



Net Express  
分散コンピューティング

Micro Focus NetExpress™

# 分散コンピューティング

Micro Focus®

第 3 版

1998 年 10 月

Copyright © 1999 Micro Focus Limited. All rights reserved.

本書、ならびに使用されている固有の商標と商品名は、国際法で保護されています。

Micro Focus は、このマニュアルの内容が公正かつ正確であるよう万全を期しておりますが、このマニュアルの内容は予告なしに随時変更されることがあります。

このマニュアルに述べられているソフトウェアはライセンスに基づいて提供され、その使用および複製は、ライセンス契約に基づいてのみ許可されます。特に、Micro Focus 社製品のいかなる用途への適合性も明示的に本契約から除外されており、Micro Focus はいかなる必然的損害に対しても一切責任を負いません。

Micro Focus® は登録商標です。Form Designer™、Micro Focus COBOL™、および NetExpress™ は Micro Focus Limited の商標です。

Microsoft®、Windows® および Windows for Workgroups® は Microsoft Corporation の登録商標です。

Visual Basic および Windows NT は、Microsoft Corporation の商標です。

IBM® は、International Business Machines Corporation の登録商標です。

UNIX® は X/Open Company Limited の登録商標です。

Copyright© 1987-1999 Micro Focus  
All Rights Reserved.

# 序文

このマニュアルでは、NetExpress で使用可能な分散オブジェクト用のサポートについて説明します。また、Orbix (IONA 社のオブジェクト リクエスト ブローカ)、OLE オートメーション、および Microsoft の Transaction Server に対するサポートの使用方法についても説明します。

## 対象読者

このマニュアルは、IONA 社の Orbix、OLE オートメーション、または Microsoft の Transaction Server を使用して、分散オブジェクトにアクセスする COBOL プログラムを作成する方を対象としています。

## このマニュアルの使用法

第 1 部では、分散オブジェクトにアクセスするための方法を概説します。

第 2 部では、NetExpress で使用できる Orbix (IONA 社のオブジェクト リクエスト ブローカ) サポートの使用法について説明します。

第 3 部では、OLE オートメーションと Microsoft Transaction Server に対するサポートを含め、Object COBOL モデルについて説明します。

## 表記規則

このユーザーガイドでは、次の書体や規則を使用します。

- 入力するテキストは、次のように表示します。

```
cat script_name | more
```

斜体テキストはコマンドの一部として入力する変数を表します。

- コマンド行やコード例でオプション入力するテキストは、角かっこ ([[]]) で囲みます。次の式では、オプションの単語 NOT を入力しない場合、*column\_name* は *pattern\_value* に設定されます。一方、NOT を入力した場合、*column\_name* は *pattern\_value* 以外に設定されます。

```
column_name [NOT] LIKE pattern_value
```

- 特定のモデルやオペレーティング システムだけに適用する項や段落については、段落のすぐ前に太字斜体の項目名が記載されています。例えば、次のようになります。

**UNIX**

この段落は UNIX システムだけに適用されます。

# 目次

|   |     |
|---|-----|
| 序文.....   | ii  |
| 対象読者.....                                       | ii  |
| このマニュアルの使用方法.....                               | ii  |
| 表記規則.....                                       | ii  |
| 第1章 はじめに.....                                   | 1-1 |
| 1.1 CORBA.....                                  | 1-1 |
| 1.1.1 Object Management Architecture (OMA)..... | 1-2 |
| 1.1.1.1 オブジェクト リクエスト ブローカ (ORB).....            | 1-2 |
| 1.1.1.2 オブジェクト サービス.....                        | 1-2 |
| 1.1.1.3 ドメイン インターフェイス.....                      | 1-3 |
| 1.2 Object Management Group (OMG).....          | 1-4 |
| 1.2.1 OMG の歴史.....                              | 1-4 |
| 1.2.2 OMG の目標.....                              | 1-4 |
| 1.3 CORBA インターフェイス定義言語 (IDL).....               | 1-4 |
| 1.3.1 IDL の定義.....                              | 1-5 |
| 1.3.2 IDL の作成方法.....                            | 1-5 |
| 1.3.3 IDL の形式.....                              | 1-5 |
| 1.3.4 IDL の使用法は.....                            | 1-6 |
| 1.4 CORBA と COBOL.....                          | 1-7 |
| 1.4.1 Object COBOL に最適な CORBA.....              | 1-7 |
| 1.4.2 NetExpress がサポートする Micro Focus CORBA..... | 1-8 |
| 第2章 NetExpress の Orbix サポート.....                | 2-1 |

|  |      |
|--|------|
| 2.1 Orbix の定義 .....                            | 2-2  |
| 2.2 Orbix 分散透過性.....                           | 2-2  |
| 2.3 Orbix のロケーション透過性.....                      | 2-3  |
| 2.3.1 ORBIX ネーミング サービス.....                    | 2-4  |
| 2.4 CORBA インターフェイス定義言語.....                    | 2-4  |
| 2.4.1 IDL の定義 .....                            | 2-4  |
| 2.4.2 IDL の使用方法 .....                          | 2-6  |
| 2.5 Orbix を使用した COBOL 分散アプリケーションの開発.....       | 2-6  |
| 2.5.1 NetExpress の Orbix 用 CORBA ウィザードの概要..... | 2-7  |
| 2.5.2 NetExpress Orbix サポート ツールの概要.....        | 2-7  |
| 2.5.3 CorbaGen コマンド行ユーティリティの概要.....            | 2-7  |
| 2.5.4 レガシー アプリケーションの基本的な移行手順.....              | 2-8  |
| 2.5.5 ORBIX アプリケーションの基本的な新規開発手順 .....          | 2-10 |
| 2.6 CORBA ウィザード for Orbix .....                | 2-10 |
| 2.6.1 新規プロジェクトの作成.....                         | 2-11 |
| 2.6.2 Orbix サーバー用ファイルの選択.....                  | 2-12 |
| 2.6.3 クライアントとサーバーのオプション指定.....                 | 2-13 |
| 2.6.4 Orbix クライアント用ファイルの選択.....                | 2-14 |
| 2.6.5 生成.....                                  | 2-15 |
| 2.6.6 最終ページ.....                               | 2-16 |
| 2.6.7 実行形式ファイルのビルド .....                       | 2-17 |
| 2.7 Orbix サポート ユーティリティ .....                   | 2-17 |
| 2.8 CorbaGen コマンド行ユーティリティ .....                | 2-17 |
| 2.8.1 CorbaGen 構文.....                         | 2-18 |
| 2.9 ORBIX クライアントと ORBIX サーバーのデバッグとアニメート .....  | 2-19 |

|   |      |
|---|------|
| 2.9.1 テスト方法.....                                      | 2-19 |
| 2.9.2 コードのアニメート方法.....                                | 2-20 |
| 2.9.2.1 アニメーションの設定のために corbagen を使用する.....            | 2-20 |
| 2.9.2.2 アニメーションの設定のために ORBIX 用の CORBA ウィザードを使用する..... | 2-21 |
| 2.10 ORBIX サポート制限事項.....                              | 2-21 |
| 第3章 他のオブジェクト ドメインの使用.....                             | 3-1  |
| 3.1 概要.....   | 3-1  |
| 3.2 汎用のドメイン サポート.....                                 | 3-1  |
| 3.2.1 別のドメインからのオブジェクト識別.....                          | 3-2  |
| 3.2.2 別のドメインのオブジェクトに対するメッセージ送信.....                   | 3-2  |
| 第4章 OLE オートメーションと DCOM.....                           | 4-1  |
| 4.1 概要.....   | 4-1  |
| 4.1.1 チェックリスト - プロジェクトを始める前に.....                     | 4-2  |
| 4.2 ActiveX クライアントの作成.....                            | 4-2  |
| 4.2.1 プログラムの ActiveX クライアント化.....                     | 4-3  |
| 4.2.2 ActiveX プロキシ オブジェクトの作成.....                     | 4-4  |
| 4.2.3 ActiveX オブジェクトへのメッセージ送信.....                    | 4-6  |
| 4.2.4 タイプ ライブラリ情報の使用.....                             | 4-8  |
| 4.2.5 ActiveX プロキシ オブジェクトの終了.....                     | 4-9  |
| 4.2.6 OLE オートメーションの例外.....                            | 4-9  |
| 4.3 Object COBOL による ActiveX オブジェクトの作成.....           | 4-13 |
| 4.3.1 OLE オートメーションのクラスの作成.....                        | 4-16 |
| 4.3.2 OLE オートメーション サーバーの登録.....                       | 4-20 |
| 4.3.3 終了オプションの設定.....                                 | 4-26 |
| 4.3.4 ActiveX オブジェクトのデバッグ.....                        | 4-26 |

|         |                             |      |
|---------|-----------------------------|------|
| 4.3.5   | タイプ ライブラリの操作 .....          | 4-28 |
| 4.3.5.1 | プロパティとメソッドの追加 .....         | 4-28 |
| 4.3.5.2 | プロパティとメソッドの削除 .....         | 4-28 |
| 4.3.5.3 | パラメータのデータ型の変更 .....         | 4-29 |
| 4.3.6   | スレッド オプションの設定 .....         | 4-30 |
| 4.3.6.1 | アウトオブプロセス サーバー .....        | 4-31 |
| 4.3.6.2 | インプロセス サーバー .....           | 4-31 |
| 4.4     | DCOM の使用 .....              | 4-32 |
| 4.4.1   | DCOM とセキュリティ .....          | 4-33 |
| 4.5     | レジストリ エントリの編集 .....         | 4-34 |
| 4.6     | 詳細情報 .....                  | 4-35 |
| 第5章     | OLE データ型 .....              | 5-1  |
| 5.1     | 概要 .....                    | 5-1  |
| 5.2     | OLE データ型指定規則 .....          | 5-2  |
| 5.3     | オブジェクト リファレンス .....         | 5-4  |
| 5.4     | Variant 型 .....             | 5-6  |
| 5.4.1   | OleVariant クラスを使用する前に ..... | 5-8  |
| 5.4.2   | OLEVariant インスタンスの作成 .....  | 5-8  |
| 5.4.3   | OLEVariant からのデータ読み取り ..... | 5-11 |
| 5.5     | SafeArray 型 .....           | 5-13 |
| 5.5.1   | SafeArray を使用する前に .....     | 5-14 |
| 5.5.2   | SafeArray の作成 .....         | 5-14 |
| 5.5.3   | SafeArray に関する情報の取得 .....   | 5-17 |
| 5.5.4   | SafeArray 要素の読み書き .....     | 5-19 |
| 5.5.5   | SafeArray データへの直接アクセス ..... | 5-21 |

|   |     |
|---|-----|
| 第6章 Microsoft Transaction Server とのインタフェース..... | 6-1 |
| 6.1 Transaction Server コンポーネントの作成.....          | 6-1 |
| 6.1.1 Transaction Server コンポーネントの構造.....        | 6-1 |
| 6.1.2 objectcontext クラス.....                    | 6-3 |
| 6.1.3 コンテキスト オブジェクト.....                        | 6-4 |
| 6.1.4 Transaction Server コンポーネントの終了.....        | 6-4 |
| 6.1.5 Objectcontext のメソッド.....                  | 6-4 |
| 6.1.5.1 SetComplete.....                        | 6-4 |
| 6.1.5.2 SetAbort.....                           | 6-5 |
| 6.1.5.3 CreateInstance.....                     | 6-5 |
| 6.1.5.4 DisableCommit.....                      | 6-5 |
| 6.1.5.5 EnableCommit.....                       | 6-5 |
| 6.1.5.6 IsInTransaction.....                    | 6-6 |
| 6.1.5.7 IsCallerInRole.....                     | 6-6 |
| 6.1.5.8 IsSecurityEnabled.....                  | 6-6 |
| 6.1.6 Transaction Server コンポーネントのビルド.....       | 6-6 |
| 6.2 Transaction Server コンポーネントの実行とデバッグ.....     | 6-7 |
| 6.3 完全な Transaction Server コンポーネントの例.....       | 6-7 |



# 第1章 はじめに

このマニュアルでは、COBOL プログラムを使用して分散コンポーネントにアクセスする方法について説明します。

分散コンポーネントのアクセス方法には 2 通りあります。

- Object COBOL 構文を使用する。

この方法については、「OLE オートメーション」の章を参照してください。

- 標準 COBOL 構文用の CORBA を使用する。

詳細については、「*NetExpress の Orbix サポート*」の章を参照してください。

## 1.1 CORBA

異種プラットフォーム間に適用する分散アプリケーションを開発するのは、複雑な作業です。この場合、次の点を考慮する必要があります。

- どの通信プロトコルを使用するか。
- 異種プラットフォーム間でデータ型の表現が異なる場合、どのように対応するか (たとえば、ASCII と EBCDIC)。
- 常に進化する環境で、各ホストに一貫した構成ファイルがない場合に、クライアントはどのようにサーバーを検出するか。
- セキュリティをどのように管理するか。

Object Management Group (OMG) は、このような問題を解決するために Common Object Request Broker Architecture (CORBA) を定義しました。CORBA は、Object Management Architecture (OMA) という包括的なアーキテクチャの一部になります。

CORBA では、オブジェクト リクエスト ブローカ、または ORB という用語がよく使用されます。これらの用語の関係をまとめると、CORBA は OMG が定義した規格を指し、ORB は CORBA 規格の個別の実装内容を指します。

CORBA の目標は、前述した分散アプリケーションの処理を自動化することです。これにより、プログラマは分散ビジネス ロジックの開発に集中できるようになります。

CORBA 規格に準拠した製品の具体例は、次のとおりです。

- Iona 社の ORBIX
- Borland 社の VisiBroker

- BEA 社の M3
- IBM 社の Component Broker

CORBA 規格では、ミドルウェアのオープン規格が定義されています。具体的には、オブジェクト リクエスト ブローカ (ORB) を使用して、クライアント サーバー関係を確立する方法が指定されています。CORBA では、分散アプリケーション同士が、場所、設計者、作成時の言語に関係なく (それぞれを「ロケーション透過性」、「コンポーネント指向」、「言語透過性」という) 互いに通信できます。

### 1.1.1 Object Management Architecture (OMA)

CORBA は、Object Management Group (OMG) が定義する Object Management Architecture (OMA) の中核です。

Object Management Architecture を完全に実現するには、CORBA を中心とし、その他いくつかのコンポーネントを使用します。これらのコンポーネントを CORBA と使用すると、分散アプリケーションの開発と配置の問題を完全に解決できます。OMA アーキテクチャで最も重要なコンポーネントは、次の 3 つです。

- ORB  
オブジェクト リクエスト ブローカ
- オブジェクト サービス  
多くのアプリケーションに共通のサービス
- ドメイン インターフェイス  
標準的な業界特有のソリューション

#### 1.1.1.1 オブジェクト リクエスト ブローカ (ORB)

オブジェクト リクエスト ブローカ (ORB) は、CORBA 標準の通信で中心的な役割を果たします。ORB は、コンポーネントの実装に使用した個別のプラットフォームや技術とは関係なく、コンポーネント間で対話するための基盤を提供します (このような分散型のコンポーネントは COBOL を含めて、どの言語でも作成することができます)。

