッピングされます。このクラスは、boxed valuetype の IDL 型に対応する型の、value という名前の public データを1つだけ持っています。

boxed valuetype の型が Java クラス(string, wstring, enum, struct, sequence, array, any, interface など)の場合は、boxed valuetype はそのクラスにマッピングされます。

複雑型のための読み書き API

Streamable Interface API は、複雑型の値の読み出しおよび書き込みをサポートしています。これは Helper クラスの static メソッドとして実装されています。また、Holder クラスの中でも out および inout パラメータの complex データタイプを読み書きするために使われます。

Streaming API

Streaming API は、マッピングされた IDL 型をストリームに対して読み書きするときに使用する API です。これらは、ORB が、パラメータをマーシャリングしたり、Any に complex データタイプを出し入れしたりするときに使います。

Streaming API は org.omg.CORBA.portable および org.omg.CORBA_2_3.portable パッケージで定義されています。

OutputStream は ORB オブジェクトによって生成され、InputStream は OutputStream から生成されます。

7.2.12.C++マッピング

IDL 言語のそれぞれの構成要素を C++言語で使用する規則を定めているのが C++言語マッピングです。本書では C++マッピングについて説明します。

スコープを持つ名前(ScopedNames)

IDL のスコープを持つ名前は C++のスコープに対応づけられます。

・OMG IDL の module は C++の namespace になります。

・OMG IDL の interface は C++の class になります。

・interface 中で定義された IDL の構成要素は C++のスコープで参照されます。たとえばインタフェース printer 中で定義された mode という型は C++から printer::mode という型で参照されます。

モジュール(module)

次のような IDL は

```
//IDL
module M {
        interface I {
                struct S {
                ...
                };
        ...
        };
};
```

C++では次のような宣言になります。

```
//C++
namespace M
{
        class I ...
        {
        ...
                struct S
                {
                ...
                };
        };
};
```

インタフェース

IDL の interface は interface 中で定義された型、例外、メソッド等が定義された C++のクラスに対応づけられます。

```
// IDL
interface I {
        void op1();
};

// C++
class I ...
{
        public void op1(...);
};
```

CORBA に準拠するアプリケーションプログラムでは、次のことを守らなければなりません。

・interface class のインスタンスを作成したり保持したりしないこと。

・interface class へのポインタ(I*)やリファレンス(I&)を使用しないこと。

次に例を示します。

```
// IDL
interface A
{
        struct S { short field; };
};

// C++マッピングに準拠した方法
A_S s; // struct の変数を宣言
s.field = 3; // フィールドのアクセス

// C++マッピングに違反した方法
// interface class を作成してはいけない
A a;
// interface class へのポインタを使用してはいけない
A *p;
// interface class へのリファレンスを使用してはいけない
void f(A &r);
```

オブジェクトリファレンス型

IDL のインタフェースをあらわすにはオブジェクトリファレンス型を使用します。オブジェクトリファレンス型はインタフェース名を A とすると A_var および A_ptr の 2 種類があります。インタフェースのメソッドを呼び出すときにはオブジェクトリファレンスに矢印オペレータ(->)を使用して呼び出します。たとえば、インタフェースのメソッド op1 を呼び出すときにはオブジェクトリファレンスを使用して、obj->op1(..)のように呼び出します。オブジェクトリファレンスとして A_var 型を用いた場合でも、A_ptr を用いた場合でも同様に矢印オペレータを使用します。

A_var と A_ptr の違いは、A_var ではオブジェクトリファレンスのメモリ管理が自動的に行われることです。

```
{
        // C++のスコープ
        A_var objvar = ...;
        A_ptr objptr = ...;
        objvar->op(...);
        objptr->op2(...);
}

// スコープの外に出ると objvar の指すオブジェクトは自動的にリリースされるが、
// objptr は有効なまま。 したがって objptr は明示的にリリースしない限り、
// リークしている。
```

A_var と A_ptr 型はお互いに代入可能です。また、A_var が使用できる場所では、A_ptr も同様に使用できますし、逆も同じです。しかし、A_var で管理されるオブジェクトリファレンスは自動的にリリースされることに十

分注意してプログラミングする必要があります。次の例は実行時エラーになります。

```
...
A_ptr objptr = ...;
{
        A_var objvar = objptr;
        objvar->op1(...);
}

// objptr の指すオブジェクトリファレンスは A_var objvar が
// スコープを出たときに自動的にリリースされている。
// したがって、ここで呼び出しを行うとリリースされてしまったものを
// 使用して呼び出すことになる。

objptr->op1(..); // 実行時エラーになる可能性あり
```

オブジェクトリファレンスの操作

Widening と Narrowing

インタフェース B がインタフェース A を継承している場合、

```
//IDL
interface A {
        ...
};

interface B : A {
        ...
};
```

次のことが可能です。

```
// C++

A_ptr ap;
B_ptr bp;
B_var bv;
CORBA::Object_ptr objp;
ap = bp;
objp = bp;
ap = bv;
objp = bv;
```

B_var から A_var や CORBA::Object_var への暗黙の変換はサポートされていません。この場合には _duplicate を呼び出して相当することを行う必要があります。

ただし、同じ型どうしの var の代入は可能です。

```
// C++
B_ptr bp = ...
A_ptr ap = bp; // 暗黙の widening
CORBA::Object_ptr objp = bp; // 暗黙の widening
objp = ap; // 暗黙の widening
B_var bv = bp; // bv が bp の所有権を持つようになる
ap = bv; // 暗黙の widening。bv は bp の所有権を持ったまま
obp = bv; // 暗黙の widening。bv は bp の所有権を持ったまま
A_var av = bv; // できない。コンパイルエラー
A_var av = B::_duplicate(bv); // _var の widening のやり方
A_var a2v;
a2v = av; // OK。自動的に_duplicate が行われる
```

オブジェクトリファレンスは_narrow というスタティックメンバ関数を持っており、これを使用することにより、Widening されたオブジェクトリファレンスからサブインタフェースを指すオブジェクトリファレンスを得ることができます。

```
A_ptr aptr = ...;

// 暗黙的に Widening がおこる
CORBA::Object_ptr objptr = aptr;
A_ptr a2ptr;

// A::_narrow を呼び出すことにより新たに
// A のオブジェクトを指すオブジェクトリファレンスが得られる
a2ptr = A::_narrow(objptr);
```

ここで注意しなければならないのは、_narrow は新たにオブジェクトリファレンスを作成して返すことです(これに対して、Widening は新たなオブジェクトリファレンスを作成しません)。したがって、返されたオブジェクトリファレンスは元のオブジェクトリファレンスとは別に解放をしなければなりません。ただし、新たに作られるのはオブジェクトリファレンスであって、オブジェクトそのものではありません。どちらのオブジェクトリファレンスを使用してオペレーションの呼び出しを行っても、実際に実行が行われるオブジェクトは同じオブジェクトです。

_narrow を行ったときに、実際にはその型に narrow できないときには_narrow は nil オブジェクトリファレンスを返します。

```
// IDL
interface Base {
        ...
};

interface Dervied : Base {
        ...
};

interface Another {
        ...
};

//C++
Derived_ptr d = ...;
Base_ptr b = d; // widening
Derived_ptr d2 = Derived::_narrow(b);

// Derived のオブジェクトリファレンスを返す
Another_ptr anotherptr = ...;

// anotherptr は Derived のベースではないので
// nil オブジェクトリファレンスが返る
Derived_ptr d3 = Derived::_narrow(anotherptr);
```

オブジェクトリファレンスに対する操作

すべてのオブジェクトリファレンスに対して_duplicate、release、is_nil という操作が定義されています。これらはオブジェクトリファレンスに対する操作であってオブジェクト自体に対する操作ではないことに注意してください。

3 つの操作のうち release、is_nil はオブジェクトリファレンスの型に依存しないので、CORBA モジュール中で定義されています。

```
// C++

CORBA::release(CORBA::Object_ptr obj);

CORBA::Boolean CORBA::is_nil(CORBA::Object_ptr obj);
```

CORBA::release はオブジェクトリファレンスを解放します。オブジェクト自体を解放するわけではありません。CORBA::is_nil はオブジェクトリファレンスが nil かどうかをチェックし、nil であれば 1 を返します。

_duplicate はそれぞれのオブジェクトの型ごとにスタティックメンバ関数として用意されています。_duplicate は引数で与えられたオブジェクトリファレンスから、同じオブジェクトをあらわす新規オブジェクトリファレンスを作成します。

```
// IDL

interface A { };

// C++

class A
{
        public:
        static A_ptr _duplicate(A_ptr obj);
};
```

引数で与えられたオブジェクトリファレンスが nil だった場合_duplicate は nil を返します。

nil オブジェクトリファレンス

interface のマッピングによってスタティックメンバ関数_nil()が定義されます。_nil()はそのインタフェースの nil オブジェクトリファレンスを返します。また、CORBA::is_nil はオブジェクトが nil かどうかをチェックします。

次の式は常に 1 を返します。

```
CORBA::Boolean b = CORBA::is_nil(A::_nil());
```

オブジェクトリファレンスの比較に==オペレータを使用することはできません。オブジェクトリファレンスが nil かどうかを調べるには上記のように CORBA::is_nil を使います。

定数(constant)

IDL の定数は C++の定数になります。IDL でグローバルなスコープで定義された定数は、C++のファイルスコープで定義される定数になり、インタフェース中で定義された定数は、C++のクラス中で定義されるスタティックな定数にマッピングされます。

次に例を示します。

```
// IDL
const string name = "testing";

interface A
{
        const float pi = 3.14159;
};

// C++
static const char *const name = "testing";

class A
{
        public:
        static const Float pi;
};
```

基本データ型

IDL の基本データ型は C++の型に次のように対応づけられます。

OMG IDL　　C++

short	CORBA::Short
long	CORBA::Long
long long	CORBA::LongLong
unsigned short	CORBA::UShort
unsigned long	CORBA::ULong
unsigned long long	CORBA::ULongLong
float	CORBA::Float
double	CORBA::Double
long double	CORBA::LongDouble
char	CORBA::Char
wchar	CORBA::WChar
boolean	CORBA::Boolean
octet	CORBA::Octet

CORBA::Boolean は 1(TRUE)か 0(FALSE)の値のみが正当な値として定義されており、それ以外の値が設定された場合の挙動は定義されていません。またプラットフォームによりデータ型のサイズが異なるのを吸収するために CORBA モジュールにそれぞれの基本型が typedef されています。

列挙型(enum)

IDL の enum は C++の enum にマッピングされます。

次に例を示します。

```
// IDL
enum Color { red, green, blue };

// C++
enum Color { red, green, blue };
```

文字列(string, wstring)

IDL の string は C++の char*にマッピングされます。さらに CORBA::String_var 型が用意されています。この型は char*を保持し、CORBA::String_var がデストラクトされるときに文字列を自動的に解放するようになっています。

同様にワイド文字列を表す wstring は CORBA::WChar*にマッピングされます。また、CORBA::WString_var 型が用意されています。

CORBA::String_var 型に char*が代入されると、char*の所有権は CORBA::String_var が持つことになり、メモリの解放は CORBA::String_var が自動的に行うようになります。

ただし、const char*を CORBA::String_var に代入する場合には、const char*の指す内容が新たにコピーされ、CORBA::String_var に代入されます。

次に例を示します。

```
char * s1ptr = CORBA::string_dup("test string");
const char *cs2ptr = "const test string";

{
        // 所有権の管理を s1var が行う
        CORBA::String_var s1var = sptr;

        // コピーが作成されそれの所有権の管理を s2var が行う
        CORBA::String_var s2var = cs2ptr;
}

// スコープを出ると sptr の指す内容は free される
```

文字列のアロケーションを行うにはアプリケーションでは次の関数を使う必要があります。通常の calloc や

free を使うと予期しない場所で実行時エラーになります。

ワイド文字列も、型が異なるだけで、ふるまいは同じです。

```
CORBA::string_alloc(CORBA::ULong len);
CORBA::string_dup(const char*);
CORBA::string_free(char*);

CORBA::wstring_alloc(CORBA::ULong len);
CORBA::wstring_dup(const CORBA::WChar*);
CORBA::wstring_free(CORBA::WChar*);
```

CORBA::string_alloc は文字列の終了をあらわすヌル文字('¥0')を含む文字列を保持できるように len+1 文字分のメモリをアロケートし、先頭アドレスを返します。アロケートできない場合にはヌル・ポインタを返します。

CORBA::string_dup は引数で与えられた文字列の長さ+1 の文字数分の領域をアロケートし、そこに文字列をコピーします。アロケートに成功した場合には先頭のアドレスを返し、失敗した場合にはヌル・ポインタを返します。

CORBA::string_free は CORBA::string_alloc、CORBA::string_dup でアロケートされた文字列を解放します。

CORBA::string_free にヌル・ポインタを渡してもかまいません。この場合は何も起こりません。

CORBA::wsting_XXX はワイド文字列に使います。使い方は同じです。

構造型(StructuredTypes)

IDL の struct, union, sequence(ただし array を除く)は、デフォルトコンストラクタ、コピーコンストラクタ、代入オペレータ、デストラクタを持つクラスまたは構造体にマッピングされます。デフォルトコンストラクタはメンバを初期化するときに、オブジェクトリファレンスに関しては nil オブジェクトリファレンスで、文字列に関してはヌル・ポインタで初期化します。コピーコンストラクタは deep-copy を行います。これを行うのに、オブジェクトリファレンスに関しては_duplicate を呼び、文字列に関しては新たにメモリのアロケートを行います。代入オペレータは以前の値を解放し、新しい値を deep-copy します。デストラクタはオブジェクトリファレンス、文字列の解放を行います。

IDL の構造型に対してはそれが固定長(fixed length)であるか、可変長(variable length)であるかによって扱いが変わります。構造型が固定長であるとは次のときをいいます。

・any 型

・string 型

・sequence 型

・オブジェクトリファレンスあるいは転送可能な擬似オブジェクトへのリファレンス

・可変長のメンバを含む struct または union

・可変長の要素を持つ array

・可変長の型の typedef

可変長型の out 引数や返り値型に対しては、それぞれの型のポインタが返されますが、このポインタの管理を容易にするための型が定義されています。構造型 T に関しては T_var という型が同時に定義されます。この型は T 型あるいは T_ptr 型と同様にふるまい、メモリ管理を自動的に行うのに使用されます。T_var のメンバアクセスを行うときには矢印オペレータ(->)を使用します。

```
// IDL
struct S { string name; float age; };

void f(out S p);

// C++
S a;
S_var b;
f(b);
a = b; // deep-copy
```

```
cout << "names" << a.name << ", " << b->name << endl;
```

T_Var 型

一般的な T_var 型の構造を次に示します。

```
// C++
class T_var

{
        public:
        T_var();
        T_var(T *);
        T_var(const T_var &);
        ~T_var();
        T_var &operator=(T *);
        T_var &operator=(const T_var &);
        T *operator-> const ();
        /* in parameter type */ in() const;
        /* inout parameter type */ inout();
        /* out parameter type */ out();
        /* return type */ _retn();
        // other conversion operators to support
        // parameter passing
};
```

デフォルトコンストラクタは、T*としてヌル・ポインタを持つ T_var を作ります。アプリケーションでは T_var に正しいポインタを設定する前に->オペレータを使用したり、T*への変換オペレータを使用してはいけません。

T*をとるコンストラクタは、デストラクトされるときにこの T*の解放を自動的に行う T_var を作成します。このとき、引数にヌル・ポインタを指定してはいけません。

T_var のコピーコンストラクタは引数で渡されたオブジェクトを deep-copy します。ここで作成されたコピーは T_var がデストラクトされるときや新しい値が代入されたときに破棄されます。

デストラクタは T_var で指されているポインタに対して delete を行います。ただし、文字列や配列の場合には、CORBA::string_free、T_free を行います。

T*をとる代入オペレータは古い値を解放後新しい値の所有権を管理するようになります。

通常の代入オペレータは deep-copy を行います。

矢印オペレータ(->)は T_var に保持されている T*を返しますが、所有権は保持したままです。T_var を T*あるいは T_var で初期化した後でないとこのオペレータを使用できません。

T_var 型を const T*に対して使用することはできません。const T*を T_var に変換して使用したい場合には次のようにして行うことができます。

```
// C++
const T *t = ...;
T_var tv = new T(*t);
```

暗黙の型変換はしばしば問題を起こすので、明示的変換関数が用意されています(in, inout, out 関数)。これらの関数はそれぞれ対応する引数渡しのモードの引数に対して用いることができます。

_retn 関数は T_var 型内部で管理しているポインタを解放せずに返します。オブジェクトの実装コード内で T_var で管理していたデータをリターン値として返すときに使用すると便利です。_retn 関数を呼び出すと、内部のポインタはヌルに変更され、新たに値を設定しない限り、矢印オペレータなどは使えなくなります。

struct 型

IDL の struct は C++の struct にマッピングされます。C++の struct はコンストラクタ、デストラクタ、代入オペレータを持ちません。IDL で定義されたメンバに対応する C++のメンバが struct のメンバとして定義されます。ただし、string オブジェクトリファレンスのメンバに対しては、メモリ管理が自動的に行われる特殊なクラスが使用されていますが、プログラマがこれを意識する必要はありません。

```
// IDL
struct Fixed{ float x, y, z; };

// C++
```

```
Fixed x1 = {1.2, 2.4, 3.6};
Fixed_var x2 = new Fixed;
x2->y = x1.z;
```

上の例は固定長の struct の使用方法を示しています。x2 がスコープの外に出ると、x2 が所有権を管理しているオブジェクト(new Fixed でアロケートされたオブジェクト)は自動的に delete を使用して解放されます。

次の例は可変長 struct の使用例です。

```
// IDL
interface A;

struct Variable {
       string name;
       A obj;
};

// C++
Variable *v1 = ....;
Variable_var vv1 = v1;
char *s = ...;
A_ptr o = ...;
vv1.name = s; // 古い値は解放され s の領域がコピーされる
vv1.obj = o; // 古い値は解放され o が_duplicate されて保持される
```

union 型

IDL の union は C++のクラスにマッピングされます。出力された C++クラスのデフォルトコンストラクタは初期化を行いません。アプリケーションでは使用する前に必ず初期化を行う必要があります。

union の弁別子(discriminant)アクセスのメンバ関数は_d という名前で定義されています。弁別子を設定するための_d は同じメンバに対して別の弁別子が割り当てられている場合にのみ使用することができます(以下の例の 4, 5 の場合など)。union の値をアクセス関数で設定することにより、弁別子は自動的に設定されます。設定されている以外の型で取り出しを行ったときの動作は規定されていません。このようなことを行ってはいけません。

default の指定を持たない union で、可能なすべての値が case で指定されていないものに関しては_default というメンバ関数が定義されます。default()は、弁別子を適当なデフォルト値に設定します。