ORB 標準に準拠した分散システムでは、システム全体での汎用性と接続性が保証されます。

一般的には、標準的な CORBA Interface Definition Language (IDL) を使用して、分散アプリケーションのさまざまなコンポーネントに対するインターフェイスを定義します。その後で、個別の IDL 言語マッピングを使用し、ORB にすべてまとめてリンクさせてコンポーネント間の対話を可能にするスタブを作成します。

#### 1.1.1.2 オブジェクト サービス

オブジェクト サービスは、OMA 全体でも ORB の次に重要な要素です。オブジェクト サービスの内容は、次のとおりです。

- トランザクション

分散アプリケーションの要素間でトランザクションの同期をとります。

- セキュリティ

情報に対する権限のないアクセス、権限のない操作妨害などから分散アプリケーションを保護します。

- ネーミング

特定のコンポーネントは、それぞれが使用可能かどうかを登録します。その結果、コンポーネント間で特定のインターフェイスを検索することにより、実行時に他のコンポーネント（たとえば、コンポーネントであるプリンタ）を動的に検出することができます。

- トレーディング

クライアントは、特定の種類のサービスを検索することができます。また、検索条件を追加することもできます（たとえば、「2 階にあるすべてのカラー プリンタコンポーネントを検出する」という検索条件を指定することが可能です）。

- 保存

コンポーネントの状態を保存します（現在の記憶域を保存して、後で再読み込みし、終了した箇所から続行することができます）。

ユーザーが作成したアプリケーションは、IDL で作成した標準的なインターフェイスにより、これらのサービスを使用します。これらのサービスは、ビルド済みの標準的な CORBA アプリケーションとして表示することができます。この CORBA アプリケーションには、ユーザー作成のアプリケーションで使用できる機能が含まれています。たとえば、コンポーネントであるサーバーは、トレーディング サービスとの標準的なインターフェイスを使用して現在の場所を登録することができます。そのコンポーネントを検索し使用しようとするクライアントは、検索手段としてトレーディング サービスを使用します。クライアントは実行時に他のコンポーネントを動的に検出することができるので、特定のホスト名を記述した静的な構成ファイルを維持する必要がありません。これは、豊富な標準オブジェクト サービスが提供する大きな付加価値のほんの一例に過ぎません。

### 1.1.1.3 ドメイン インターフェイス

OMA アーキテクチャでもう一つ重要なコンポーネントは、ドメイン インターフェイスです。ドメイン インターフェイスは、特定のアプリケーション ドメインのエンドユーザーと直接関係のある機能を提供する垂直領域を表します。

これらは、特定の業界を対象とした標準インターフェイスです。たとえば、ユーザーが一般的な元帳機能を作成する場合は、ユーザーが独自にインターフェイスを定義する代わりに、CORBA の汎用元帳機能への定義済みの標準インターフェイスを使用することができます。この標準インターフェイスは、同じインターフェイスを使用する他のアプリケーションでも使用できます。

OMG 標準インターフェイスを使用すると、個別の要件を満たす最高の環境を整えることができ、将来的に投資がむだになることはありません。

すべての条件を満たす標準インターフェイスはありませんが、OMG が定義した業界標準準拠のコンポーネントを利用すれば、ユーザーの労力を減らすことができます。

## 1.2 Object Management Group (OMG)

### 1.2.1 OMG の歴史

OMG は、1989 年 5 月に 3Com Corporation、American Airlines、Canon Inc.、Data General、Hewlett-Packard、Philips Telecommunications N.V.、Sun Microsystems、Unisys Corporation の 8 社により設立されました。また、1989 年 10 月に OMG は非営利法人として独立運営を開始しました。OMG では、「卓越した技術を使用し、商業的にも成功する、ベンダに依存しない仕様をソフトウェア業界に向けて開発する」という理念を掲げています。この理念により、OMG は、現在 900 のメンバーが属す世界最大のソフトウェア開発組織となっています。OMG の本部は、米国マサチューセッツ州フラミンガムにあります。また、その他の提携先は、英国、ドイツ、日本、オーストラリアにあります。

### 1.2.2 OMG の目標

OMG は、標準化されたオブジェクト ソフトウェアの導入を促進し、コンポーネントベースのソフトウェア市場を開拓するために設立されました。組織の方針は、業界のガイドラインとオブジェクト管理仕様を確立し、アプリケーション開発のための共通のフレームワークを提供することです。これらの仕様に準拠することにより、すべての主要なハードウェア プラットフォームやオペレーティング システムを通じて異種コンピューティング環境を開発することができるようになります。

OMG は、オブジェクト管理を、「オブジェクト」の作成により現実の世界をモデリングするソフトウェア開発として定義します。これらのオブジェクトは、ソフトウェアを識別できるプログラムのコンポーネントの属性、関係、およびメソッドをまとめたものです。オブジェクト指向システムの重要な利点は、既存のコンポーネントを拡張し、システムに新しいコンポーネントを追加することにより、機能を拡張できることです。オブジェクト管理を行うと、アプリケーション開発の高速化、メンテナンスの簡易化、ソフトウェアの再利用が可能になります。

オブジェクト指向のアーキテクチャは、広く一般的に受け入れられ、使用されるようになりつつあります。実質的には、世界中で、コンピュータ システムの主要なプロバイダやユーザーはすべてオブジェクト指向のツールやアプリケーションを使用しているか、または実装を計画しています。5 年以内には、オブジェクト指向のソフトウェアの売上は 30 億ドルを超えるものと予測されています。

## 1.3 CORBA インターフェイス定義言語 (IDL)

CORBA インターフェイス定義言語 (IDL) は、OMG が定義した OMA アーキテクチャの重要なコンポーネントです。言語に依存せず、透過的にすべてを機能させることができます。

### 1.3.1 IDL の定義

CORBA アプリケーションは、通常、いくつかの対話型コンポーネントで構成されています。クライアントとして動作するコンポーネントは、1 つまたは複数のサーバーによるサービスを利用します。また、ある対話でサーバーとして動作するコンポーネントが、他の対話でクライアントとして動作することもできます。

コンポーネントは、正しく定義されたインターフェイスだけを通して対話することができます。CORBA インターフェイス定義言語 (IDL) は、CORBA インターフェイスを指定するために OMG が定義した、言語に依存しない機構です。

IDL は、基本的にインターフェイスを抽象的に定義します。一度インターフェイスを定義すると、この IDL 定義を使用して、すべてを結合するコードを生成することができます。

### 1.3.2 IDL の作成方法

通常、IDL 定義は、標準的なテキスト エディタを使用して、普通のテキスト ファイルで作成します。規約により、IDL を含むファイルには、.idl という拡張子が付けられます。

COBOL の場合、IDL 定義の作成は自動化されています。適切なツールで通常の COBOL プログラムを処理すると、抽象的な IDL 構文を使用して、インターフェイスを記述する IDL ファイルを生成することができます。

### 1.3.3 IDL の形式

各 IDL ファイルには、インターフェイス定義を 1 つ以上含めることができます。各インターフェイス定義では、起動できる操作とそれらの操作に対するパラメータをインターフェイスとして定義します。次に、例を挙げます。

```
//  
  
//   Sample IDL  
  
//  
  
interface sample  
  
{  
  
    struct parm_struct {  
  
        string<50> message;  
  
        float      calc_result;  
  
        long       code;  
  
    };
```

```

exception FaultDetected {string reason; };

unsigned long first_operation(
                                in string<12>      arg1,
                                out parm_struct     arg2,
                                out string<80>     arg3)
                                raises (FaultDetected) ;

unsigned long next_operation(
    in  short      arg1,
    inout parm_struct arg2
);
};

```

上記の例では、「サンプル」というインターフェイスを定義しています。このインターフェイスには、次の定義が記述されています。

- 下記の 2 つの操作でパラメータとして使用される "parm\_struct" という構造体を定義する型定義
- ユーザーが指定した文字列を含む、"FaultDetected" という例外定義
- "first\_operation" と "next\_operation" の 2 つの操作

### 1.3.4 IDL の使用方法は

プログラム A がプログラム B を呼び出す場合を例として、IDL の使用方法を説明します。分散 CORBA 環境でプログラム A がプログラム B を呼び出すように指定するには、次の手順にしたがいます。

- IDL を使用して、プログラム B に対するインターフェイスの抽象定義を作成します。
- CORBA に実装されている IDL コンパイラで IDL を処理し、クライアント スタブとサーバー スタブを作成します。

次の図は、コンパイルと相互リンクが完了した場合の形式を示します。

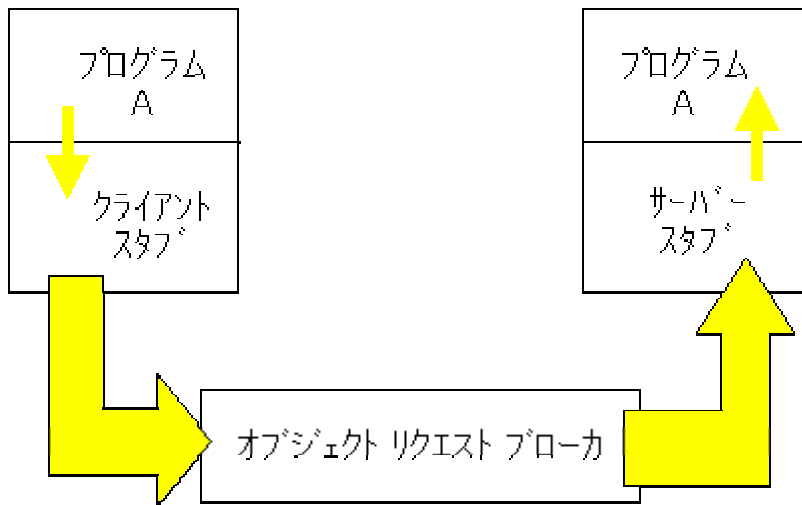


図1-1: IDL

プログラム A は、プログラム B を呼び出すように見えますが、実際に呼び出しているのは、プログラム B に対するインターフェイスのクライアントスタブです。

クライアントスタブは、呼び出しをパッケージ化し、ORB に渡して、サーバーに送ります。

サーバースタブは、呼び出しを受け取り、そのパッケージを解除して、プログラム B に渡します。

プログラム A とプログラム B は、同じ言語で作成する必要はありません。また、プログラムを作成した言語を互いに認識する必要もありません。プログラム A が、JAVA クライアントで、プログラム B が COBOL サーバーでもかまいません。この場合、JAVA クライアントスタブと COBOL サーバースタブを生成するには、IDL コンパイラを使用します。

## 1.4 CORBA と COBOL

### 1.4.1 Object COBOL に最適な CORBA

CORBA で使用する "Object Management Group"、"オブジェクトリクエストブローカ"、"Object Management Architecture" などの用語は、CORBA が分散オブジェクトでしか使用できないような印象を与えるかもしれません。

たしかに、CORBA は、分散コンポーネントで使用します。ただし、CORBA のコンテキストで、Smalltalk や Java のような純粋な OO 言語、C++ のような擬似 OO 言語、C や COBOL のような通常の手続き言語などを使用して、CORBA の分散コンポーネントを作成することができます。実際に、CORBA 仕様で最初に定義された言語バインディングは、C++ ではなく、C です。

CORBA で「オブジェクト」という用語を使用する場合、CORBA がオブジェクトモデルを使用して、分散コンポーネントの設計、開発、配置に関する問題を解決することを意味します。各分散コンポーネントは、クライアントに対する一連の操作をサポートするインターフェイスを使用します。そのため、この概念では、個別の実装内容（コンポーネントが実際に作成された言語など）は影響しません。

## 1.4.2 NetExpress がサポートする Micro Focus CORBA

NetExpress でサポートされている CORBA は、Iona 社の Orbix です。Orbix に対するサポート内容は、次のとおりです。

- Orbix 用の CORBA ウィザード。これは、NetExpress Orbix の新規プロジェクトを作成したり、既存の NetExpress プロジェクトを NetExpress Orbix プロジェクトに変換する場合に、使用することができます。ウィザードを使用すると、COBOL Orbix サーバーを作成することができます。また、オプションで COBOL Orbix クライアントを作成することもできます。

コマンド行ユーティリティ CorbaGen。これを NetExpress のコマンド行で使用すると、Orbix COBOL アプリケーションを開発することができます。



## 第2章 NetExpress の Orbix サポート

NetExpress の Orbix サポートを使用すると、COBOL に Iona 社のオブジェクト リクエスト ブローカ (ORB) 技術 (Orbix) を適用することができます。Orbix と Orbix サポートを併用すると、次のような作業が可能になります。

- 広範囲な異種プラットフォームで既存の (レガシー) COBOL アプリケーションを分散させる。
- COBOL で作成された新しいビジネス ロジックのコンポーネントを Orbix 分散環境で開発し、配置する。

---

注記: Orbix は、NetExpress に含まれているわけではありません。NetExpress の Orbix サポートを使用するには、あらかじめ Iona 社の Orbix を購入する必要があります。Iona 社の連絡先については、後述するホームページを参照してください。

---

NetExpress の Orbix サポートには、次の機能が含まれます。

- Orbix 用の CORBA ウィザード。これは、NetExpress IDE で起動することができます。
- CorbaGen ユーティリティ。これは、NetExpress の DOS プロンプトから起動します。COBOL 開発者は、上記の Orbix サポートをすべて使用できます。

Orbix 用 CORBA ウィザードと CorbaGen ユーティリティは、どちらも標準的な COBOL アプリケーションの分散化を次のようにサポートします。

- 言語に依存しない分散 Orbix コンポーネントを生成するために COBOL コードをカプセル化するコードを自動生成します。この生成コードは、ラッパーと呼ばれます。

ラッパーは、Orbix 分散の問題を実際のビジネス ロジックとは切り離して処理する場合に使用します。Orbix 分散の問題とは、Orbix の初期化や、Orbix ORB の言語に依存しない分散環境で COBOL 固有のデータ型を処理する方法などを指します。

- COBOL で作成された分散 Orbix コンポーネント用のインターフェイス記述を自動生成します。これらの記述は、言語に依存しない標準的な CORBA インターフェイス定義言語 (IDL) を使用して作成されます。

Orbix IDL コンパイラは、IDL を使用して、Orbix にコンポーネントを接続するスタブを生成します。また、IDL を使用すると、ORB でサポートされている他の言語で作成したコンポーネントを有効化し、COBOL で作成した Orbix コンポーネントとの透過的なインターフェイスを使用できます。

- 完全な分散 Orbix コンポーネントをビルドするためのサポートを自動生成します。
  - Orbix 用 CORBA ウィザードでは、分散 Orbix コンポーネントをビルドするために、NetExpress の

新規プロジェクトを NetExpress Orbix プロジェクトに作成し (または、現在のプロジェクトを交換し)、正しいオプションを設定することができます。

- コマンド行ツールでは、分散 Orbix コンポーネントをビルドするために、自動的に makefile を生成します。

## 2.1 Orbix の定義

Iona 社の Orbix は、Object Management Group (OMG) の Common Object Request Broker Architecture (CORBA) によって定義されている仕様に準拠したオブジェクト リクエスト ブローカ (ORB) です。Orbix は、OrbixTransactions や OrbixSecurity などの補足的なサービスを使用して拡張することもできます。

Orbix は、現在、市場で最先端を行く ORB で、多くの主要企業が使用しています。Orbix のサポート内容は、次のとおりです。

- アプリケーションコンポーネント間での、複数言語のバインディングと完全な言語透過性
- 完全なマルチスレッド
- さまざまなオブジェクト サービス
- ホストの異種プラットフォーム

Orbix の詳細については、次の場所の Iona Orbix のホームページを参照してください。

<http://www.ionasoft.com>

<http://www.ot.tis.co.jp>

## 2.2 Orbix 分散透過性

CORBA をベースとする製品の主な機能は、分散透過性です。これは、全く異なる環境に常駐するリモート プログラムをローカル プログラムのように起動できることを意味します。通信コードの作成やトランスポート プロトコルの処理は必要ありません。また、異種ハードウェア環境で基本的なデータ型表現が異なる場合を想定する必要もありません。これらの問題は、オブジェクト リクエスト ブローカがすべて自動的に処理します。

分散環境以外でプログラム A がプログラム B を呼び出す場合に、CORBA 分散アプリケーションを作成するには、次の作業が必要です。

- プログラム B へのインターフェイスを定義します。

すべての異なるエントリ ポイントを操作としてプログラム B に記述し、それぞれに個別のパラメータ セットを与えます。この操作とパラメータの集合は、インターフェイス定義と呼ばれ、個別にインターフェイス名が指定されます。インターフェイス定義は、言語に依存しない標準 CORBA IDL (インターフェイス定

義言語) で別のファイル (通常は .idl という拡張子が付いたファイル) に作成されます。

- CORBA IDL コンパイラを使用して、インターフェイス定義を含むファイルを処理します。必要な特定の言語バインディングを指定します。その結果、次のスタブが生成されます。
  - インターフェイスを起動するクライアント アプリケーションをコンパイルし、リンクさせるクライアント スタブ
  - リモート サーバー アプリケーションにより起動されるアプリケーションをコンパイルし、リンクするサーバー スタブ。

分散型モデルでは、プログラム A は、クライアント スタブ コードを呼び出します。スタブは、メッセージをパッケージ化し、ORB に渡します。ORB は、リモート アプリケーションを検索し、サーバー スタブにメッセージをルーティングします。サーバー スタブがメッセージを受け取ると、パッケージ化されていたメッセージはローカル環境に適した形式で解除され、ローカル アプリケーションから送信されたメッセージと同様にアプリケーションに渡されます。次の図は、この流れを示したものです。

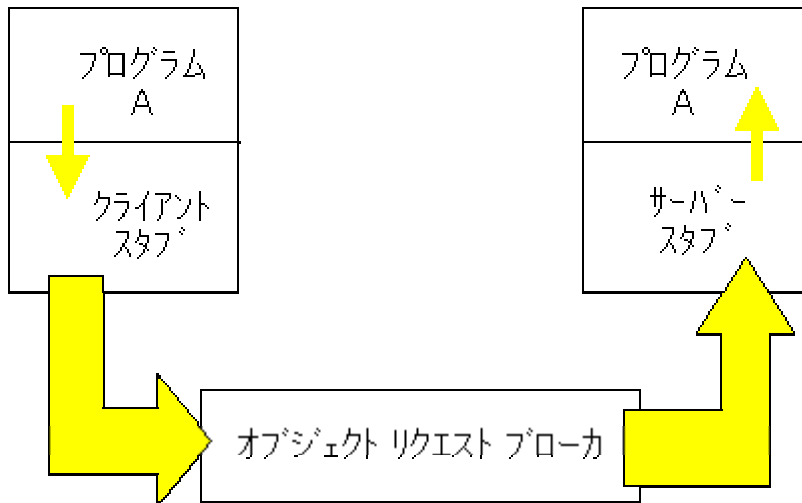


図 2-1 Orbix 分散型透過性

## 2.3 Orbix のロケーション透過性

オブジェクト リクエスト ブローカは、基本的には、メッセージをビルドし、クライアントから分散コンポーネントにルーティングするための機構です。クライアントのプログラマは、インターフェイスを使用して接続するコンポーネントを識別するための固有のオブジェクト リファレンスを作成します。ORB は、このオブジェクト リファレンスを使用して、メッセージを必要なコンポーネントにルーティングします。クライアントのプログラマは、接続の確立方法、使用するプロトコル、またはクライアントの環境とコンポーネントの環境の違いを気にする必要ありません。これらの問題は、ORB がすべて処理します。

また、クライアントのプログラマは、インターフェイスを使用して接続しているコンポーネントの実際の場所を気に

する必要もありません。オブジェクト リファレンスを指定したコンポーネントは、同じネットワーク ノードの別のプロセスである場合も、地球の裏側の別のノードで動作するコンポーネントである場合もあります。

単純な場合は、クライアントをサーバーに結合することができます。場所に対する完全な透過性が必要な場合は、ORBIX ネーミング サービスを通して、コンポーネントに対する固有のオブジェクト リファレンスを取得することができます。

### 2.3.1 ORBIX ネーミング サービス

大きな分散環境（常に変化する）でクライアントから操作を行うには、実行時にコンポーネントを動的に検索する必要があります。そのためには、ORBIX ネーミング サービスを使用します。

コンポーネントがアクティブになると、ネーミング サービス インターフェイスを使用して、コンポーネントを自動的に登録することができます。コンポーネントと通信するクライアントは、ネーミング サービスで登録されたコンポーネントのオブジェクト リファレンスについて、ネーミング サービスに問い合わせることができます。

ネーミング サービスの詳細については、該当する ORBIX マニュアルを参照してください。

このアーキテクチャの大きな利点は、クライアントがコンポーネントを実行時に動的に検索できることです。ハードコーディングされた構成ファイルは必要ありません。コンポーネントは、別の場所、別のハードウェア環境、別のソフトウェア環境に移動することが多いため、コンポーネントはクライアントにより自動的に検索できるようになっています。

## 2.4 CORBA インターフェイス定義言語

### 2.4.1 IDL の定義

代表的な ORBIX アプリケーションは、いくつかの対話型のコンポーネントから構成されています。クライアントとして動作するコンポーネントは、1 つまたは複数のサーバーによるサービスを利用します。また、ある対話でサーバーとして動作するコンポーネントが、他の対話でクライアントとして動作することもできます。

コンポーネントは、正しく定義されたインターフェイスだけを通して対話することができます。CORBA インターフェイス定義言語 (IDL) は、CORBA インターフェイスを指定するために OMG が定義した、言語に依存しない機構です。

通常、IDL 定義は、標準的なテキスト エディタを使用して、普通のテキスト ファイルで作成します。COBOL の場合、IDL 定義の作成は自動化されています。

- CorbaGen ユーティリティは、IDL を自動的に生成します。また、必要に応じてIDLを処理する makefile も生成します。
- Orbix 用 CORBA ウィザードは、IDL を自動的に生成し、Orbix IDL コンパイラで IDL をコンパイルして、スタブを作成します。生成されたスタブは、ウィザードにより NetExpress Orbix プロジェクトに挿入され

ます。

各 IDLファイルには、インターフェイス定義を 1 つ以上記述することができます。各インターフェイス定義では、起動できる操作とそれらの操作に対するパラメータをインターフェイスとして定義します。次に、例を挙げます。

```
//  
  
// Sample IDL  
  
//  
  
interface sample  
{  
  
    struct parm_struct {  
  
        string<50> message;  
  
        float      calc_result;  
  
        long       code;  
  
    };  
  
    exception FaultDetected {string reason; };  
  
    unsigned long first_operation(  
  
        in string<12>  arg1,  
  
        out parm_struct arg2,  
  
        out string<80> arg3  
  
    ) raises (FaultDetected) ;  
  
    unsigned long next_operation(  
  
        in  short      arg1,  
  
        out parm_struct arg2  
  
    );  
  
};
```

上記の例では、次の内容を記述しています。

- 下記の 2 つの操作でパラメータとして使用される構造体を定義する型定義
- ユーザー例外定義
- "first\_operation" と "next\_operation" の 2 つの操作シグネチャ (最初の操作で、定義済みのユーザー例外を挙げるすることができます)。

IDL 構文の詳細については、Iona 社が提供する『ORBIX リファレンス ガイド』を参照してください。

## 2.4.2 IDL の使用方法

IDL 指定が完全に準備できた後の通常の手順を次に示します。

- ORBIX IDL コンパイラを使用して IDL を処理し、ORBIX にコンポーネントを接続するために使用するクライアント スタブとサーバスタブを生成します。
- 実行可能な ORBIX コンポーネントを作成するために、アプリケーション コードを使ってコンポーネントをコンパイルし、リンクします。

ただし、NetExpress では上記の作業は自動化されています。また、Orbix 用 CORBA ウィザードでは、プロセスを簡略化するために IDL が表示されません。

- コマンド行 CorbaGen ユーティリティは、IDL を処理する makefile を自動生成します。
- Orbix 用 CORBA ウィザードは、自動的に IDL をコンパイルしてから、作成する NetExpress Orbix プロジェクトに、生成したスタブを挿入します。

---

注記: コンポーネントを実行するプラットフォームで IDL をコンパイルする必要があります。これは、IDL コンパイラが作成したコードは、実行されるプラットフォームに固有のコードであるためです。生成されたコードは、環境固有の問題を処理し、そのような問題をアプリケーションから除去します。その結果、このコードは、環境固有のコードから汎用性のあるコードになります。

---

## 2.5 Orbix を使用した COBOL 分散アプリケーションの開発

この項では、ORBIX 分散アプリケーション開発の背後にある理論について説明します。NetExpress で提供されるデモンストレーション用の CORBA アプリケーションに添付された readme.txt ファイルには、簡単な、手順ごとの指示が含まれているので、これを参照することもできます。このデモンストレーション アプリケーションと readme.txt ファイルは、NetExpress でインストールする基本ディレクトリの ¥DEMO¥CORBA¥ORDER ディレクトリにあります。

## 2.5.1 NetExpress の Orbix 用 CORBA ウィザードの概要

Micro Focus の Orbix 用 CORBA ウィザードを使用すると、NetExpress プロジェクトで ORBIX 分散アプリケーションを自動的に開発することができます。

ウィザードを起動するには、NetExpress の [ファイル]メニューから [新規作成] を選択し、[Orbix 用 CORBA ウィザード] を選択します。ウィザードは、NetExpress Orbix プロジェクトを作成するために必要な情報を収集するページです。

- プロジェクトが開いている間にウィザードを起動すると、現在のプロジェクトが NetExpress Orbix プロジェクトに変換されます。
- プロジェクトを開かないでウィザードを起動すると、NetExpress Orbix の新規プロジェクトが作成されます。

NetExpress Orbix プロジェクトには IDL は含まれません。生成された IDL は、プロジェクト ディレクトリに保存されます。ウィザードは、Orbix IDL コンパイラを呼び出し、必要なスタブを生成してから、それをプロジェクトに配置します。

プロジェクトが作成されると、[プロジェクト] メニューの [ビルド] を選択することにより、Orbix 実行可能プログラムをビルドすることができます。ウィザードを使用すると、プロジェクトに必要なオプションをすべて正しく設定できます。

## 2.5.2 NetExpress Orbix サポート ツールの概要

NetExpress の [ツール] メニューでは、次の [Orbix サポート] ユーティリティを使用することができます。

- [Orbix デーモンを起動]

これは、ローカルの Orbix デーモンを起動する場合に使用します。Orbix デーモンは、Iona 社の Orbix の一部で、Orbix コンポーネント間の最初のハンドシェイクを実行するために使用されます。

- [Orbix にサーバーを登録]

これは、Orbix デーモンで現在の Orbix COBOL サーバーの詳細を登録するために使用します。デーモンは最初に起動する必要があります。インターフェイス名と実行可能な Orbix COBOL サーバーへのパスを指定できるため、デーモンはクライアントからの要求に応じてサーバーを起動することができます。

## 2.5.3 CorbaGen コマンド行ユーティリティの概要

Micro Focus の CorbaGen コマンド行ユーティリティを使用すると、NetExpress コマンド プロンプトのコンテキストで ORBIX 分散アプリケーションを自動的に開発することができます。実行時に、1 つまたは複数の COBOL ソース ファイルを入力として受け付け、インターフェイスの詳細を抽出します。その結果、指定したコマンド行指令に応じて、次のファイルやラッパーを生成します。

- IDL ファイル

COBOL コードへのインターフェイスを記述する CORBA IDL。主エントリ ポイントと各 ENTRY 文は IDL 操作として定義します。

- CorbaGen に対する入力をカプセル化する C++ または OrbixCOBOL サーバー ラッパー

このモジュールは、生成された IDL ファイルで定義されたインターフェイスに対してサーバーとして機能する ORBIX COBOL コンポーネントへの主エントリ ポイントです。コードの実行内容は、次のとおりです。

- 主エントリ ポイントで、ORBIX の初期化を実行する。
  - 特定の副エントリ ポイントで、定義された各 IDL 操作に対して、正しいパラメータを使用して実際のビジネス ロジックを呼び出す。
- 定義したインターフェイスを呼び出す COBOL コードをカプセル化するための、オプションの C++ または OrbixCOBOL クライアント ラッパー。コード内容は、次のとおりです。
    - 主エントリ ポイントで、ORBIX の初期化を実行する。
    - クライアントと Orbix の間でインターフェイスとして機能する定義済みの IDL 操作には、副エントリ ポイントも使用できる。
    - ORBIX 分散コンポーネントをビルドする makefile (また、-client オプションを使用してクライアント モジュールの名前を識別する場合には、ORBIX クライアント)。

---

注記: ORBIX COBOL サーバーを呼び出すために COBOL クライアントを使用する必要はありません。生成された IDL を使用して、他の言語 (Visual Basic や Java など) で作成されたクライアントをビルドすることができます。コンポーネント間では完全な言語透過性がサポートされています。

---

## 2.5.4 レガシー アプリケーションの基本的な移行手順

レガシー COBOL アプリケーションを ORBIX 分散環境に移行するには、次の手順にしたがいます。

1. まず、ORBIX サーバーコンポーネントを形成するビジネス ロジックを識別し、切り離す必要があります。

ORBIX コンポーネントは、複数のインターフェイスをサポートする一方で、他のコンポーネントでサポートされているインターフェイスのクライアントとなることも可能です。

さらに、サーバー ロジックに対するクライアントとして動作するビジネス ロジックを識別し、切り離すこともでき



ます。この場合、サーバー ロジックとクライアント ロジックを同じプラットフォームで互いを呼び出し合う 2 つのモジュールとしてビルドすると、分散サポートを追加する前に、サーバー ロジックをローカルでテストすることができます。

また、分散ビジネス ロジックを呼び出すクライアントが実際には、Java や Visual Basic などの他の言語で作成されている場合でも、ORBIX 分散サポートを追加すると、クライアント ロジックを使用して、サーバー ロジックを調べることができます。

- ORBIX 分散サポートを追加する準備ができたら、NetExpress の [ファイル] メニューで [新規作成] を選択します。表示されたリストからウィザードを選択し、Orbix 用 CORBA ウィザードを起動します。さらに、次の手順にしたがって、ウィザードを使用します。
- NetExpress プロジェクトが開いている場合は、現在のプロジェクトを NetExpress Orbix プロジェクトに変換することにより、この NetExpress プロジェクトのファイルを分散させます。
- 現在開いている NetExpress プロジェクトがない場合は、NetExpress Orbix プロジェクトを作成し、Orbix コンポーネントに変換する COBOL とポピュレートします。
- NetExpress プロジェクトを作成すると、実行可能プログラムをビルドすることができます。
- 実行可能プログラムをテストするには、次のように [ツール] メニューの [Orbix サポート] オプションを使用します。
- Orbix デーモンを起動します。
- Orbix デーモンで実行可能な Orbix サーバーを登録します。
- Orbix COBOL クライアントを実行します。これにตอบสนองして、Orbix デーモンは自動的に Orbix COBOL サーバーを起動します。