アクセス関数で新しい値を挿入すると古い値は自動的に解放され、新しい値のコピーが作成されて挿入されます。

次に例を示します。

```
// IDL
typedef octet Bytes[64];
struct S { long len; };

interface A;

union U switch (long) {
       case 1: long x;
       case 2: Bytes y;
       case 3: string z;
       case 4:
       case 5: S w;
       default: A obj;
};

// C++
typedef CORBA::Octet Bytes[64];
typedef CORBA::Octet Bytes_slice;

class Bytes_forany { ... };

struct S { CORBA::Long len; };
```

```
typedef ... A_ptr;

class U
{
        public:
        U();
        U(const U&);
        ~U();

        // 古い値が解放され新しい値のコピーが設定される
        U &operator=(const U&);
        void _d(CORBA::Long); // 弁別子を設定する
        CORBA::Long _d() const; // 弁別子を参照する
        void x(CORBA::Long);
        CORBA::Long x() const;
        void y(Bytes);
        Bytes_slice *y() const;
        // 古いメモリ領域を解放。コピーは行わない。
        void z(char*);
        // 古いメモリ領域を解放。コピーを行う。
        void z(const char*);
        // 古いメモリ領域を解放。コピーを行う。
        void z(const CORBA::String_var &);
        const char *z() const;
        void w(const S &); // deep-copy<を行う。
        const S &w() const; // リードオンリーアクセス
        S &w(); // リードライトアクセス
        // 古いオブジェクトをリリース。新しいオブジェクトを      _duplicate
        void obj(A_ptr);
        A_ptr obj() const; // _duplicate は行われない
};
```

sequence 型

sequence は現在の長さと最大長をもつ、配列のようにふるまう C++のクラスとしてマッピングされます。最大長指定ありの sequence(bounded sequence)の最大長は固定で決められ、プログラマが制御することはできません。最大長指定なしの sequence(unbounded sequence)は、コンストラクタで最大長を指定することができます。これによって最初にアロケートされるバッファの大きさを制御することができます。現在の長さは最大長指定あり、なしに関わらずプログラマが制御することができます。

最大長指定なしの sequence では、長さを最大長よりも大きなものを指定すると、バッファを再割り当てします。これは新しいバッファを割り当て中身をコピーし、古いバッファを解放することによって行われます。

最大長指定ありの sequence に対して最大長よりも大きな長さを設定してはいけません。この時の動作は規定されていません。

IDL で定義された最大長指定なしの sequence、最大長指定ありの sequence に対して、それぞれ次のようなクラスが生成されます。

```
// IDL
typedef sequence<long> LongSeq;
typedef sequence<LongSeq, 3> LongSeqSeq;

// C++
class LongSeq // unbounded sequence
{
        public:
        LongSeq(); // default constructor
        LongSeq(CORBA::ULong max); // maximum constructor
        LongSeq( // T *data constructor
                  CORBA::ULong max,
                  CORBA::ULong length,
                  CORBA::Long *value,
                  CORBA::Boolean release = 0
```

```
        );
        LongSeq(const LongSeq&);
        ~LongSeq();
        ...

};

class LongSeqSeq // bounded sequence
{
        public:
        LongSeqSeq(); // default constructor
        LongSeqSeq( // T *data constructor
                CORBA::ULong length,
                LongSeq *value,
                CORBA::Boolean release = 0
        );
        LongSeqSeq(const LongSeqSeq&);
        ~LongSeqSeq();
        ...

};
```

最大長指定あり/なしに関わらずデフォルトコンストラクタは sequence の長さを 0 に設定します。最大長指定ありの sequence に対して最大長を変更することはできません。最大長指定なしの sequence のデフォルトコンストラクタは、最大長を 0 に設定します。最大長指定ありの sequence のデフォルトコンストラクタは sequence の中身のバッファ領域をアロケートします。したがって、release フラグは 1 に設定されます。最大長指定なしの sequence には最大長を指定できるコンストラクタがあります。このコンストラクタは指定された長さを格納するだけのバッファをアロケートします。このコンストラクタも release フラグを 1 に設定します。

T* data コンストラクタは長さと sequence に挿入する内容を指定することができます。最大長指定なしの sequence に関しては同時に最大長を指定することもできます。このコンストラクタに関してはメモリの所有権の管理を release フラグで行います。このフラグが 0 だと、メモリ管理はこのコンストラクタの呼び出し元が行うことを意味します。1 を設定すると sequence のオブジェクトが破棄されるときに内容も同時に破棄されるようになります。

release が 1 に設定されたときには、バッファは allocbuf 関数でアロケートされたものでなければなりません。sequence のクラスは内容ベクタを解放するときに freebuf という関数を使用します。

代入オペレータは、左辺をまず解放し、右辺を左辺に deep-copy します。

デストラクタは release フラグが 1 の場合にはそれぞれの要素をデストラクトし、内容ベクタを freebuf で解放します。

最大長指定なしの sequence に対しては maximum 関数は現在の内容ベクタとしてアロケートされているサイズを返します。これにより、新たなメモリ獲得をせずにどれだけの要素を追加できるかなどの情報を得ることができます。最大長指定ありの sequence の maximum 関数は常に最大長を返します。

サブスクリプトオペレータ(operator[])は指定された場所にある要素を返します。

operator[]を使用する前に必ず length メソッドで長さを指定しておくか、あるいは sequence を T*data コンストラクタで構築しておかなければなりません。

release 関数は sequence クラス内部の release フラグの現在値を返します。

get_buffer 関数を使用すると sequence クラス内部で管理されている領域のポインタを直接得ることができます。orphan 引数が FALSE あるいは const 版の get_buffer 関数を呼び出した場合、領域は引き続き sequence クラスが管理します。このとき、戻り値のポインタは、その sequence クラスオブジェクトに対して、非 const 版の関数が呼ばれるまで有効です。orphan 引数を TRUE で呼び出すと、領域の管理は呼び出し側に移ります。すなわち、get_buffer 関数を呼び出した側で、領域の使用後に freebuf で解放する必要があります。

replace 関数は sequence クラスの内部で管理されている領域を置き換えます。すなわち、replace 関数は、T* data コンストラクタと同様の動作を行います。

sequence の例

IDL で定義された最大長指定なしの sequence と最大長指定ありの sequence に対する C++のコードを以下に示します。

```
// IDL
typedef sequence<T> V1; // 最大長指定なしの sequence
typedef sequence<T, 2> V2; // 最大長指定ありの sequence

// C++
class V1 // 最大長指定なしの sequence
{
        public:
        V1();
        V1(CORBA::ULong max);
        V1(CORBA::ULong max, CORBA::ULong length, T *data,      CORBA::Boolean
release = 0);
        V1(const V1&);
        ~V1();
        V1 &operator=(const V1&);
        CORBA::ULong maximum() const;
        void length(CORBA::ULong);
        CORBA::ULong length() const;
        T &operator[](CORBA::ULong index);
        const T &operator[](CORBA::ULong index) const;
        Boolean release() const;
        void replace(ULong max, Ulong length, T* data,
                 Boolean release = FALSE);
        T* get_buffer(Boolean orphan = FALSE);
        const T* get_buffer() const;
};

class V2 // 最大長指定ありの sequence
{
        public:
        V2();
        V2(CORBA::ULong length, T *data, CORBA::Boolean release = 0);
        V2(const V2&);
        ~V2();
        V2 &operator=(const V2&);
        CORBA::ULong maximum() const;
        void length(CORBA::ULong);
        CORBA::ULong length() const;
        T &operator[](CORBA::ULong index);
        const T &operator[](CORBA::ULong index) const;
        Boolean release() const;
        void replace(ULong length, T* data,
                 Boolean release = FALSE);
        T* get_buffer(Boolean orphan = FALSE);
        const T* get_buffer() const;
};
```

sequence のメモリ管理

```
// IDL
typedef sequence<string, 3> StringSeq;

// C++
char *static_arr[] = {"one", "two", "three"};
char **dyn_arr = StringSeq::allocbuf(3);

dyn_arr[0] = CORBA::string_dup("one");
dyn_arr[1] = CORBA::string_dup("two");
dyn_arr[2] = CORBA::string_dup("three");

StringSeq seq1(3, static_arr); // release フラグは 0
StringSeq seq2(3, dyn_arr, 1);
```

```
seq1[1] = "2"; // 左辺の解放も文字列のコピーも行われない …(1)

char *str = CORBA::string_dup("2");

seq2[1] = str; // 左辺を解放してから str ポインタが代入される
               // コピーは行われない …(2)
```

release フラグが 0 の場合には T*コンストラクタに渡されたバッファ領域の所有権の管理はアプリケーションで行わなければなりません。これに対し 1 が指定された場合には、このメモリ領域の管理は sequence クラスが行うことになります。1 を指定してバッファ領域を sequence に渡した場合にはアプリケーションはこの領域が存在しつづける期間についていかなる仮定もしてはいけません。sequence はより大きなバッファ領域が必要となったときなどにこの領域を解放し、新たに大きな領域を割り当ててそこに値をコピーすることが可能だからです。0 を指定した場合には sequence がバッファを自動的に解放することはありませんが、大きなバッファ領域が必要となったときに新たな領域を割り当てることは起こり得ます。

T* data コンストラクタを使用して構築された、要素として string あるいはオブジェクトリファレンスを持つ sequence の operator[]は、コンストラクタに渡された release フラグによって動作が変わります。上記の例で(1)の代入は sequence の release フラグに 0 が指定されていますので、要素の代入のときに左辺の古い文字列の解放は行わず、右辺のポインタの値が直接代入されます。これに対して release フラグを 1 で指定した(2)の代入では、sequence がメモリ管理の責任を持つため、代入に先立ち左辺の古い文字列を解放し、その後ポインタが代入されます。(1)の例でも(2)の例でもポインタが直接代入されることには注意を要します。

```
seq2[1]="bad habit";
```