Orbix 用 COBOL ウィザードを使用する代わりに、NetExpress の DOS プロンプトから CorbaGen ユーティリティを使用することもできます。最初に、このユーティリティを使用して、必要なサポートをすべて生成します。実行可能プログラムをビルドするには、CorbaGen ユーティリティにより生成された makefile を実行します。

---

注記: 上記の手順では、ORBIX の各コンポーネントが 1 つの特定のインターフェイスのサーバーまたはクライアントのどちらかである場合を想定しています。ただし、実際には、複数のインターフェイスをサポートする ORBIX コンポーネント、または、あるインターフェイスのサーバーであると同時に他のコンポーネントのインターフェイスのクライアントとしても動作する ORBIX コンポーネントをビルドすることもできます。このような ORBIX コンポーネントをビルドする場合、アクセスする必要があるすべてのインターフェイスをサポートするように、ORBIX コンポーネントのラッパーで生成されるサポートを修正します。

---

## 2.5.5 ORBIX アプリケーションの基本的な新規開発手順

ORBIX アプリケーションを新規に開発するには、次の手順にしがいます。

1. 新規コンポーネントのインターフェイスを設計します。
2. インターフェイスによりサポートされている各操作に対してサーバー ロジックと、コードとは別の ENTRY 文を作成します。
3. 前項で説明した Orbix 用 CORBA ウィザード、または、CorbaGen ユーティリティのどちらかを使用します。

## 2.6 CORBA ウィザード for Orbix

CORBA ウィザード for Orbixは、NetExpress プロジェクトで ORBIX 分散アプリケーションを自動的に開発するための NetExpress のウィザードです。ウィザードを起動するには、NetExpress の [ファイル] メニューから [新規作成] を選択し、[CORBA ウィザード for Orbix] をクリックします。その結果、次の画面が表示されます。

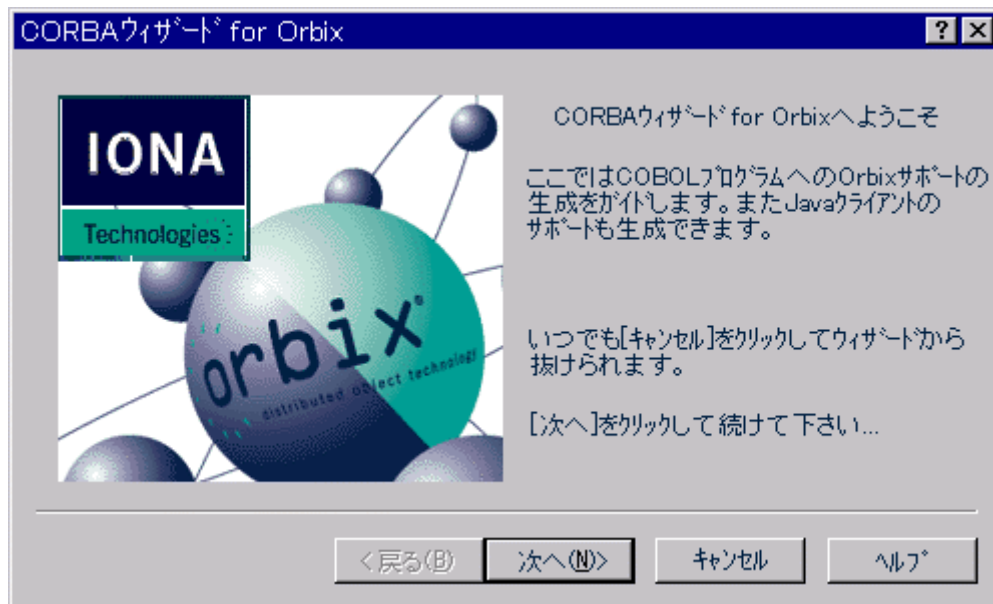


図 2-2 ウィザードへようこそ

この画面には、ウィザードの紹介、使用目的の簡単な説明などが表示されます。先に進むには、[次へ] ボタンをクリックします。

- 開いている NetExpress プロジェクトがない状態でウィザードを起動した場合、[次へ] をクリックすると、「新規プロジェクト」のページが表示されます。

- NetExpress プロジェクトを開いた状態でウィザードを起動した場合、[次へ] をクリックすると、分散させるプロジェクト ファイルを選択するためのページが表示されます。

---

注記: すべてのウィザード ページには [キャンセル] ボタンと [ヘルプ] ボタンがあります。[キャンセル] ボタンを押すと、いつでもウィザードを取り消すことができます。[ヘルプ] ボタンを押すと、いつでもウィザードのオンライン ヘルプを起動することができます。

---

## 2.6.1 新規プロジェクトの作成

開いている NetExpress プロジェクトがない状態でウィザードを起動した場合、[次へ] をクリックすると、次のように表示されます。

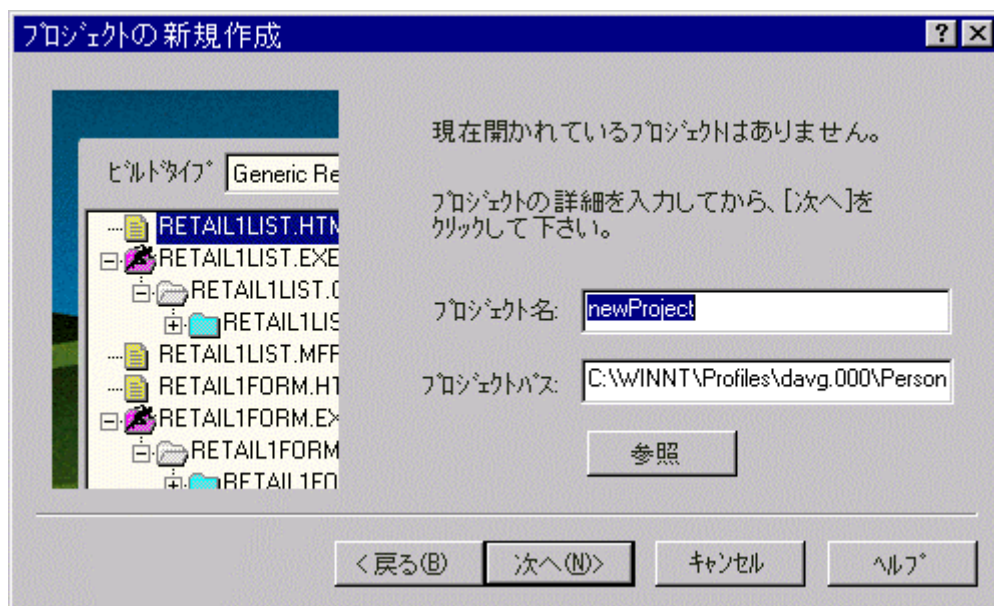


図 2-3 新規プロジェクトの作成

ここでは、NetExpress の新規プロジェクト名と、プロジェクトを検索するディレクトリを指定することができます。入力が終了したら、[次へ] をクリックします。その結果、次のように表示されます。



図 2-4 新規プロジェクトに使用するファイルの選択

この 2 ページ目では、NetExpress の新規プロジェクトを構成するファイルを選択することができます。少なくとも 1 つのファイルを選択しないかぎり、先に進むことはできません。ファイルは、[追加] ボタンをクリックすると選択できます。その結果、標準的なファイル選択ダイアログが起動されます。1 つ以上のファイルを選択すると、[次へ] ボタンが有効化されます。

[次へ] をクリックすると、新規プロジェクトが作成されます。

## 2.6.2 Orbix サーバー用ファイルの選択

プロジェクトを作成すると (または、ウィザードを起動したときにプロジェクトがすでに開いていると)、次のように表示されます。

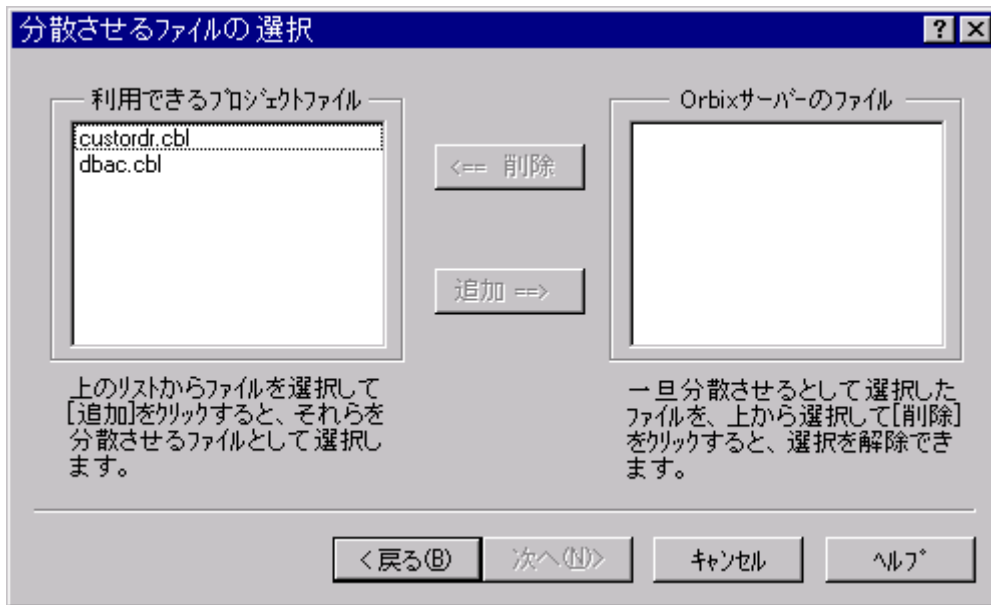


図 2-5 Orbix サーバー用ファイルの選択

[利用できるプロジェクト ファイル]のリストからファイルを選択し、[追加] ボタンをクリックして、Orbix サーバーに追加します。選択したファイルは、左側のリスト ボックスから削除され、右側のリスト ボックス「Orbix サーバーのファイル」に追加されます。右側のリストボックスからファイルを選択し、[削除] をクリックすると、「Orbix サーバーのファイル」リストからファイルを削除できます。

[次へ] ボタンを有効化するには、Orbix サーバーに少なくとも 1 つのファイルを選択します。選択が終了したら、[次へ] をクリックします。

### 2.6.3 クライアントとサーバーのオプション指定

次のページでは、クライアントとサーバーのオプションを指定することができます。

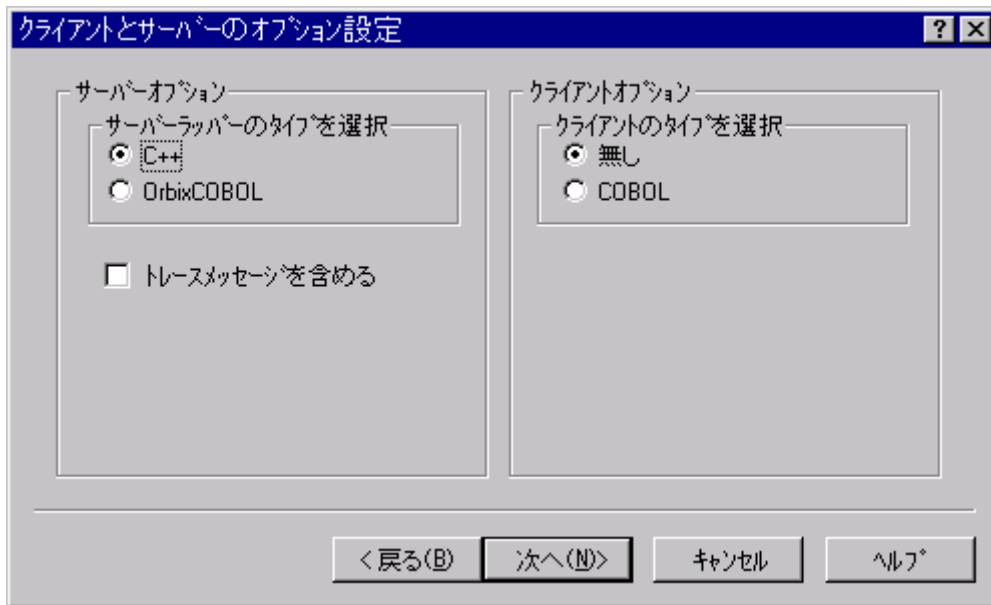


図 2-6 クライアントとサーバーのオプション指定

使用できるサーバー オプションは、次のとおりです。

- ラッパーの種類

Orbix にコンポーネントを接続するために作成するラッパーの種類を指定できます。IDL は、C++ または OrbixCOBOL のスタブを生成するために使用されます。

- トレース

トレース メッセージは、サーバー ラッパーで生成することができます。

オプションで、Orbix COBOL サーバーのインターフェイスを使用する Orbix COBOL クライアントに対するサポートも選択することができます。Orbix COBOL クライアントを選択した場合、[次へ] ボタンをクリックすると、Orbix COBOL クライアントに対して COBOL モジュールを選択するように指示する次のウィザード ページが表示されません。

#### 2.6.4 Orbix クライアント用ファイルの選択

このページは、上記のページで「クライアント オプション」に「COBOL」を選択した場合にだけ表示されます。

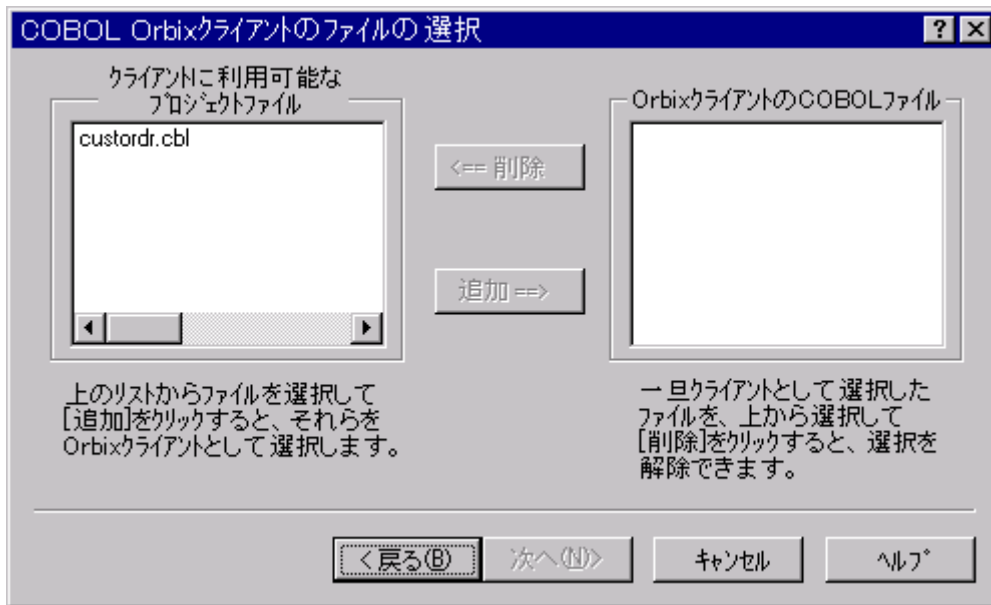


図 2-7 Orbix クライアント用ファイルの選択

ファイルを選択し、[追加] をクリックすると、Orbix クライアントにファイルを追加することができます。選択されたファイルは「クライアントに利用可能なプロジェクト ファイル」のリストから削除され、右側のリストボックス「Orbix クライアントの COBOL ファイル」に表示されます。「Orbix クライアントの COBOL ファイル」に表示されたファイルを選択し、[削除] をクリックすると、このリストからファイルを削除することができます。

[次へ] ボタンを有効化するには、Orbix クライアントに対して少なくとも 1 つのプロジェクト ファイルを選択する必要があります。

正しく選択したら、[次へ] をクリックします。

## 2.6.5 生成

次に示すページでは、分散サポート ファイルを生成することができます。



図 2-8 生成

[生成] ボタンをクリックすると、Orbix サポートファイルが生成されます。生成プロセスが進むにつれて、リストボックスにメッセージが表示されます。生成が完了すると、[次へ] ボタンが有効化されます。[次へ] をクリックします。

## 2.6.6 最終ページ

最後に、プロジェクトに挿入されるすべてのファイルが表示されます。



図 2-9 最終ページ



[終了] をクリックすると、必要な Orbix サポートをすべて挿入し、正しいオプションをすべて設定して、NetExpress プロジェクトを Orbix NetExpress プロジェクトに変換することができます。

## 2.6.7 実行形式ファイルのビルド

Orbix NetExpress プロジェクトを生成したら、それが実行形式ファイルをビルドするのに必要なすべてのファイルと設定を含んでいることがわかります。実行形式ファイルをビルドするには、NetExpress [プロジェクト] メニュー下の [リビルド] を選びます。

## 2.7 Orbix サポート ユーティリティ

NetExpress の [ツール] メニューから、[Orbix サポート] オプションを選択することができます。

- Orbix デーモンを起動

これは、ローカルで Iona 社の Orbix デーモンを起動します。

- Orbix にサーバーを登録

このオプションを選択すると、次のように表示されます。

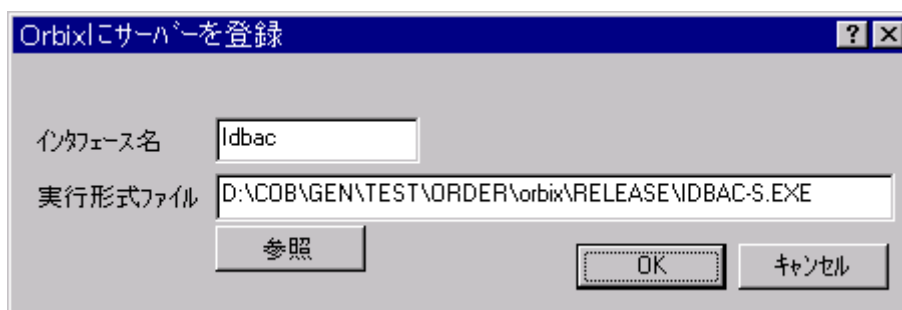


図 2-10 サーバーの登録

Orbix サーバーのインターフェイス名と実行可能なサーバーへのパスを指定し、[OK] をクリックします。その結果、Orbix putit コマンドが作成されるので、このコマンドを実行して Orbix デーモンでサーバーの詳細を登録します。

現在、NetExpress Orbix プロジェクトが開いている場合、このダイアログのフィールドは最初に自動的に入力されます。

## 2.8 CorbaGen コマンド行ユーティリティ

CorbaGen ユーティリティは、NetExpress のコマンド プロンプトから使用できるコマンド行ツールで、COBOL ア

アプリケーションに対する ORBIX サポートを提供します。

## 2.8.1 CorbaGen 構文

CorbaGen コマンドの構文は次のとおりです。

```
CorbaGen [options] filename [filename...]
```

*filename* COBOL ソース ファイルの名前です。

*options* CorbaGen のオプションです。

有効な CorbaGen オプションを、次の表に示します。

| CorbaGen オプション                               | 説明   |   |
|--|--|---|
| <i>/cob option</i> [ <i>/cob option...</i> ] | <p><i>option</i> は、CorbaGen に渡される COBOL コンパイラ オプションです。</p> <p>通常は、このオプションを使用する必要ありません。ただし、特定のプリプロセッサを指定する場合のように、このオプションが必要になることがあります。</p>                                      |   |
| <i>/make</i>                                 | CorbaGen に、実行可能プログラムをビルドする <i>makefile</i> を生成するように命令します。  |   |
| <i>/client basename</i>                      | CorbaGen にクライアントの名前を渡します。次のような指定もできます。   | <ul style="list-style-type: none"><li>• COBOL クライアントのラッパーとサーバー ラッパーが生成されます。</li><li>• <i>/make</i> オプションが指定されている場合、<i>makefile</i> は実行可能なクライアントと実行可能なサーバーをビルドします。</li></ul> |
| <i>/v</i>                                    | 詳細オプションです。これが設定されている場合、生成された <i>makefile</i> で詳細フラグを設定します。   |   |
| <i>/debug</i>                                | CorbaGen に対し、生成されたサーバー ラッパーにデバッグ コードを挿入し、生成された <i>makefile</i> で COBOL コンパイラのコマンド行に <i>animate</i> オプションを挿入するよう命令します。実行時には、このフラグを設定した Orbix コンポーネントは、次のような詳細なテキスト メッセージを発行します。 | <ul style="list-style-type: none"><li>• レガシー COBOL に対する各呼び出し</li><li>• レガシー COBOL 呼び出しからの各戻り値</li></ul>   |
| <i>/octet</i>                                | このオプションでは、COBOL DBCS の型 (PIC N と PIC G) を 8 ビットの配列   |   |

として処理するように指定します。このオプションが指定されていない場合、IDL の `wstring` 型と `wchar` 型が使用されます (Orbix のバージョンとプラットフォームによっては `wchar` 型と `wstring` IDL 型を処理しないため、デフォルトの代わりにこのオプションを使用できるようにしてあります)。

一度呼び出されると、CorbaGen コマンドは、次のような処理を行います。

- 指定したソース ファイルからインターフェイスの詳細を抽出する。
- 次のように要求された出力を生成する。
  - インターフェイスの CORBA IDL 記述。
  - 指定した COBOL コンポーネントの ORBIX C++ ラッパーまたは OrbixCOBOL ラッパー。ラッパーには、コンポーネントを ORBIX サーバーとして初期化するためのコードが記述されます。
  - COBOL クライアントの ORBIX C++ ラッパーまたは OrbixCOBOL ラッパー。ラッパーには、クライアントを ORBIX クライアントとして初期化するコードが記述されます。
  - 実行可能プログラムをビルドする `makefile`。

## 2.9 ORBIX クライアントと ORBIX サーバーのデバッグとアニメート

通常、他のアプリケーションに適用するテストを使用することができます。ただし、ORBIX コンポーネントは、分散アプリケーションなので、テスト時には別途考慮すべきことがあります。

### 2.9.1 テスト方法

ORBIX 分散を追加する前に、ローカル ホストで COBOL サーバー ロジックをテストすることをお勧めします。テスト内容は、次のとおりです。

- シーケンス エラーがないことを確認する。
- ロジック エラーがないことを確認する。
- 予期しない結果を検知するために、文の実行前後のデータ値を調べる。

ORBIX コンポーネントをテストする場合、次のような手順にしたがうことをお勧めします。

- NetExpress を使用して、COBOL のクライアント ロジックとサーバー ロジックを、互いに呼び出し合うスタンドアロン プログラムとしてテストし、デバッグします。この場合、ループでは ORBIX 分散サポートを使用しません。(他の言語で作成された ORBIX コンポーネントによりアクセスされる COBOL サーバーを作成している場合でも、最初にサーバー ロジックをチェックアウトする COBOL クライアント テスト ハーネスを作成すると便利です。)

- クライアント ロジックとサーバー ロジックが正しいことを確認したら、Orbix 用 CORBA ウィザードか CorbaGen ユーティリティを使用して、ORBIX 分散ロジックを追加し、COBOL コードを ORBIX 分散コンポーネントに変換します。
- ORBIX サーバーコンポーネントと ORBIX クライアントコンポーネントを一緒にテストします。
- 最後に ORBIXサーバーコンポーネントとこれを使用する実際の ORBIX コンポーネント (Java または Visual Basic で作成されたコンポーネント) を一緒にテストします。

## 2.9.2 コードのアニメート方法

ORBIX アプリケーションをアニメーションのために設定するには、次のアプローチの 1 つを選択します。

- corbagenコマンド行 ユーティリティを使用する。
- 代わりに、ORBIX 用の CORBA ウィザードを使用する。

以降に、これらのアプローチをより詳しく説明します。

### 2.9.2.1 アニメーションの設定のために corbagen を使用する

以下の手順に従ってください。

- アニメーションの生成

[/debug] オプションで、corbagenコマンド行 ユーティリティを使います。生成された makefile は、NetExpress を使ってアニメートできる実行形式ファイルをビルドするのに必要なオプションをすべて含んでいます。

- 実行形式ファイルのビルド

ORBIX 実行形式ファイルをコンパイルとリンクするには、nmakeで makefile を処理します。

- アニメート

実行形式ファイルができたなら、次に NetExpress を使ってアニメートします。これを行うには、コマンド行 プロンプトから NetExpress を次のように起動します。

```
mfnetx program-name
```

ここで`program-name`は、.exeモジュールの名前です。

NetExpress が起動したら、次に進んで、選択した ORBIX アプリケーションをアニメートできます。

クライアントをアニメートしている場合、まず、ORBIX でサーバーを登録する必要があります。そうすることによって、クライアントからの要求に応じてアニメーションは、自動的に開始します。

サーバーをアニメートしている場合、サーバー アニメーションを開始する必要があります。サーバーがクライアントの入力を待っている間に、NetExpress コマンド プロンプトから、クライアントを手動で起動できます。

### 2.9.2.2 アニメーションの設定のために ORBIX 用の CORBA ウィザードを使用する

以下の手順に従ってください。

- アニメーションの生成

ORBIX の CORBA ウィザードを使用しているとき、選択する必要があるデバッグ オプションがあります。[クライアントとサーバーの設定] オプションパネルで、[メッセージのデバッグとトレース] ボックスをクリックする必要があります。このアクションにより、生成コードにデバッグ メッセージが生成され、また NetExpress プロジェクトに必要なアニメーション オプションが自動的に設定されます。

ORBIX アプリケーションを既に生成している場合、次の 2 つの選択肢があります。

- プロジェクトを生成するか、
- 代わりに、すべてのプロジェクトの COBOL ファイルのビルド設定で [ANIM] オプションを手動で設定します。

- 実行形式ファイルのビルド

[一般デバッグビルド] をビルドのタイプとして選択します。次に、[プロジェクト] メニューで、[リビルド] オプションを選択して実行形式ファイルをビルドします。

- アニメート

クライアントをアニメートするには、まず ORBIX でサーバーを登録します。登録が済んだら、ORBIX クライアントの.exeを右クリックし、使用できるオプションのリストから、[アニメート] を選択します。

サーバーをアニメートしている場合、ORBIX サーバーの.exeを右クリックします。サーバーがクライアントの入力を待っている間に、NetExpress コマンドプロンプトから、クライアントを手動で起動できます。

## 2.10 ORBIX サポート制限事項

ORBIX サポートを COBOL アプリケーションに適用しているとき、次の制限事項が適用されます。

- POINTER をパラメータとして使うのはサポートしません。POINTER タイプは、あまりにも漠然としています。ORBIX は、整然とされるべきデータの特定の定義を必要とします。
- ADDRESS を渡したり返したり (例えば、EXIT PROGRAM RETURNING ADDRESS OF xxを使って戻る COBOL プログラム) は、サポートしません。これを行う COBOL アプリケーションへの ORBIX サポート

トを適用すると、返却アドレスを使おうとするクライアント アプリケーションによって持ち出される例外に潜在的になります。

## 第3章 他のオブジェクト ドメインの使用

Object COBOL では、オブジェクト リクエスト ブローカ (ORB) を使用して、オブジェクトからメッセージを送受信することができます。

Object COBOL は、各 ORB を通じてアクセスされるオブジェクトに別々のドメインを指定します。Object COBOL プログラムは、複数の異なるドメインからオブジェクトにアクセスすることができます。ドメインを通してアクセスされる各オブジェクトは、さまざまな言語で作成することができます。また、常にではありませんが、別のプロセスで実行することもできます。

### 3.1 概要

Object COBOL で使用できるさまざまな型の ORB は、さまざまなレベルの機能を装備しています。各 ORB は、異なるドメインで表され、各ドメインについて、最低でも次の内容がサポートされています。

- クライアント サポート

別のドメインのオブジェクトに Object COBOL プログラムからメッセージを送信します。この場合、Object COBOL のプログラムやクラスは、サーバー オブジェクトのクライアントになります。

- サーバー サポート

別の (または同じ) ドメインのオブジェクトからメッセージを受け取ります。この場合、Object COBOL クラスは、ORB を通じて受信した要求を処理するサーバー オブジェクトになります。

- 型指定:

ORB がサポートするデータ型は、COBOL と比べると少なくなります。各ドメインは、COBOL データ型と ORB でサポートされているデータ型の間で自動的に型を指定します。COBOL データ型を ORB データ型に直接変換できない場合は (たとえば、編集されたフィールドなど)、文字列型に変換されます。

ORB によっては、追加サービスを提供するものがあります。サービスについては、ORB ごとにまとめられています。

### 3.2 汎用のドメイン サポート

次の項では、すべての ORB に共通するドメイン サポートについて説明します。

- 別のドメインからオブジェクトを識別する。
- 別のドメインにあるオブジェクトへメッセージを送信する。

### 3.2.1 別のドメインからのオブジェクト識別

別のドメインのサーバー オブジェクトにアクセスする場合、そのサーバー オブジェクトがあるドメインを識別する必要があります。これは、Class-Control の段落を通じて行います。

```
class-control.  
    class is "$domain-name$class-name".
```

パラメータの内容は、次のとおりです。

|                    |  |
|--------------------|--|
| <i>domain-name</i> | オブジェクトのドメイン名。  |
| <i>class-name</i>  | ドメインがクラスを認識するための名前。これは、通常、ORB でそのドメインについて登録したクラス名により決まります。 |

次のコード例では、2 つのクラス オブジェクト CharacterArray と Excel を識別します。CharacterArray はネイティブの Object COBOL オブジェクトで、Excel は \$OLE\$ で識別できる OLE オートメーション サーバーです。

```
class-control.  
    CharacterArray is class "chararray"  
    Excel is class "$OLE$excel.application"  
    .
```

Excel のインスタンスを使用するには、OLE オートメーション サーバーに次のメッセージ "new" を送信します。

```
invoke excel "new" returning anExcelServer
```

このメッセージにより、OLE オートメーション サーバーのインスタンスが作成されます。この段階で、ExcelServer にメッセージを送信できるようになります。