このような行は動的にアロケートされたメモリでないものがポインタとして sequence に代入されますが、sequence はそのメモリを解放する責任がありますから、sequence が破棄されるときにバッファのメモリおよび要素のメモリを解放します。この例では"bad habit"という文字列を解放してしまうことになり、予期しない場所でアプリケーションが実行時エラーを起こすことになります。

上記の文字列の例はオブジェクトリファレンスの場合も同様です。

一般的に release フラグは 1 に設定して使用するとメモリリークを起こしにくいプログラムを作成することができますが、0 に設定するときには注意深く行う必要があります。

sequence バッファの獲得/解放

sequence に設定するバッファは次のメソッドで獲得/解放を行わなくてはいけません。

```
//IDL
typedef sequence<T> ...;

//C++
static T * allocbuf(CORBA::ULong nelm);
static void freebuf(T*);
```

sequence<T>を typedef を使用して IDL 定義すると、対応する C++のクラスには上記のように allocbuf、freebuf というスタティックでパブリックなメンバ関数が定義されます。バッファの獲得は allocbuf を使用します。allocbuf の引数は要素数です。メモリが獲得できなかった場合にはヌル・ポインタを返します。freebuf は allocbuf で獲得したメモリを解放します。

allocbuf で獲得された領域のそれぞれの要素のオブジェクトに対してはデフォルトコンストラクタが呼び出されます。ただし、文字列の要素の場合にはヌル・ポインタが、オブジェクトリファレンスを要素として持つ場合には、要素の型を持つ nil オブジェクトリファレンスが設定されます。

配列型

配列は C++の配列にマッピングされます。配列の要素が文字列あるいはオブジェクトリファレンスの場合には、要素に対して代入が行われたときには古い値は自動的に解放されます。

```
// IDL
typedef float F[10];
typedef string V[10];
typedef string M[1][2][3];

void op(out F p1, out V p2, out M p3);

// C++
```

```
F f1; F_var f2;
V v1; V_var v2;
M m1; M_var m2;

op(f2, v2, m2);

f1[0] = f2[1];
v1[1] = v2[1]; // 古い値を解放し、新しい値をコピーします
m1[0][1][2] = m2[0][1][2]; // 古い値を解放し、新しい値をコピーします
```

上記の最後の 2 行では、代入前に左辺の要素に格納されている値は解放されてから、右辺の値がコピーされます。

配列が out 引数および返り値に使用されている場合には、その配列の slice 型へのポインタに対応づけられます。IDL コンパイラは配列の定義に対して、slice 型の typedef を出力します。slice 型の名前は配列の名前に_slice をつけたものになります。_slice 型は配列の次元から最初の次元を取り除いた配列に対する typedef として定義されます。

```
// IDL
typedef long LongArray[4][5];

// C++
typedef CORBA::Long LongArray[4][5];

// IDL コンパイラによって出力された_slice 型の定義
typedef CORBA::Long LongArray_slice[5];
```

IDL コンパイラは配列名に_var という名前をつけたクラスも出力します。T_var 型はオーバロードされた operator[]を持っており、T_var 型をあたかも通常の C++の配列のように扱うことができます。また、T_var 型は_slice 型を引数にとるコンストラクタと代入オペレータも持っています。配列のアクセスに T_var 型を使用すると、配列領域のメモリ管理を自動的に行ってくれるので、メモリ解放忘れなどが少なくなります。前の例の IDL 定義では次のようなクラスが出力されます。

```
// C++
class LongArray_var
{
        public:
        LongArray_var();
        LongArray_var(LongArray_slice*);
        LongArray_var(const LongArray_var &);
        ~LongArray_var();
        LongArray_var &operator=(LongArray_slice*);
        LongArray_var &operator=(const LongArray_var &);
        LongArray_slice &operator[](CORBA::ULong index);
        const LongArray_slice &operator[](CORBA::ULong index) const;
        ...

};
```

配列型に対しては CORBA::Any 型への挿入や取り出しのために_forany というサフィックスのついたクラスも定義されます。Array_forany 型も格納されている配列に対して Array_var 同様にアクセスすることができます。ただし Array_forany は Array_var と異なり、Array_forany の破壊時に、配列のメモリ領域が自動的に解放されることはありません。Arra_slice*型を引数にとる Array_forany のコンストラクタは、nocopy という CORBA::Boolean 型の引数をとり、この引数はデフォルトが 0 となっています。

nocopy フラグを 1 にすると any への Array_slice の挿入時にコピーをしないように指定できます。

```
// C++
class Array_forany
{
        public:
        Array_forany(Array_slice*, CORBA::Boolean nocopy = 0);
        ...

};
```

配列を動的にアロケートするためには、アプリケーションは特別な関数を使用しなければなりません。配列が T 型として定義されているとすると、次の関数が IDL コンパイラによって生成されます。

```
// C++
T_slice *T_alloc();
T_slice *T_dup(const T_slice*);
void T_free(T_slice *);
```

T_alloc 関数は配列を動的にアロケートします。もしアロケートに失敗したときにはヌル・ポインタが返されます。T_dup は動的に同じサイズの新しい配列をアロケートし、要素を新しい配列にコピーします。アロケートに失敗したときにはヌル・ポインタが返されます。T_free 関数は T_alloc あるいは T_dup でアロケートされた領域を解放します。T_free にヌル・ポインタを渡しても構いません。この場合は何の動作も行われません。

配列の領域をアロケートするときには T_alloc を使用することに注意してください。T_alloc を使用しない場合には、予期しない場所でアプリケーションエラーとなる危険があり、原因の究明が非常に困難になる場合があります。

typedef

typedef は型に対する別名を作ります。オリジナルの型が C++でいくつかの型に対応づけられる場合には、typedef はそれぞれの型に対して対応する別名を作ります。次に例を示します。

```
// IDL
typedef long T;
interface A1;
typedef A1 A2;
typedef sequence<long> S1;
typedef S1 S2;

// C++
typedef CORBA::Long T;

//A1 の定義
typedef A1 A2;
typedef A1_ptr A2_ptr;
typedef A1_var A2_var;

// S1 の定義
typedef S1 S2;
typedef S1_var S2_var;
```

配列などのように複数の C++型に対応づけられるような IDL 型に対しては、typedef によりすべての C++型および関数に対して、C++の typedef が生成されます。

以下に例を示します。

```
// IDL
typedef long array[10];
typedef array another_array;

// C++
// 配列自体に対する typedef
typedef array another_array;

// _var に対する typedef
typedef array_var another_array_var;

// _slice に対する typedef
typedef array_slice another_array_slice;

// _forany に対する typedef
typedef array_forany another_array_forany;

// 関数に対しても alias 名でアクセスできるように
// another_array_alloc()が定義される
```

```
another_array_slice *another_array_alloc() {
        return array_alloc();
}

another_array_slice*another_array_dup(another_array_slice *a) {
        return array_dup(a);
}

void another_array_free(another_array_slice *a) {
        array_free(a);
}
```

any

IDL の any 型は IDL の任意の型を、安全に挿入および取り出しのできる型です。any には IDL から生成された C++を挿入したり、あるいは、コンパイル時には型のわかっていないデータを処理する機能があります。コンパイル時にわからない型が any に挿入されているというケースは、たとえば、受け取る側にはあらかじめわからない型のデータが挿入されている any をリクエストあるいはレスポンスとして受け取った場合などに起こります。

any に対して、その中に格納されている値と、型を識別する typecode との不一致が起こることを避けるために IDL で定義された個々の型に対して、挿入と取り出しのオーバーロードされた関数が生成されます。生成された関数を使用して any へ値を挿入することにより、値と typecode が一致しないというケースや、実際に挿入されている型と異なった型として any から値を取り出すというケースを防ぐことができます。オーバロード関数で区別がつくように、IDL から C++への変換時には、個々の IDL の型は、C++として異なった型に変換されるようになっていますが次のようなケースでは C++の型だけでは区別ができません。

・IDL の boolean, octet, char は互いに区別できない C++の型に対応づけられます。したがって、any への挿入や取り出しには、これらの型を区別して行う方法が必要になります。

・文字列(string)はすべて char*型に対応づけされるので、長さ制限付きの文字列と(bounded string)と長さ制限なしの文字列(unbounded string)を C++の型から区別することができません。これらの型を区別して扱う方法が必要になります。

・C++では、配列が関数の引数リストで使用される時、先頭の要素へのポインタとなります。したがって、異なる長さの配列を、型で区別することができません。したがって、any への挿入や取り出しには、長さの異なる配列を区別できるような方法が必要になります。

C++マッピングでは上記のことが問題なく行えるようなしくみを提供しています。

any への値の挿入方法

IDL の T 型に対して IDL コンパイラは次のようなオーバーロードされた関数を出力します。

```
// C++
void operator<<=(CORBA::Any&, T);
```

このタイプの関数は short, unsigned short, long, unsigned long, float, double, 列挙型、長さ制限なしの文字列(unbounded string)、オブジェクトリファレンス型に対して用意されます。

コピーすると性能を落とすような型に対しては、次の挿入関数が生成されます。

```
// C++
void operator<<=(CORBA::Any&, const T&); // コピーが行われる。
void operator<<=(CORBA::Any&, T*); // コピーは行われない。
```

これらの<<=オペレータを使用して any に値を挿入することができます。次に例を示します。

```
// C++
CORBA::Long value = 42;
CORBA::Any a;
a <<= value;
```

上記の例では any に CORBA::Long の値として 42 が設定されると同時に、CORBA::Long をあらわす TypeCode が any に設定されます。

コピーが行われないタイプの<<=が使用されたときには、格納されたオブジェクトの所有権は any に移ります。つまり any がデストラクトされると格納されているオブジェクトも自動的に解放されます。any に格納した