### 3.2.2 別のドメインのオブジェクトに対するメッセージ送信

クラスが別のドメインにあることが判明した場合、INVOKE 文を使用してメッセージを送信することができます。別のドメインで実行されているオブジェクトと通信するには、Object COBOL オブジェクトと通信する場合と同じ構文で INVOKE 文を使用します。

別のドメインですべての COBOL データ型がサポートされているとはかぎらないので、パラメータは別の形式に変換されてからメッセージと送信されることがあります。データ キャスト規則は、各ドメインごとにまとめられています。

メッセージ セレクタによっては、別のドメインで特別な意味を持つ場合があります。たとえば、"get" と "set" が前に付くメッセージ セレクタは、OLE ドメインで、自動的にプロパティ設定操作またはプロパティ表示操作に変換されます。これらの規則も、ドメインごとにまとめられています。



## 第4章 OLE オートメーションと DCOM

オブジェクトのリンクと埋め込み (OLE) は、Microsoft Windows の機能です。OLE を使用すると、アプリケーションで、他のアプリケーションでも利用できる機能を公開することができます。そのため、既存のパッケージとカスタム ソフトウェアと一緒に使用して、新しいアプリケーションを作成することができます。Microsoft Office の全アプリケーションと、Microsoft BackOffice の多くのアプリケーションでは、OLE オートメーションを通して機能を公開しています。これらのアプリケーションの一部を再利用して、共通の機能を実行することができます。たとえば、ユーザー作成のアプリケーションで見やすいレポートを生成し、印刷する場合、Microsoft Word の機能を使用して、レポートを作成し、印刷することができます。

### 4.1 概要

Micro Focus の OLE オートメーション サポートを使用すると、Object COBOL のプログラムとクラス (ActiveX クライアントとしての Object COBOL) から ActiveX にメッセージを送信することができます。また、ActiveX オブジェクトを作成し、OLE オートメーションを通して Object COBOL クラスを操作することもできます。

OLE オートメーションをビルドする Component Object Model (COM) と Object COBOL のような 00 言語のモデルの間には、次のような 2 つの大きな違いがあります。

- COM は、継承をサポートしていない。
- COM は、クラス メソッドとクラスデータをサポートしていない。

OLE サポートについては、主に、次の 3 項目にまとめることができます。

- ActiveX クライアントの作成

ActiveX クライアントは、ActiveX オブジェクトが公開する機能にアクセスし、これらの ActiveX オブジェクトを制御するアプリケーションです。

- ActiveX オブジェクトの作成

ActiveX オブジェクトは、他のアプリケーションから制御できる機能を提供するアプリケーションです。このようなオブジェクトの例として、Microsoft Word などが挙げられます。Micro Focus ActiveX サーバは、デュアルインターフェイスをサポートし、NetExpress クラスとメソッド ウィザードは、これをサポートするためにタイプ ライブラリを自動的に作成しアップデートします。

- OLE データ型指定規則

Object COBOL の ActiveX クライアントとオブジェクトは、必要な場合に自動的に DCOM を使用します。この場合、レジストリ エントリを変更するだけですみます。詳細は、この章の「DCOM の使用」を参照してください。

---

注記: ActiveX クライアントと ActiveX オブジェクトは、以前、OLE オートメーション クライアントと OLE オートメーション サーバーという名称でした。

---

#### 4.1.1 チェックリスト - プロジェクトを始める前に

OLE と DCOM の技術は、急速に発達しています。また、プロジェクトに使用するサードパーティのソフトウェアが必ずしも必要な機能をサポートしているとはかぎりません。そのため、プロジェクトを開始する前に、すべてのサードパーティの最新版ソフトウェアを入手し、必要な機能がすべてサポートされていることを確認してください。

特に、次の点を確認してください。

- オペレーティング システムのリリースが必要なレベルの機能をサポートしているかどうか。

たとえば、プロセス間でローカル通信を行うための OLE オートメーションは、Windows 95 と Windows NT のすべてのリリースで使用可能です。一方、インプロセス サーバー用の DCOM (別のマシンとの間で OLE オートメーションを使用している場合) は、Windows NT 4.0 とサービス パック 2、またはそれ以降のリリースでしかサポートされていません。Windows 95 で DCOM を使用するには、DCOM サポートを追加する必要があります。

この情報は、Microsoft の Web サイトから入手できます (詳細情報を参照してください)。

- 使用するサードパーティのソフトウェアまたは言語ツールが、すべて、必要なレベルの機能をサポートしているかどうか。

たとえば、DCOM は、Visual Basic 5.0 以降のリリースでしかサポートされていません。

## 4.2 ActiveX クライアントの作成

Object COBOL プログラムからは、次の操作を行うことができます。

- ActiveX オブジェクトにメッセージを送信する。
- ActiveX プロパティを設定または取得する。

これにより、OLE オートメーション インターフェイスをもつ Windows アプリケーションを Object COBOL から直接制御できるようになります。このような ActiveX オブジェクトを使用するプログラムを ActiveX クライアントと呼びます。

COM を使用すると、ActiveX オブジェクトをクライアントと異なる言語で作成することができます。また、COM は、クライアントとオブジェクトの間で送受信できるデータの型と形式を定義します。Object COBOL プログラムから

OLE オブジェクトに送信されるすべてのデータは、「OLE データ型」の章で説明する規則にしたがって型指定されます。

Object COBOL の OLE オートメーション サポートは、クライアントが使用している各 ActiveX オブジェクトのプロキシを指定することにより、ActiveX オブジェクトを Object COBOL オブジェクトとして表します。クライアントは、まずプロキシにメッセージを送信します。これらのメッセージは、Object COBOL ランタイム システムにより ActiveX オブジェクトに転送されます。

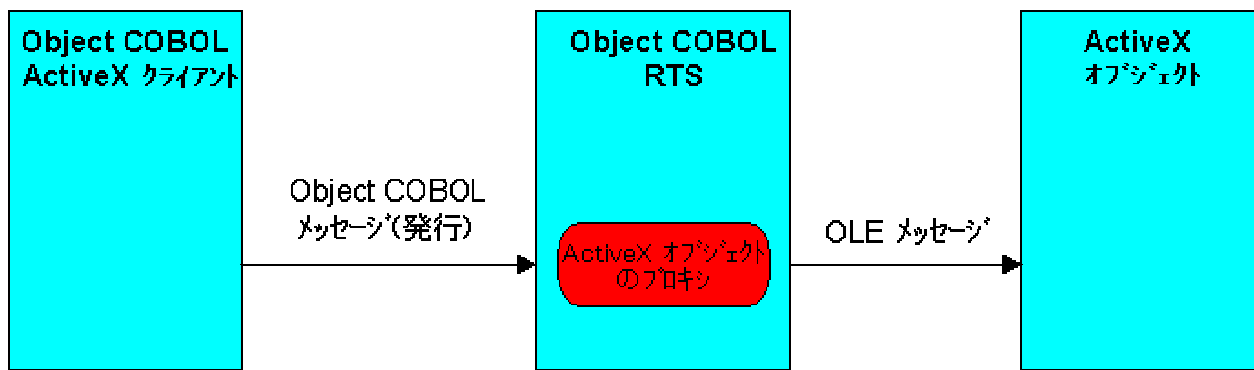


図 4-1 ActiveX プロキシにメッセージを送信する Object COBOL クライアント

次の各項では、ActiveX クライアントの作成手順について説明します。

- プログラムの ActiveX クライアント化
- ActiveX プロキシ オブジェクトの作成
- ActiveX オブジェクトへのメッセージ送信
- ActiveX プロキシ オブジェクトの終了
- OLE オートメーションの例外
- タイプ ライブラリ情報の使用

#### 4.2.1 プログラムの ActiveX クライアント化

Object COBOL プログラムは、ActiveX クライアントにすることができます。Object COBOL クラスを作成する必要はありません。必要な作業は、次のとおりです。

1. プログラムにコンパイラ指令 `OOCTRL(+P)` を設定する。

```
$set ooctrl(+p)
```

注記: これは、起動されたか宣言された任意のメソッドのパラメータの数を 31 プラス オプションの RETURNING パラメータに制限します。

2. 使用する ActiveX オブジェクトを CLASS-CONTROL の段落で Object COBOL クラス名にマップする。

```
class-control.
```

```
class-name IS CLASS "$OLE$windows-registry-name"
```

*windows-registry-name* は、Windows システム レジストリで ActiveX オブジェクトとして入力したサーバー名です。人間が読み取ることができる名前である ProgID、または、世界中の各 ActiveX オブジェクトに固有な 16 バイトの数字 CLSID のどちらかを使用することができます。CLSID は、ActiveX オブジェクトの新しいバージョンがリリースされるたびに変更されることが多いため、通常は ProgID を使用します。

例

```
class-control
```

```
Word is class "$OLE$word.basic"
```

```
HTMLHelp is class "$OLE${adb880a6-d8ff-11cf-9377-00aa003b7a11}"
```

.

3. ActiveX オブジェクトのプロキシを識別するハンドルを管理するために OBJECT REFERENCE データ項目を宣言します。

例

```
$set ooctrl(+P)
```

```
class-control.
```

\*> comploan は、ActiveX オブジェクト名 (ProgID) です。

```
comploan is class "$OLE$comploan"
```

...

.

```
working-storage section.
```

```
01 theCompLoanServer          object reference.
```

## 4.2.2 ActiveX プロキシ オブジェクトの作成

使用する各 ActiveX オブジェクトに対してプロキシ オブジェクトを作成する必要があります。プロキシを作成すると、Object COBOL ランタイム システムは次の 2 つの操作を行います。

- Object COBOL ランタイム システムは、Windows に対して必要な ActiveX を検索するように要求します。オブジェクトがまだ実行されていない場合、そのオブジェクトは Windows により起動されます。
- Object COBOL ランタイム システムは、オブジェクトに対してプロキシを作成し、プロキシのオブジェクト ハンドルを与えます。

プロキシを作成するには、次の手順にしたがいます。

- Object COBOL プロキシ クラスにメッセージ "new" を送信します。

この段階で、プロキシを通して ActiveX オブジェクトにメッセージを送信することができます。

例

```
working-storage section.

01 theCompLoanServer          object reference.

...

procedure division.

...

invoke comploan "new" returning theCompLoanServer

...
```

また、プロキシの作成時にサーバーの場所を指定することもできます。サーバーの場所を指定するには、次の手順にしたがいます。

- Object COBOL プロキシ クラスに、メッセージ "newWithServer" とサーバーのマシン名を送信します。

例

```
working-storage section.

01 theCompLoanServer          object reference.

...

procedure division.

...

invoke comploan "newWithServer" using z"machine1" returning theCompLoanServer

...
```

### 4.2.3 ActiveX オブジェクトへのメッセージ送信

ActiveX オブジェクトを作成すると、このオブジェクトにメッセージを送信し、プロパティを設定したり取得したりすることができます。メッセージの送信とプロパティの取得操作または設定操作は、プロキシ オブジェクトにメッセージを送信するための INVOKE 文によって処理されます。次の図のように、"get" または "set" で始まる名前のメッセージをプロキシ オブジェクトに送信するたびに、このメッセージは、Object COBOL ランタイム システムにより ActiveX オブジェクトで自動的にプロパティ取得操作またはプロパティ設定操作に変換されます。

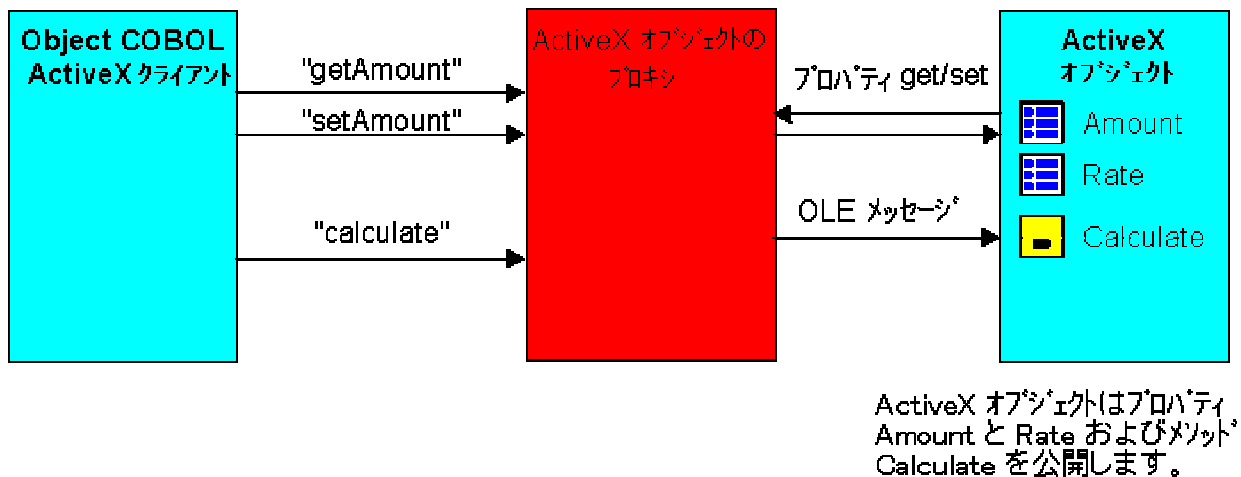


図 4-2 メッセージ送信とプロパティ設定

メッセージを送信するには、次のように記述します。

```
invoke proxyObject "messagename" [using param-1 [param-2...]]  
[returning result]
```

*proxyObject* ActiveX オブジェクトのプロキシ。プロキシを作成し、ActiveX オブジェクトを起動する方法については、「ActiveX プロキシ オブジェクトの作成」の項を参照してください。

*messagename* 送信するメッセージ。

*param-1* メッセージに必要なパラメータ。COBOL データ型は、「OLE データ型」の章で説明するとおり、OLE データ型に変換されます。すべてのパラメータは、リファレンス (COBOL のデフォルト) により渡します。

*result* このメソッドが結果を返した場合の戻り値。COBOL データ型は、「OLE データ型」の章で説明するとおり、OLE データ型に変換されます。

プロパティを設定するには、次のように記述します。

```
invoke proxyObject "setPropertyname" using value
```

*proxyObject* ActiveX オブジェクトのプロキシ。プロキシを作成し、ActiveX オブジェクトを起動する方法については、「ActiveX プロキシ オブジェクトの作成」の項を参照してください。

*PropertyName* 設定するプロパティの名前。

*value* プロパティの新しい値。COBOL データ型は、「OLE データ型」の章で説明するとおり、OLE データ型に変換されます。値は、リファレンス (COBOL のデフォルト) により渡します。

プロパティを取得するには、次のように記述します。

```
invoke proxyObject "getPropertyName" returning value
```

*proxyObject* ActiveX オブジェクトのプロキシ。プロキシを作成し、ActiveX オブジェクトを起動する方法については、「ActiveX プロキシ オブジェクトの作成」の項を参照してください。

*PropertyName* 設定するプロパティの名前。

*value* プロパティの値。OLE データ型は、「OLE データ型」の章で説明するとおり、COBOL データ型に変換されます。この値は、OLE が上書きできるメモリ領域に返されるため、保存する必要がある場合は、値をコピーしてください。

次の例では、プロパティを設定し、メッセージを送信してから、プロパティを取得します。

```
working-storage section.
```

```
...
```

```
01 theCompLoanServer    object reference.
```

```
01 AnAmount             pic 9(7).99.
```

```
01 ARate                pic 99.99.
```

```
01 Amount20             pic $9(7).99.
```

```
...
```

```
procedure division.
```

```
...
```

```
    invoke theCompLoanServer "SetLoanTermYears" using
```

```
        "20"
```

```
    invoke theCompLoanServer "calculate"
```

```
invoke theCompLoanServer "GetYearPayment" returning
```

```
Amount20
```

```
...
```

## OLE メッセージ型の指定

ランタイム システムのデフォルト設定では、Object COBOL から送られた "set" または "get" で始まる OLE メッセージはすべてプロパティ設定操作またはプロパティ取得操作に変換されます。ただし、この設定は、無効にすることができます。また、"set" または "get" が前についていないメッセージをプロパティ設定操作またはプロパティ取得操作となるように強制することもできます。メッセージ型を指定するには、メッセージ "setDispatchType" をクラス olesup (ファイル名 olesup) に送信します。

```
invoke olesup "setDispatchType" using by value lsType
```

```
lsType          PIC X(4) COMP-5
```

値が 0 の場合 = 次のメッセージは、強制的にメソッドを起動します。

値が 1 の場合 = 次のメッセージは、強制的にプロパティ設定を起動します。

値が 2 の場合 = 次のメッセージは、強制的にプロパティ取得を起動します。

メッセージ型は、次に送信する OLE メッセージに対してだけ指定できます。その後は、再度 "setDispatchType" を送信するまでデフォルトの動作に戻ります。クラス olesup は、OLE オートメーション プログラミング用にいくつか便利な関数をカプセル化します。クラス olesup の詳細については、「OLE オートメーション クラス ライブラリ リファレンス」を参照してください。NetExpress の [ヘルプ] メニューで [ヘルプ トピック] をクリックし、[リファレンス]、[OO クラス ライブラリ]、[クラス ライブラリ リファレンス] をクリックし、クラス ライブラリ リファレンスを起動するためのショートカットをクリックしてください。

## 4.2.4 タイプ ライブラリ情報の使用

ActiveX オブジェクトのタイプ ライブラリには、クライアントの作成時に役立つ情報が定義されています。タイプ ライブラリ アシスタントを使用すると、タイプ ライブラリの COBOL 用コピー ファイルを作成することができます。NetExpress の [ツール] メニューで [タイプ ライブラリ アシスタント] をクリックします。

タイプ ライブラリの情報に応じて、COBOL コピー ファイルには、次のような情報が格納されます。

- タイプ ライブラリが定義する各データ型に対する COBOL 型の定義
- ライブラリが定義する各オートメーション インターフェイスについては、次の情報が格納されます。
  - タイプ ライブラリが定義するインターフェイスの IID (インターフェイスID) を含むレベル 78 データ項目



- インターフェイス記述を示すコメント
- インターフェイスの名前を含むコメント付きポインタ データ項目
- インターフェイスの全プロパティの設定方法と取得方法を示すコメント
- インターフェイスの全メソッドを呼び出す方法を示すコメント
- タイプ ライブラリが定義する各クラスについては、次の情報が格納されます。
  - ライブラリが定義するクラスの CLSID と ProgID を含むレベル 78 データ項目
  - クラスを使用するために必要な class-control エントリを示すコメント
  - クラス記述を示すコメント
- 各列挙型のデータ項目

#### 4.2.5 ActiveX プロキシ オブジェクトの終了

Object COBOL ランタイム システムは、ActiveX オブジェクトを実行するために作成したプロキシ オブジェクトを、自動的に終了させることはありません。ActiveX オブジェクトの操作を終了した後で、プロキシに "finalize" メッセージを送信する必要があります。ActiveX オブジェクトへの接続がすべて解除されると、ActiveX オブジェクトは、ActiveX オブジェクトが見えないか明示的なメソッド呼び出しを必要としていないならば、ActiveX オブジェクト自身を通常シャットダウンします (例えば、"quit"は、word 97 をシャットダウンします)。

例

```
invoke theCompLoanServer "finalize" returning theCompLoanServer
```

#### 4.2.6 OLE オートメーションの例外

ActiveX クライアントは、Object COBOL の例外を通して OLE エラー通知を受け取ります。ActiveX オブジェクトで発生した OLE エラーは (たとえば、オブジェクトが認識しないメッセージを送信した場合など)、例外として、Object COBOL で作成された ActiveX クライアントに返されます。デフォルトでは、例外が発生した場合、クライアントが例外を警告するメッセージを表示してから、終了します。例外をトラップし、それを独自のエラー処理コードで処理することもできます。

次の手順では、最初にオンライン ヘルプで例外処理に関する情報を把握していることを前提としています。

NetExpress の [ヘルプ] メニューで [ヘルプ トピック] をクリックし、[プログラミング]、[オブジェクト指向プログラミング]、[クラス ライブラリ フレームワーク]、[例外処理] の順に選択します。

OLE 例外をトラップするには、次の手順にしたがいます。

1. ActiveX クライアントの Class-Control の段落でクラス ExceptionManager、OLEExceptionManager、olesup、

さらに Callback か EntryCallback のどちらか一方 (手順 2 を参照) を宣言します。

```
class-control.  
  
...  
  
OLEExceptionManager is class "oleexpt"  
  
ExceptionHandler is class "exptnmgr"  
  
olesup is class "olesup"  
  
Callback is class "callback"  
  
EntryCallback is class "entrycbl"  
  
...
```

2. 例外ハンドラ (メソッドまたはエントリ ポイント) を記述し、それに Callback または EntryCallback を作成します。たとえば、Callback の場合、次のように記述します。

```
invoke Callback "new" using anObject z"methodName"  
  
    returning aHandler
```

EntryCallback の場合、次のように記述します。

```
invoke EntryCallback "new" using z"entryPointname"  
  
    returning aHandler
```

3. Callback または EntryCallback を OLEExceptionManager クラスに対して登録します。たとえば、次のように記述します。

```
invoke ExceptionManager "register"  
  
    using OLEExceptionManager aHandler
```

これで、クライアントに送信されたすべての OLE 例外が例外ハンドラに送信されます。

例外ハンドラは、2 つのパラメータを受け取ります。最初のパラメータは、OLEExceptionManager のオブジェクト ハンドル、2 番目のパラメータは例外 ID です。OLE エラー コードを取得するには、例外 ID から基本 OLE 例外エラー番号を除く必要があります。"getBaseOleException" メッセージをクラス olesup に送信すると、基本 OLE 例外エラー番号の値を取得することができます。基本番号を取得するには、次のように記述します。

```
invoke OleSup "getBaseOleException" returning lsBase
```

lsBase のデータ型は、次のとおりです。

lsBase PIC X(4) COMP-5

次の表では Object COBOL ランタイム システムにより ExceptionManager を通して返された例外番号を簡単に説明します。

| 値  | 説明                    |
|----|-----------------------|
| 1  | サーバーが定義した OLE 例外      |
| 2  | パラメータ数の不一致            |
| 3  | OLE 型の不一致によるエラー       |
| 4  | OLE 名が見つかりません。        |
| 5  | OLE 操作を行うにはメモリが足りません。 |
| 6  | 名前がメソッドです。            |
| 7  | 名前がプロパティです。           |
| 8  | OLE オートメーション エラー      |
| 9  | OLE サーバーが使用できません。     |
| 10 | OLE サーバーの例外           |

OLE エラーについては、例外メソッドによりメッセージ "getLastSCode" を OleSup に送ると、詳細情報を入手できます

```
invoke olesup "getLastSCode" returning lsErrorCode
```

lsErrorCode のデータ型は、次のとおりです。

lsErrorCode PIC X(4) COMP-5

OLE エラー コードについては、Win 32 SDK のマニュアルを参照してください。

次に示すコード例の最初の部分は、OLE 例外を処理する ole-err1.cbl と呼ばれる短い COBOL サブルーチンです。2 番目の部分では、このサブルーチンを EntryCallback にラップし、これを OLE 例外ハンドラとして登録しています。

例外ハンドラの例

```
class-control.  
  
    olesup is class "olesup"
```

working-storage section.

01 wsOffset                   pic x(4) comp-5.

01 wsOLEException           pic x(4) comp-5.

linkage section.

01 lnkExceptionObject       object reference.

01 lnkExceptionNumber       pic x(4) comp-5.

procedure division using lnkExceptionObject

                          lnkExceptionNumber.

...

    invoke olesup "getBaseOleException" returning wsOffset

    subtract wsOffset from lnkExceptionNumber

                          giving wsOLEException \*> OLE 例外

...

    exit program.

例外ハンドラの登録例

class-control.

...

    EntryCallback is class "entryc11"

    oleexpt is class "oleexpt"

    ExceptionHandler is class "exptnmgr"

.

working-storage section.

...

01 exceptionHandler       object reference.

...

```
procedure division.
```

...

\*> OLE 例外ハンドラの例外を設定します。

```
invoke EntryCallBack "new" using z"exception-section"
```

```
    returning exceptionHandler
```

```
invoke ExceptionManager "register" using oleexpt
```

```
    exceptionHandler
```

...

### 4.3 Object COBOL による ActiveX オブジェクトの作成

Microsoft 社の『OLE プログラマーズ リファレンス』では、ActiveX オブジェクトを「ActiveX クライアントに対してプロパティ、メソッド、イベントを公開するクラスのインスタンス」として定義しています。次の条件を満たす Object COBOL のクラスから ActiveX オブジェクトを作成することができます。

- クラスは OLEBase または OLEBase-s から継承する。

これらのクラスは、OLE オートメーションを通して操作を行うために必要なコードを提供します。OLE オートメーションの基礎となる Component Object Model (COM) は、継承をサポートしません。OLEBase-s は、シングル ユーザー サーバーを指定します。そのため、OLEBase-s に接続しようとする各クライアントに対して新しいサーバーが作成されます。OLEBase は、マルチユーザー サーバーに対して使用します。ほとんどのサーバーは、マルチユーザー サーバーです。インプロセス サーバーについては、シングル ユーザーとマルチユーザーのどちらも適用できません。

クラスは、クラス olebase から継承する必要があります。OLE は、継承をサポートしていないため、OLE クラスからサブクラスを作成することはできません。

- クラスは、OOCTRL(+P) 指令でコンパイルする。

OOCTRL(+P) 指令を使用すると、プログラムが OLE オートメーションを操作するために必要な追加の型確認情報をコンパイラにより入手できます。

注記: これは、起動されたか宣言された任意のメソッドのパラメータの数を 31 プラス オptional の RETURNING パラメータに制限します。

- クラス ID、プログラム ID、およびプログラムパスを Windows で登録する。

クライアントは、Windows レジストリを通してクラスを検索します。登録の形式は、ActiveX オブジェクトの実行方法によって異なります (次を参照)。

NetExpress のクラス ウィザードを使用して、クラスのスケルトン、レジストリ エントリ、タイプ ライブラリ、起動プログラム (必要な場合) を生成します。クラス ウィザードを実行するには、[ファイル] メニューの [新規作成] をクリックし、「新規作成」ダイアログから [クラス] を選択します。

ActiveX プログラムの実行方法は、次のように 3 種類あります。

- アウトオブプロセス

オブジェクトは、クライアントが使用しているプロセスとは別のプロセス領域で実行されます。アウトオブプロセス オブジェクトは、スタンドアロンの .exe プログラムとしてビルドされます。アウトオブプロセスのオブジェクトが破損しても、クライアントはエラーから回復することができます。

- インプロセス

オブジェクトは、クライアントと同じプロセス領域に読み込まれます。インプロセス オブジェクトは、.dll プログラムとしてビルドされます。インプロセス ActiveX オブジェクトへのアクセスはより高速です。ただし、オブジェクトが破損すると (たとえば、保護違反が生じた場合)、クライアントも同様に破損します。

- リモート

クライアントとオブジェクトが別のマシンで実行されます。メッセージは、ネットワークを通して渡されます。リモートの ActiveX オブジェクトは、インプロセス サーバーとしても、またアウトオブプロセス サーバーとしてもビルドすることができます。「DCOM の使用」の項で説明する手順を実行すると、ローカルで実行する ActiveX オブジェクトを開発し、配置の準備ができたときに、これをリモート オブジェクトに変換することができます。

リモートの ActiveX オブジェクトをプロセス内 サーバーとしてビルドした場合、クライアントのプロセス領域では実行されず、プロキシ クライアントによりリモート マシンに読み込まれます。これにより、プロセス内 サーバーが読み込まれたままになり、その後のクライアントのために再読み込みされることはないため、性能が向上します。

リモートの ActiveX オブジェクトは、COM、Distributed Component Object Model (DCOM) の拡張機能によりサポートされています。DCOM は、Windows の全バージョンでサポートされているわけではありません。詳細については、「チェックリスト - プロジェクトを始める前に」を参照してください。

これら 3 種類の ActiveX オブジェクトのソース コードは、同じですが、ビルド方法と登録方法が異なります。次の 2 つの図は、インプロセス オブジェクト、アウトオブプロセス オブジェクト、リモート オブジェクトを比較したものです。