後で、元のオブジェクトをプログラムで明示的に解放すると 2 重解放が起こることになり、アプリケーションは予期しない場所でクラッシュする可能性が出てきますので注意する必要があります。また、挿入した後でオブジェクトにアクセスしてもいけません。

```
// IDL
struct s {
       string ss;
};

// C++
CORBA::Any a;
sp = new s;
a <<= sp; // 所有権が any に移る。
delete sp; // 行ってはいけない。2 重解放が起こる。
char *s = sp->ss; // 行ってはいけない。
                  // any に挿入した後は sp 経由でアクセスを行ってはいけない。
```

コピーされるタイプでもコピーされないタイプの関数でも、<<=オペレータは any に格納されていた古い値は自動的に解放されます。

配列の any への挿入

配列の any への挿入取り出しは Array_forany 型を通して行います。

```
// IDL
typedef long LongArray[4][5];

// C++
typedef CORBA::Long LongArray[4][5];
typedef CORBA::Long LongArray_slice[5];

class LongArray_forany { ... };

void operator<<=(CORBA::Any &, const LongArray_forany &);
```

Array_forany 型は常に<<=に対して const へのリファレンスとして渡されます。any に値がコピーされるのかされないのかは Array_forany のコンストラクタで制御されます。コンストラクタの nocopy 引数に 1 を指定したときにはコピーは行われず、メモリ領域の所有権は any に移ります。0 を指定したときにはコピーが行われます。

```
// IDL
struct S {... };
typedef S SA[5];

// C++
struct S { ... };

typedef S SA[5];
typedef S SA_slice;

class SA_forany { ... };

SA s;

// ...s を初期化...

CORBA::Any a;

a <<= s; // 1 行目
a <<= SA_forany(s); // 2 行目
```

上記の例で 1 行目ではコピーされないタイプの operator<<=(CORBA::Any&,S*)が呼ばれます。これは配列の挿入ではなく、単なる構造体の挿入になり、意図した動作にはなりません。配列を挿入するという意図した動作を行うためには 2 行目のように一度 SA_forany 型を作ってから any に挿入する必要があります。

オブジェクトリファレンスに対しては次のオペレータが定義されます。

```
// IDL
interface T { ... };

// C++
void operator<<=(CORBA::Any&, T_ptr); // コピーされるタイプ
void operator<<=(CORBA::Any&, T_ptr*); // コピーが行われないタイプ
```

コピーが行われないタイプのオブジェクトリファレンスの挿入においては、所有権が any に移ります。したがって、挿入後はオブジェクトリファレンスに対して CORBA::release を呼び出したり、このオブジェクトリファレンスを使用してオブジェクトの参照を行ってはいけません。これに対してコピーが行われるタイプのオペレータを使用すると、オブジェクトリファレンスに対して CORBA::Object::_duplicate が呼び出されます。

any からの値の取り出し

IDL の T 型に対して any からの取り出しのための次のようなオペレータが用意されます。

```
// C++
CORBA::Boolean operator>>=(const CORBA::Any&, T&); // コピーされる
```

このオペレータは、boolean, char, octet, short, unsigned short, long, unsigned long, float, double, 列挙型, 長さ制限のない文字列型(unbounded string), オブジェクトリファレンスに対して使用されます。これら以外の型に対しては、次のオペレータが用意されます。

```
// C++
// コピーは起こらない
CORBA::Boolean operator>>=(const CORBA::Any&, T*&);
```

取り出しのメソッドを使用して次のように値を取り出すことができます。

```
// C++
CORBA::Long value;
CORBA::Any a;

a <<= CORBA::Long(42);

if (a >>= value) {
        // 値を使用する。
}
```

上記の例で、オペレータ>>=が呼び出されると、挿入されている値が実際に CORBA::Long 型かどうかチェックされます。CORBA::Long 型の場合には値が value にコピーされ、返り値として 1 を返します。any の中に入っているのが CORBA::Long ではなかった場合には value の値は変更されず、オペレータは 0 を返します。

コピーが起こるタイプの型に対しては取り出しはポインタで行われます。

```
// IDL
struct MyStruct {
        long lmem;
        short smem;
};
```

上記のような struct は次のようにして取り出すことができます。

```
// C++
CORBA::Any a;

// a に MyStruct の値を設定

MyStruct *struct_ptr;

if (a >>= struct_ptr) {
        // 値を使用可能
}
```

取り出しが成功すると呼び出し元は any が所有権を持つメモリ領域へのポインタを獲得し、>>=は 1 を返しま

す。呼び出し元は返されたポインタを delete したり CORBA::release したりしてはなりません。ポインタは any の管理するポインタを得るだけですので、any の値を変更するような操作を行った後では、ポインタを使用してアクセスすることをしてはなりません。特に T_var 型を使用するときには注意が必要です。T_var は格納されているポインタを自動的に解放するからです。

取り出しに失敗するとヌル・ポインタが設定され、>>=は 0 を返します。

配列の取り出しは次のように、Array_forany という補助型を使用して行います。array を IDL で定義すると次のような補助型が IDL コンパイラによって生成されます。

```
// IDL
typedef long A[20];
typedef A B[30][40][50];

// C++
typedef CORBA::Long A[20];
typedef CORBA::Long A_slice;

class A_forany { ... };

typedef A B[30][40][50];
typedef A B_slice[40][50];

class B_forany { ... };

CORBA::Boolean operator>>=(const CORBA::Any &, A_forany&);

CORBA::Boolean operator>>=(const CORBA::Any &, B_forany&);

CORBA::Any a;
A_forany arr;

// a には A 型の配列が入っているとする
if (a >>= arr) {
        // arr が使用可能
}
```

>>=オペレータは CORBA::Any_var 型にも定義されています。

boolean, octet, char, 長さ制限付きの文字列(Bounded String)の区別

boolean, octet, char, 長さ制限付きの文字列の any への挿入取り出しのために CORBA::Any には次の補助型が定義されています。

```
// C++
class CORBA::Any
{
        public:
        // boolean の挿入のための補助型
        struct from_boolean {
                from_boolean(CORBA::Boolean b) : val(b) {}
                CORBA::Boolean val;
        };

        // octet の挿入のための補助型
        struct from_octet {
                from_octet(CORBA::Octet o) : val(o) {}
                CORBA::Octet val;
        };

        // char の挿入のための補助型
        struct from_char {
                from_char(CORBA::Char c) : val(c) {}
                CORBA::Char val;
        };
```

```
        // 文字列の挿入のための補助型
        struct from_string {
                from_string(char* s, CORBA::ULong b, CORBA::Boolean nocopy
        = 0) :
                val(s), bound(b) {}
                char *val;
                CORBA::ULong bound;
        };

        void operator<<=(from_boolean);
        void operator<<=(from_char);
        void operator<<=(from_octet);
        void operator<<=(from_string);

        // boolean の取り出しのための補助型
        struct to_boolean {
                to_boolean(CORBA::Boolean &b) : ref(b) {}
                CORBA::Boolean &ref;
        };

        //char の取り出しのための補助型
        struct to_char {
                to_char(CORBA::Char &c) : ref(c) {}
                CORBA::Char &ref;
        };

        // octet の取り出しのための補助型
        struct to_octet {
                to_octet(CORBA::Octet &o) : ref(o) {}
                CORBA::Octet &ref;
        };

        // 文字列の取り出しのための補助型
        struct to_string {
                to_string(char *&s, CORBA::ULong b) : val(s), bound(b)  {}
                char *&val;
                CORBA::ULong bound;
        };

        // CORBA::Object 型の取り出しのための補助型
        struct to_object {
                to_object(CORBA::Object_ptr &obj) : ref(obj) {}
                CORBA::Object_ptr &ref;
        };

        CORBA::Boolean operator>>=(to_boolean) const;
        CORBA::Boolean operator>>=(to_char) const;
        CORBA::Boolean operator>>=(to_octet) const;
        CORBA::Boolean operator>>=(to_string) const;
        CORBA::Boolean operator>>=(to_object) const;
};
```

上記の補助型を使用して実際の挿入取り出しは次のように行うことができます。

```
// C++
CORBA::Boolean b = 1;
CORBA::Any any;

any <<= CORBA::Any::from_boolean(b);

// ...
```

```
if (any >>= CORBA::Any::to_boolean(b)) {
        // ...boolean の取り出し成功
}

char* p = "bounded";

any <<= CORBA::Any::from_string(p, 8); // 長さ制限付き文字列の挿入

// ...

if (any >>= CORBA::Any::to_string(p, 8)) {
        // ...文字列取り出し成功
}
```

取り出しに成功した場合、指している先のオブジェクトの所有権は any にあります。any の有効期間と独立に値を使用する場合には明示的にコピーや_duplicate を行う必要があります。

from_string の第 2 引数が 0 のときは長さ制限なしの文字列をあらわします。from_string のコンストラクタの nocopy で 1 が指定されていると挿入後 any が文字列の所有権を獲得します。

```
// C++
char* p = CORBA::string_alloc(8);

// ...p を初期化...

any <<= CORBA::Any::from_string(p, 8, 1); // any が p の所有権を管理
```

その他の any の操作

any には前節以外に、型に関してチェックを行わない関数があります。これらを使用するとコンパイル時には知らない型の操作を行うことも可能になりますが、any の値とその型を示す TypeCode の一貫性を保つ責任はプログラマが持つことになります。

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

このコンストラクタは指定した TypeCode と値が設定された any を作ります。TypeCode は Any 内で_duplicate されます。release 引数が 1 の場合は any が所有権を管理することになり、any の消滅時や他の値が挿入されたときなどには、古い値を any が解放します。any に値を挿入した後では、その値を操作してはいけません。

```
// C++
void replace(
        CORBA::TypeCode_ptr,
        void *value,
        CORBA::Boolean release = 0
);
CORBA::TypeCode_ptr type() const;
const void *value() const;
```

replace 関数は any の中に挿入されている値を指定されたもので置き換えます。古い TypeCode は_release され、必要な場合は古い値は解放されます。

type()は any に挿入されている型の TypeCode を返します。返された TypeCode は呼び出し元で_release する必要があります。

value()関数は any に格納されている値を取り出します。値が入っていないときにはヌル・ポインタを返します。

any のコンストラクタ・デストラクタ・代入オペレータ

any のデフォルトコンストラクタは TypCode として tk_null、値として何も持たない any を作成します。コピーコンストラクタは TypeCode に対して_duplicate を行い、値に関しては deep-copy を行います。代入オペレータでは、まず、古い TypeCode は_release し、値も必要に応じて解放します。次に新しい TypeCode を_duplicate したのち値を deep-copy します。

デストラクタでは TypeCode を_release し、必要な場合には値を解放します。

CORBA::Any_var クラス

any にも構造体の T_var などと同様の CORBA::Any_var というクラスが用意されています。any を CORBA::Any_var に格納すると any の領域は CORBA::Any_var が消滅するときに自動的に削除されるようになります。

例外

インタフェース固有の例外(ユーザ定義例外)を定義することができます。例外を定義するには exception を使います。また、定義された例外を返すにはオペレーションの後ろに raises を指定します。raises には複数のユーザ定義例外を指定することができます。

例外には、詳細な情報を得るためにメンバを持たせることができます。

下記に例を示します。

```
// IDL
module M {
        interface Foo {
                // 例外の理由を表す enum
                enum Reason { reason_A, reason_B, reason_C };

                // メンバを持たない例外
                exception UserDefinedException1 {};

                // enum と unsigned short のメンバを持つ例外
                exception UserDefinedException2 {
                        Reason m1;
                        unsigned short m2;
                };

                op() raises(UserDefinedException1, UserDefinedException2);
        };
};
```

IDL で定義された例外は C++の CORBA::UserException を継承するクラスに対応づけられます。このクラスは可変長 struct で対応づけられるクラスと同様に使用します。

```
// C++
namespace M {
        class Foo {
                ...

                class UserDefinedException1 : CORBA::UserException {
                        ...

                };
                ...

        };
};
```

標準例外は CORBA::SystemException から継承されるクラスに対応づけられます。
CORBA::SystemException クラスは次のような定義になっています。

```
// C++
enum CORBA::CompletionStatus {
        CORBA::COMPLETED_YES,
        CORBA::COMPLETED_NO,
        CORBA::COMPLETED_MAYBE
};

class CORBA::SystemException : public CORBA::Exception
{
        public:
        CORBA::SystemException();
```

```
        CORBA::SystemException(const CORBA::SystemException &);
        CORBA::SystemException(CORBA::ULong minor, CORBA::CompletionStatus status);
        ~CORBA::SystemException();

        CORBA::SystemException &operator=(const CORBA::SystemException &);
        CORBA::ULong minor() const;
        void minor(CORBA::ULong);
        void _raise();
        CORBA::CompletionStatus completed() const;
        void completed(CORBA::CompletionStatus);
};
```

CORBA::SystemException のデフォルトコンストラクタで作成されたオブジェクトは minor()で 0 を返し、completed()は CORBA::COMPLETED_NO を返します。

_raise()は自分自身を throw するための関数です。CORBA::Exception クラスは下記のとおりに定義されています。

```
// C++
class Exception
{
        public:
        virtual ~Exception();
        virtual void _raise() = 0;
};
```

個々の例外では、_raise()を多重定義して自分自身を throw するようにしています。_raise()の実装は下記のとおりです。

```
// C++
void IndividualException::_raise()
{
        throw *this;
}
```

個別のシステム例外は CORBA::SystemException を継承して定義されています。次に例を示します。

```
// C++
class CORBA::UNKNOWN : public CORBA::SystemException { ... };
class CORBA::BAD_PARAM : public CORBA::SystemException { ... };
```

コンパイル時に型が決定していないユーザ例外を受け取った場合には CORBA::UnknownUserException を返します。exception()というメンバ関数を持っており、これにより返された any から実際の例外情報を取り出すことができます。

```
// C++
class CORBA::UnknownUserException : public CORBA::UserException
{
        public:
        CORBA::Any &exception();
};
```

例外の調べ方

オペレーション呼び出しの後で、例外が存在したかどうか、またどの例外が発生したかを調べるには次のようにします。

```
// C++
A_var a = ...;

try {
        a->op(..., env);
} catch (CORBA::SystemException& sys) {
        // システム例外
} catch (XX& xx) {
        // XX 例外
}
```

catch した例外は解放してはいけません。

オペレーションとアトリビュート

オペレーションは、オペレーションと同じ名前の C++関数に対応づけられます。read-write のアトリビュートは C++の二つのオーバーロードされた関数に対応づけられます。一つは値をセットするためのものでもう一つは値を得るためのものです。セットする関数は、アトリビュートと同じ型の in 引数を一つとり、値を得る関数は、同じ型を返す引数のない関数になります。

readonly のアトリビュートは値を得る関数のみが定義され、セットする関数は定義されません。アトリビュート関数に対する引数と返り値の扱いは、通常のオペレーションの引数の扱いのルールと同じです。

oneway オペレーションも通常のオペレーションと同様に対応づけられます。

```
// IDL
interface A
{
        void f();
        oneway void g();
        attribute long x;
};

// C++
A_var a;
CORBA::Environment env;

a->f(env);
a->g(env);
CORBA::Long n = a->x(env);
a->x(n + 1, env);
```

IDL で定義したオペレーション以外に、CORBA::Environment 型の引数を最後の引数として必ずとります。例外が起こった場合には、この変数に情報が設定されます。

オペレーションに対する暗黙の引数

IDL のオペレーション定義に Context が使用されている場合には、オペレーションの最後から 2 番目の引数が CORBA::Context となります。最後の引数は CORBA::Environment 型の引数です。

引数渡しのモード

本節では、オペレーションの引数として与えるオブジェクトに対して、誰がその領域を取得し、その領域を解放するか、また、オペレーションの型は何型で渡されるのかを示します。

基本型や列挙型などでは、引数渡しの方法は単純で、Pという基本型や列挙型を渡すときにはPという型を直接渡し、オブジェクトリファレンスの場合には A_ptr というように_ptr 型で渡します。

複雑な型は誰がメモリを獲得/解放するかの問題があり、複雑になります。in 引数に関しては、呼び出し元がメモリを獲得し、読み出しのみなので単純です。out と inout に関しては、少し複雑です。

複雑な型に関しては、固定長の型に関しては、T&型となり、可変長の型に対しては、T*&となります。struct 等では固定長の場合と可変長の場合で型が違ってくることになり注意が必要ですが、T_var を使用するとどちらの場合でも同じように、 T_var&と考えて渡すことができます。

out, inout 引数の場合には、オペレーション呼び出し前に、変数に格納されている領域を解放する必要があるときがあります。

```
// IDL
struct S { string name; float age; };
void f(out S p);

// C++
S_var s;
f(s);

// s を使用
f(s); // はじめの呼び出し結果が解放される
S *sp; // out に渡すときには初期化不要
```

```
f(sp);
// sp を使用
delete sp; // 次のオペレーションでは sp は解放されない
f(sp);
```

上記の例のように T_var を使用していれば、呼び出し前に変数に格納されている値を解放する必要はありませんが、そうでない場合には解放する必要がある場合がありますので注意する必要があります。

```
// IDL
void q(out string s);

// C++
char *s;
for (int i = 0; i < 10; i++)
        q(s); // メモリリーク
```

上記の例では out で返されたメモリ領域が必ずリークしています。これを防ぐには二つの方法があります。

```
// C++
char *s;

CORBA::String_var svar;

for (int i = 0 ; i < 10; i++) {
        q(s);
        CORBA::string_free(s); // 明示的に解放

        // または

        q(svar); // 暗黙的に解放される
}
```

out 引数に char*が使われるときには呼び出し元でそれを解放しなければなりませんが、CORBA::String_var を使用することによって解放を自動的に行わせることができます。コンパイル時、実行時ともにチェックは行われませんが、呼び出し元では、呼び出しによって返された引数等に対して変更を行うことは禁止されています。変更を行うときには、一旦別の領域にコピーしてから、新しい領域を変更して使用しなければなりません。

可変長データは、書き換える前に明示的に解放する必要があります。たとえば、inout の文字列型の引数に対して代入する前に、オペレーションの実装内で、古い文字列データを解放する必要があります。同様に inout のインタフェース型の引数は古い値を CORBA::release()する必要があります。

これらのことを確実にするために T_var 型のローカル変数に代入してしまうことができます。

```
// IDL
interface A;

void f(inout string s, inout A obj);

// C++

void Aimpl::f(char *&s, A_ptr &obj) {
        CORBA::String_var s_tmp = s;
        s = /* new data */;
        A_var obj_tmp = obj;
        obj = /* new reference */;
}
```

ポインタあるいはポインタへのリファレンス(T*&)で返される引数に関しては、ヌル・ポインタを設定してはいけません。設定した場合の挙動は未定義です。以下のどの場合においてもヌル・ポインタを設定することは禁止されています。

・in と inout での文字列

・in と inout での配列

呼び出し元は out 引数の値としてヌル・ポインタを使用することは許されています。

呼び出される側(オペレーションの実装側)は以下のすべてにおいてヌル・ポインタを返してはいけません。

・out と返り値での可変長 struct

・out と返り値での可変長 union

・out と返り値での文字列

・out と返り値でのシーケンス

・out と返り値での可変長配列、および返り値での固定長配列

・out と返り値での any

表 1 にそれぞれの IDL 型に対する、引数と返り値の C++の型を示します。

表 1 引数と返り値の型

Data Type	In	Inout	Out	Return
short	CORBA::Short	CORBA::Short&	CORBA::Short&	CORBA::Short
long	CORBA::Long	CORBA::Long&	CORBA::Long&	CORBA::Long
long long	CORBA::LongLong	CORBA::LongLong&	CORBA::LongLong&	CORBA::LongLong
unsigned short	CORBA::UShort	CORBA::UShort&	CORBA::UShort&	CORBA::UShort
unsigned long	CORBA::ULong	CORBA::ULong&	CORBA::ULong&	CORBA::ULong
unsigned long long	CORBA::ULongLong	CORBA::ULongLong&	CORBA::ULongLong&	CORBA::ULongLong
float	CORBA::Float	CORBA::Float&	CORBA::Float&	CORBA::Float
double	CORBA::Double	CORBA::Double&	CORBA::Double&	CORBA::Double
long double	CORBA::LongDouble	CORBA::LongDouble&	CORBA::LongDouble&	CORBA::LongDouble
boolean	CORBA::Boolean	CORBA::Boolean&	CORBA::Boolean&	CORBA::Boolean
char	CORBA::Char	CORBA::Char&	CORBA::Char&	CORBA::Char
wchar	CORBA::WChar	CORBA::WChar&	CORBA::WChar&	CORBA::WChar
octet	CORBA::Octet	CORBA::Octet&	CORBA::Octet&	CORBA::Octet
enum	enum	enum&	enum&	enum
object reference ptr	objref_ptr	objref_ptr&	objref_ptr&	objref_ptr
struct, fixed	const struct&	struct&	struct&	struct
struct, variable	const struct&	struct&	struct*&	struct*
union, fixed	const union&	union&	union&	union

union, variable	const union&	union&	union*&	union*
string	const char*	char*&	char*&	char*
wstring	const CORBA::WChar*	CORBA::WChar*&	CORBA::WChar*&	CORBA::WChar*
sequence	const sequence&	sequence&	sequence*&	sequence*
array, fixed	const array	array	array	array slice*
array, variable	const array	array	array slice*&	array slice*
any	const any&	any&	any*&	any*
fixed	const fixed&	fixed&	fixed&	fixed
valuetype	valuetype*	valuetype*&	valuetype*&	valuetype*

表 2 呼び出し元での、メモリ領域の解放に関する決まり

型	inout	out	return
short	1	1	1
long	1	1	1
long long	1	1	1
unsigned short	1	1	1
unsigned long	1	1	1
unsigned long long	1	1	1
float	1	1	1
double	1	1	1
long double	1	1	1
boolean	1	1	1
char	1	1	1
wchar	1	1	1
octet	1	1	1
enum	1	1	1
object reference ptr	2	2	2
struct, fixed	1	1	1
struct, variable	1	3	3
union, fixed	1	1	1
union, variable	1	3	3

string	4	3	3
wstring	4	3	3
sequence	5	3	3
array, fixed	1	1	6
array, variable	1	6	6
any	5	3	3
fixed	1	1	1
valuetype	7	7	7

表 2 の数字の説明

1. 呼び出し元が領域を確保します。inout 引数に対しては呼び出し元が初期値を提供し、呼び出し先はその値を変えてもかまいません。out 引数に関しては呼び出し元が領域を確保しますが、初期化の必要はなく、値は呼び出し先で設定します。関数の戻り値は値渡しで返されます。
2. 呼び出し元がオブジェクトリファレンスの領域を確保します。inout 引数に関しては呼び出し元は初期値を提供します。呼び出し先において inout 引数の値を変更したいときには、入力値に対して CORBA::relase を行い、返却値のための領域を確保しなおします。inout で渡した値を使い続けるためには、呼び出し元は最初にリファレンスを_duplicate する必要があります。呼び出し元は out と return のオブジェクトリファレンスを release する必要があります。その他の構造中に埋め込まれているオブジェクトリファレンスに関してはその構造自体が自動的に release を行います。
3. out 引数に関しては、呼び出し元がポインタを用意し、呼び出し先に参照渡しで渡します。呼び出し先はそのポインタに正しい引数の型のインスタンスへのポインタをセットします。return に対しては呼び出し先は同様にポインタを返します。呼び出し先ではどちらの場合でもヌル・ポインタを返してはいけません。どちらの場合でも、返された領域を解放する責任を持つのは呼び出し元です。ローカルでもリモートでも同じように処理できるように、呼び出し元では、呼び出し元が呼び出し先と同じアドレス空間にあるか別の空間にあるかに関わりなく、返された領域を常に解放する必要があります。要求の完了後、呼び出し元は返された領域の値を変更してはいけません。変更したいときには呼び出し元で、返された値の新しいコピーを作成しその値を変更する必要があります。
4. inout(ワイド)文字列に関しては、呼び出し元が文字列の領域を用意しそのポインタを渡します。呼び出し先では渡された文字列を解放し新しい領域を指すポインタを代入してもかまいません。つまり、返される文字列の長さは入力として渡される文字列の長さで制限されることはありません。呼び出し先で領域が解放されることを考慮し、呼び出し元では文字列の領域を CORBA::(w)string_alloc()を使って確保する必要があります。呼び出し元では、out で返された文字列を CORBA::(w)string_free()を使用して解放する必要があります。呼び出し先では inout,out,return に対してヌル・ポインタを返してはいけません。
5. inout のシーケンスと any に関しては、シーケンス/any に対する代入や変更のときに、シーケンス/any が構築されたときの boolean 型の引数 release の設定に依存して、その内部に格納されている領域が解放されます。
6. out 引数に関しては、呼び出し元が array slice へのポインタを用意し、そのポインタを参照渡しで呼び出し先に渡します。呼び出し先では配列の正しいインスタンスへのポインタを設定します。返り値に対しては、呼び出し先は同様にポインタを返します。どちらの場合でも、呼び出し先ではヌル・ポインタを返してはいけません。どちらの場合でも呼び出し元では返された領域を解放する責任があります。ローカル/リモートで同様に処理できるようにするため、呼び出し元では、返された領域を常に解放する必要があります。リクエストの完了後、呼び出し元は返された領域を変更してはいけません。変更したい場合にはまずコピーを作成しそのコピーを変更する必要があります。
7. 呼び出し元が valuetype 型のインスタンスの領域を確保します。inout 引数に関しては呼び出し元は初期値を提供します。呼び出し先において inout 引数の値を変更したいときには、入力値に対して remove_ref を行い、返却値のための領域を確保しなおします。inout で渡した値を使い続けるためには、呼び出し元はオペレーションの呼び出し後の最初に_add_ref を呼び出す必要があります。呼び出し元は out と return の valuetype 型のインスタンスに対して、_remove_ref を呼び出す必要があります。その他の構造中に埋め込まれている valuetype 型のインスタンスに関してはその構造自体が自動的に_remove_ref を行いま

す。

CORBA2.2 で新たに加わった型

固定小数点数(Fixed Types)

IDL の fixed 型は以下のテンプレートで表される抽象データ型にマップされます。

```
// C++ class template
template<CORBA::UShort d, Short s>
class CORBA::Fixed
{
public:
      // Constructors...
      CORBA::Fixed(int val = 0);
      CORBA::Fixed(CORBA::LongDouble val);
      CORBA::Fixed(const CORBA::Fixed<d,s>& val);
      ~CORBA::Fixed();
      // Conversions...
      operator LongDouble() const;
      // Operators...
      CORBA::Fixed<d,s>& operator=(const CORBA::Fixed<d,s>& val);
      CORBA::Fixed<d,s>& operator++();
      CORBA::Fixed<d,s>& operator++(int);
      CORBA::Fixed<d,s>& operator--();
      CORBA::Fixed<d,s>& operator--(int);
      CORBA::Fixed<d,s>& operator+() const;
      CORBA::Fixed<d,s>& operator-() const;
      int operator!() const;
      // Other member functions
      CORBA::UShort fixed_digits() const;
      CORBA::Short fixed_scale() const;
};
template<CORBA::UShort d, CORBA::Short s>
istream& operator>>(istream& is, CORBA::Fixed<d,s>& val);
template<CORBA::UShort d, CORBA::Short s>
ostream& operator<<(ostream& os, const CORBA::Fixed<d,s>& val);
```

Fixed T_var 型

他の型と同様に Fixed 型には T_var 型が定義されます。Fixed の T_var 型のセマンティックスは固定長構造体の T_var 型と同様です。

abstract interface 型

IDL abstract interface は、オブジェクトリファレンス型とバリュー型のどちらも扱うことが可能です。そのため、以下の点が通常の interface と異なります。 IDL abstract interface は通常の IDL interface と同様に、同じ名前のクラスにマッピングされますが、CORBA::Object を継承せず、CORBA::AbstractBase を継承します。また、通常の IDL abstract interface は通常の IDL interface を継承できません。

```
// IDL
abstract interface Absl1 {
      void op1();
};

abstract interface Absl2 : Absl1 {
      void op2();
};

// C++
class Absl1...
{
public :
```

```
        void op1(…)
}

class Abs12 : public virtual Abs11
{
public :
        void op2(…)
}
```

バリュー型(valuetype)

バリュー型(IDL valuetype)は、オブジェクトを参照(オブジェクトリファレンス)ではなく、値で渡すことができます。バリュー型は、CORBA 呼び出しに渡されると、そのステート(データメンバ)は、構造体のメンバのように値が順に送られ、受信側では、それを元にコピーのバリュー型インスタンスが作成される。そのため、コピーされたバリュー型インスタンスは送信側のバリュー型インスタンスとは独立した実体となります。

バリュー型は、任意のステート(データメンバ)を持つことができ、データメンバのアクセシビリティとして、privateとpublicを指定できます。このアクセシビリティのセマンティクスは、C++言語でのそれと同様です。すべてのバリュー型は、CORBA::ValueBase バリュー型を暗黙的に継承し、この CORBA::ValueBase バリュー型は、CORBA::Object がインタフェースについて果たす役割と同様の役割をバリュー型について果たします。

IDL コンパイラは、IDL で定義した valuetype と同名のクラスと state メンバのアクセス関数を実装をもつクラス(OBV_<valuetype 名>クラス)を提供します。valuetype の実装クラスは、この OBV_<valuetype 名>クラスと CORBA::DefaultValueRefCountBase クラスを継承して作成します。IDL の valuetype 内に定義した operation と attribute は、この valuetype が継承している valuetype やサポートしているインタフェースで定義されたのものも含めて、実装する必要があります。

また、IDL で定義した operation と attribute 以外にも、自身のコピーインスタンスを生成する_copy_value メソッドを実装する必要があります。IDL の valuetype 定義に custom を指定した場合、ORBA::CustomMarshal クラスを継承しますので、実装クラス内に marshal および unmarshal メソッドも実装する必要があります。

valuetypeを受信する場合、CORBA::ValueFactoryBaseを継承したValueFactory実装クラスを実装する必要があります。このクラスでは、valuetype の実装クラスを生成する create_for_unmarshal メソッドを実装する必要があります。IDL で factory 宣言により独自の実装クラス生成メソッドを持つファクトリクラスを定義できます。クラス名は、<valuetype 名>_<factory 名>で、メソッド名は factory 名と同じです。メソッドの引数は IDL operation のパラメータ(in 引数のみ)と同じ方法で展開されます。valuetype の実装者は、このファクトリクラスを継承した実装クラスを作成する必要があります。実装した ValueFactory は、valuetype 受信前に、CORBA::ORB::register_value_factory 関数を使用して、ORB に登録しておく必要があります。

```
valuetype の例
// IDL
valuetype Val {
        public long state1;
        private long state2;
        factory val_init(in long state1,in long state2);
        attribute long my_data;
        void my_ope();
};

// C++

// IDL コンパイラ生成クラス

class Val : public virtual CORBA::ValueBase {
protected:
        Val();
        virtual ~Val();

        virtual void state2(CORBA::Long state2) = 0;
        virtual CORBA::Long state2() const = 0;

public:
        virtual     CORBA::Long     lmy_data(CORBA::Environment&     _env     =
Ob_default_environment()) = 0;
```

```
        virtual void my_data(CORBA::Long local_data, CORBA::Environment& _env =
Ob_default_environment()) = 0;
        virtual void my_ope(CORBA::Environment& _env = Ob_default_environment()) =
0;

        virtual void state1(CORBA::Long state1) = 0;
        virtual CORBA::Long state1() const = 0;

        static Val* _downcast(CORBA::ValueBase*, CORBA::Environment& _env =
Ob_default_environment());
};

class OBV_Val : public virtual Val {
protected:
        OBV_Val();
        OBV_Val(CORBA::Long state1, CORBA::Long state2);
        virtual ~OBV_Val();

        virtual void state2(CORBA::Long state2);
        virtual CORBA::Long state2() const;

public:
        virtual void state1(CORBA::Long state1);
        virtual CORBA::Long state1();
};

// valuetype 実装クラス

class Val_i : public OBV_Val, public virtual CORBA::DefaultValueRefCountBase {
private:
        CORBA::Long m_my_data;
protected:
        Val_i() : OBV_Val(), m_my_data(0) {}
        Val_i(CORBA::Long a1, CORBA::Long a2, CORBA::Long a3) : OBV_Val(a1, a2),
m_my_data(a3) {}
        virtual ~Val_i() {}

public:
        virtual CORBA::Long my_data(CORBA::Environment&) { return m_my_data; }
        virtual void my_data(CORBA::Long _my_data, CORBA::Environment&) { m_my_date
= _my_data; }
        virtual void my_ope(CORBA::Environment&) { ... }
        virtual CORBA::ValueBase_ptr _copy_value(CORBA::Environment&)
        { return new Val_i(state1(), state2(), _my_data); }
...
};


class Val_init_i : public Val_init {
public:
        Val_init_i() {}
        virtual ~Val_init_i() {}

        virtual Val* val_init(CORBA::Long state1, CORBA::Long state2,
CORBA::Environment&)
        { return new Val_i(state1, state2, 0); }
        virtual CORBA::ValueBase_ptr create_for_unmarshal(CORBA::Environment&)
        { return new Val_i(); }
...
};
```

バリュー型には、インスタンス化可能なコンクリートバリュー型とステートを持たず、インスタンス化できない

アブストラクトバリュー型があります。バリュー型は、1 つのコンクリートバリュー型と複数のアブストラクトバリュー型を継承することができますが、アブストラクトバリュー型はコンクリートバリュー型を継承できません。またアブストラクトバリュー型の場合は、OBV_<valuetype 名>クラスは生成されません。

1 つの通常の(アブストラクトでない)インタフェースと複数のアブストラクトインタフェースをサポートすることができます。ここでのサポートとは、インタフェースで定義されているオペレーション(アトリビュートも含む)をバリュー型に取り込むことです。

```
concrete valuetype の例
//IDL
valuetype Val {
        public long state1;
        private long state2;
        factory val_init();
        factory val_init(in long state1,in long state2);
        void local_method(in long arg);
        attribute long local_data;
};

//C++
class Val : public virtual CORBA::ValueBase {
protected:
        Val();
        virtual ~Val();
        virtual void state2(CORBA::Long state2) = 0;
        virtual CORBA::Long state2() const = 0;
public:
        virtual void local_method(CORBA::Long arg, CORBA::Environment& _env =
Ob_default_environment()) = 0;
        virtual CORBA::Long local_data(CORBA::Environment& _env =
Ob_default_environment()) = 0;
        virtual void local_data(CORBA::Long local_data, CORBA::Environment& _env =
Ob_default_environment()) = 0;
        virtual void state1(CORBA::Long state1) = 0;
        virtual CORBA::Long state1() const = 0;
        ...
}
class Val_init : public virtual CORBA::ValueFactoryBase {
protected:
        Val_init();

public:
        virtual ~Val_init();
        virtual Val* val_init(CORBA::Environment& _env = Ob_default_environment())
= 0;
        virtual Val* val_init(CORBA::Long state1, CORBA::Long state2,
CORBA::Environment& _env = Ob_default_environment()) = 0;
        ...
};


abstract valuetype の例
// IDL
abstract valuetype AbsVal {
        void local_method(in long arg);
        attribute long local_data;
};
//C++
class AbsVal : public virtual CORBA::ValueBase {
protected:
        AbsVal();
        virtual ~AbsVal();
```

```
public:
        virtual void local_method(CORBA::Long arg, CORBA::Environment& _env =
Ob_default_environment()) = 0;
        virtual CORBA::Long local_data(CORBA::Environment& _env =
Ob_default_environment()) = 0;
        virtual void local_data(CORBA::Long local_data, CORBA::Environment& _env =
Ob_default_environment()) = 0;
        ...
}
```

バリュー型のオペレーションの実装は、CORBA 呼び出しの発生しないローカル関数になります。

バリュー型をローカル関数の引数に渡した場合は、値渡しになません(コピーされない)。したがって、ローカルスタブのオペレーションに渡された場合もコピーされません。

バリュー型は、null 値を送信することができます。

バリューボックス型

バリューボックス型は、単一のステート(データメンバ)を持つバリュー型ですが、以下の事が通常のバリュー型と異ります。

・オペレーションを持たない。

・他の型から派生しない、および、他の型から継承されない。

・インタフェースをサポートできない。

・アブストラクトにできない。

・ユーザによる実装が不要。(IDL コンパイラが生成)

・バリューファクトリの実装および登録が不要。

付録. WebOTX Object Broker C++/Java™ 機能差分

WebOTX Object Broker C++と同 Java では機能や使い方が異なる部分があります。Java で作成されたクライアントから C++で作成されたサーバを呼び出すなど、両者が混在した環境で使うプログラムを作成するときや、C++で書かれたプログラムを Java に移植するときなどは注意する必要があります。

以下に両者の違いを示します。'-'は CORBA 仕様に定義されていないことを表します。

項目	Java™	C++
IDL コンパイラ	i2j(i2j.exe)	i2cc(i2cc.exe)
タイムアウト既定値	無限	30 秒
サーバタイムアウト既定値	なし	30 秒
module マッピング	package	namespace
interface マッピング	interface	class
out, inout 引数	Holder クラス	&または*&(型による)
Helper クラス	○	-
signed/unsigned の区別	なし	あり
OAD が動作するホストの指定	ORBInitialHost(*)	なし
OAD のポート番号指定	ORBInitialPort(*)	OadPort
名前サーバが動作するホストの指定	NameServiceHost(*)	NameServiceHostName
インタフェースリポジトリが動作するホストの指定	InterfaceRepositoryHost(*)	InterfaceRepositoryHostName
ログ出力レベルの指定	LogLevel(*)	LoggingLevel
環境設定ファイルの指定	PropertyFile(*)	環境変数 ORBCONFIG

(*)は Java で書かれたアプリケーション(もしくはアプレット)に対して設定を行う場合のものです。Java 版でも、OAD や名前サーバなど、C++で書かれたものは C++の設定に従います。