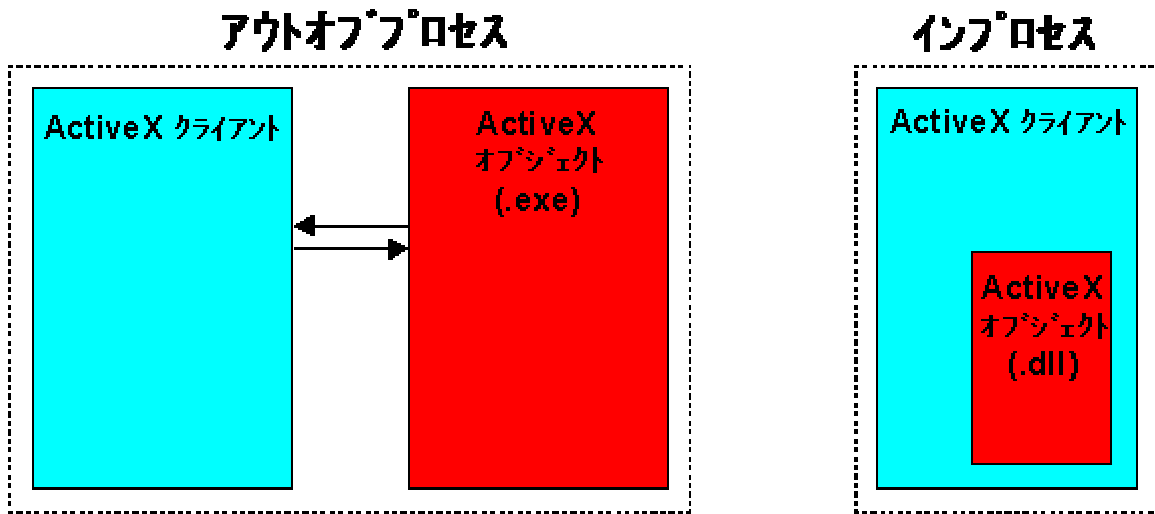


図 4-3アウトオブプロセス ActiveX オブジェクトとインプロセス ActiveX オブジェクト

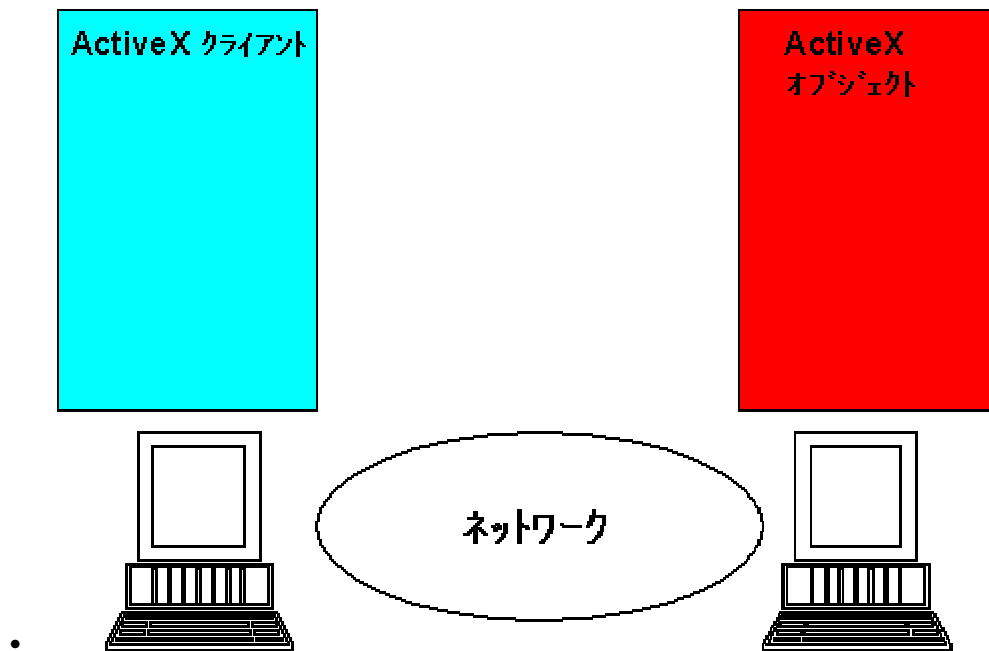


図 4-4: リモート ActiveX オブジェクト

次の項では、Object COBOL を使用して、固有の OLE オートメーション サーバーを作成、登録、デバッグする方法について説明します。

- OLE オートメーションのクラスの作成  
クラスを作成し、メソッドとプロパティを追加する方法を説明します。
- OLE オートメーション サーバーの登録  
サーバーを登録するための技法をいくつか説明します。
- 終了オプションの設定

OLE がサーバーを終了できる条件を指定します。

- ActiveX オブジェクトのデバッグ

サーバー コードをデバッグする方法について説明します。

- タイプ ライブラリの更新

サーバー クラスの変更を ActiveX オブジェクトのタイプ ライブラリに反映する方法について説明します。

- スレッド オプションの設定

マルチスレッド ActiveX オブジェクトを実行するために COM スレッドのデフォルト値を無効にする方法を説明します。マルチスレッド サーバー クラスを作成する場合にだけ、この項を参照してください。

### 4.3.1 OLE オートメーションのクラスの作成

Object COBOL の OLE オートメーション サーバーは、Object COBOL のクラスです。このクラスのインスタンスは、ActiveX オブジェクトです。ここでは、クラスの作成方法を 3 段階に分割して、説明します。

1. スケルトン サーバー クラスの生成
2. ActiveX オブジェクトへのメソッドの追加
3. ActiveX オブジェクトへのプロパティの追加

#### スケルトン サーバー クラスの生成

NetExpress のクラス ウィザードは、OLE オートメーション クラス、起動プログラム、およびレジストリ エントリのアウトラインを生成します。ウィザードを実行する前に、次の事項を決める必要があります。

- クラスの名前とファイル名
- サーバーをインプロセスまたはアウトオブプロセスのどちらで実行するか。
- サーバーをローカルまたはリモートのどちらで実行するか。

サーバーをローカル サーバーとして作成し、完成した時点でリモート サーバーに変更することができます。ローカル サーバーとリモート サーバーの違いは、Windows への登録方法だけです。

- タイプ ライブラリを生成するかどうか。

タイプ ライブラリは、クラスのクライアントにとって役立つ情報を提供します。

デュアル インターフェイスをサポートするライブラリを作成する場合 (デフォルト)、クライアントはタイプ ライブラリに早い段階で結合することができるので、性能が大幅に向上します。タイプ ライブラリには、



最新のクラスを反映する必要があります。「タイプ ライブラリの更新」を参照してください。また、デュアル インターフェイス クライアントはタイプ ライブラリを通じて、早い段階でクラスに結合されるため、タイプ ライブラリを変更した場合は、これらのインターフェイスを再リンクする必要があります。タイプ ライブラリとクライアントに最新の変更が反映されていない場合、プログラムは、実行時にパラメータの不一致が原因で破損する可能性があります。

ウィザードを実行するには、次の手順にしたがいます。

- [ファイル] メニューで [新規作成] をクリックして、「新規作成」ダイアログ ボックスの [クラス] を選択します。

その結果、クラス ウィザードが起動されます。その後は、プロンプトにしたがいます。

ウィザードの最後のダイアログ ボックスで [完了] ボタンをクリックすると、クラス ウィザードにより、次のファイルが生成または更新され、プロジェクトに追加されます。

- Object COBOL クラス *filename.cbl*
- クラスの Windows レジストリ エントリを含む *filename.reg* ファイル (オプション)
- クラスのトリガを含む *filenameTrigger.cbl* プログラム (オプション)

タイプ ライブラリを要求した場合、次のファイルも作成されます。

- クラスの詳細を含む *filename\_if.idl* ファイル
- タイプ ライブラリの詳細を含む *filename.idl* ファイル
- *filename.rc* ファイル。サーバーが *.rc* ファイルをすでに持つ既存の *.dll* ファイルまたは *.exe* ファイルの一部とならない限り、クラスのタイプ ライブラリを含みます。

クラス ウィザードでは、クラスをインプロセス サーバーとしてビルドする場合、*.dll* ファイルが作成されます。アウトオブプロセス サーバーとしてビルドする場合、*.exe* ファイルが作成されます。

---

注記: インプロセス サーバーを作成する場合、クラス ウィザードは、*dllserver.obj* から *.dll* ファイルにリンクさせる指令を NetExpress プロジェクトに追加します。*Dllserver.obj* は、NetExpress リンク ライブラリに含まれています (これらのファイルは、デフォルトでは、¥netexpress¥base¥lib にインストールされています)。

アウトオブプロセス サーバーからインプロセス サーバーに変換するには、*.dll* 用「ビルド設定」ダイアログ ボックスの [リンク] タブで、クラスを *.dll* ファイルとしてリビルドし、起動プログラムを省略します。また、*dllserver.obj* を「CBLLINK 指令の追加」フィールドに追加します。

1. 「プロジェクト」ウィンドウの左側のペインで OLE オートメーション サーバー (起動プログラムは含ま

ない) のファイルを選択し、右クリックして、コンテキスト メニューから [選択したファイルをパッケージ化]、[ダイナミック リンク ライブラリ] を選択します。

2. .dll ファイルの名前を「新規ダイナミック リンク ライブラリを定義」ダイアログ ボックスに入力し、[OK] ボタンをクリックします。
3. 「プロジェクト」ウィンドウで .dll ファイルを右クリックし、コンテキスト メニューで [ビルド設定] をクリックします。
4. [リンク] タブをクリックし、次のように設定します。
  - [共有] ランタイム ライブラリを選択し、「動的」チェック ボックスをチェックします。
  - [カテゴリ] ドロップダウン メニューから [詳細] を選択し、「OBJ とリンク」フィールドで、次のとおり入力します。

```
dllserver.obj
```

[閉じる] ボタンをクリックします。


5. プロジェクトをリビルドして、.dll ファイルを作成します。
6. [ツール] メニューで [OLE レジストリ エントリ ジェネレータ] をクリックし、「サーバーの詳細」の下にある「インプロセス DLL」オプション ボタンを選択して、サーバーの新しいレジストリ エントリを作成します。

---

## ActiveX オブジェクトへのメソッドの追加

OLE オートメーション サーバー クラスに追加するすべてのインスタンス メソッドは、ActiveX オブジェクトのメソッドとして公開されます。ただし、メソッド名が "set" または "get" で始まる場合は除きます（「ActiveX オブジェクトへのプロパティの追加」を参照してください）。メソッドの連絡節で、メソッドと受け渡しするパラメータを宣言してください。「OLE データ型」の章で説明するとおり、Object COBOL ランタイム システムは、COBOL データ型と OLE オートメーション データ型の間で変換を行います。

新しいメソッドを追加するには、次の手順にしたがいます。

- ツールバーの  ボタンをクリックして、メソッド ウィザードのプロンプトにしたがってください。

メソッドを追加または削除した場合には、オブジェクトのタイプ ライブラリを更新する必要があります。メソッド ウィザードを使用すると、これは自動的に行われます。詳細については、「タイプ ライブラリの操作」を参照してください。

---

## 注記

COM は、クラス オブジェクトとクラス メソッドをサポートしていません。OLE オートメーション サーバー用に作成するクラス メソッドは無視されます。ただし、クラス オブジェクトに作成される新しいインスタスを通知する必要がある場合には、"new" という名前のメソッドを作成してください。このメソッドに通知コードを追加します。OLE Object COBOL 以外のクラスと同様に、実際には、「新しい」インスタスを作成するために、この "new" メソッドにコードを入れる必要はありません。


---

## ActiveX オブジェクトへのプロパティの追加

"set" と "get" メソッドを書き込むことによって、ActiveX オブジェクトに OLE オートメーションのプロパティを定義します。クライアントからの OLE プロパティ設定操作または取得操作は、"set" および "get" メソッドにマップされます。プロパティを設定するには、メソッドに "setpropertyname" という名前を付けます。その場合、propertyname はプロパティの名前を指します。プロパティを取得するには、メソッドに "getpropertyname" という名前を付けます。この場合、propertyname はプロパティの名前です。

Set メソッドには、USING パラメータを 1 つ記述します。プロパティ設定は、このパラメータから開始されます。一方、Get メソッドには、プロパティの値を返すための RETURNING パラメータを記述します。「OLEデータ型」の説明のように、Object COBOL ランタイム システムは、COBOL データ型と OLE オートメーション データ型との間でデータ型の変換を行います。

新しいメソッドを追加するには、次の手順にしたがいます。

- ツールバーの  ボタンをクリックし、プロンプトにしたがいます。

メソッドを追加または削除する場合、オブジェクトのタイプ ライブラリを更新する必要があります。メソッド ウィザードを使用すると、これは自動的に行われます。詳細については、「タイプ ライブラリの操作」を参照してください。

次の例では、OLE オートメーション クライアントが LoanTermYears というプロパティを設定するためのメソッドと、このクライアントが YearPayment というプロパティを取得するための別のメソッドを示します。

```
method-id. "SetLoanTermYears".  
  
linkage section.  
  
01 NewLoanTermYears pic 99.  
  
procedure division using NewLoanTermYears.
```

```

    move NewLoanTermYears to LoanTermYears

    exit method.

end method "SetLoanTermYears".

method-id. "GetYearPayment".

linkage section.

01 CurrentYearPayment pic $9(7).99.

procedure division returning CurrentYearPayment.

    move YearPayment to CurrentYearPayment

    exit method.

end method "GetYearPayment".

```

### 4.3.2 OLE オートメーション サーバーの登録

OLE オートメーション サーバーは、あらかじめ Windows システムに登録しておかないと、クライアントが、メッセージを送信したり、ActiveX オブジェクトを作成するために使用したりすることができません。Object COBOL オートメーション サーバーの登録方法は、次の 4 種類です。

- サーバーを実行します。

すべての Object COBOL オートメーション サーバーは、実行時に自動的に登録されます。これは、スタンドアロンで実行できるアウトオブプロセス サーバーにしか適用できません。レジストリに入力される情報は、このメソッドを使用する場合に必要となります。

#### *Windows NT*

登録は、オペレーティング システムが再起動されるまで有効です。

#### *Windows 95*

登録は、永続的に有効です。

- サーバーの .dll に対して Microsoft ユーティリティ regsvr32 を実行します。

これにより、サーバーを永続的に登録します。この場合、サーバー .dll に追加コードが必要となります。この追加コードは、クラス ウィザードでは、自動的に作成されます。詳細については、「自己登録の有効化」を参照してください。

- アウトオブプロセス サーバー (.exeファイル) は、実行時に、永続的に自己登録することができます。詳細

については、「自己登録の有効化」を参照してください。

- Windows システム レジストリにサーバーの詳細を入力します。

この場合、システムにサーバーを永続的に登録します。

最初の方法は、サーバーの開発時とデバッグ時にたいへん役立ちます。クライアントがサーバーの起動要求を送信するタイミングがわからない場合には、その他の方法を使用します。サーバーがレジストリに登録されると、Windows は、クライアントがサーバーの新しいインスタンスを要求したときにサーバーを自動的に起動します。

一部の開発環境（たとえば、Visual Basic など）では、作成した新しい ActiveX オブジェクトが自動的に開発マシンに登録されます。登録過程は見えませんが、バックグラウンドでこの処理が行われています。オブジェクトを配置するマシンでそれぞれ登録する必要があります。

OLE オートメーション サーバーを作成するために NetExpress クラス ウィザードを使用すると、オブジェクトの登録に必要なすべての情報を含む .reg ファイルが生成されます。これは ASCII テキスト ファイルで、Windows レジストリに挿入されるエントリを含みます。このファイルの形式は、「レジストリ エントリの編集」を参照してください。

クライアントは、この情報によりオブジェクトの ProgID からオブジェクトに固有な ID (Clsid) を取得できます。Object COBOL クラスの ProgID は、デフォルトではクラスの名前です。ただし、メソッド "queryClassInfo" を編集すると、これを変更することができます（次項を参照）。クライアントは、Clsid を使用して、OLE オートメーション サーバーを起動するために必要な情報を提供する別のエントリをレジストリで検索できます。

Windows レジストリに .reg ファイルの情報を入力する方法には 2 通りあります。

- NetExpress プロジェクト メニューの .reg ファイルを右クリックし、コンテキスト メニューで [登録] を選択します。

または、

- コマンド プロンプトから次のように入力します。

```
regedit filename.reg
```

## 自己登録の有効化

正しく登録するために必要な情報をすべて含む OLE オートメーション サーバーをビルドすることができます。情報は、各 OLE オートメーション クラス の 2 つのメソッドに格納されます。

- "queryClassInfo"

クラスに関する情報を返します。

- "queryLibraryInfo"

クラス タイプ ライブラリに関する情報を返します。

これらのメソッドは、OLE オートメーション クラスを作成するときに、クラス ウィザードにより自動的に生成されます ("queryLibraryInfo" は、同時にタイプ ライブラリを生成する場合にだけ生成されます)。これらのメソッドで返される情報の一部を変更すると、クラスの自己登録の詳細を変更できます。

"queryClassInfo" メソッドは、次のように記述されます。

```
method-id. queryClassInfo.  
  
linkage section.  
  
01 theProgId          pic x(256).  
  
01 theClassId        pic x(39).  
  
01 theInterfaceId    pic x(39).  
  
01 theVersion        pic x(4) comp-5.  
  
procedure division using by reference theProgId  
  
                        by reference theClassId  
  
                        by reference theInterfaceId  
  
                        by reference theVersion.  
  
move z"{clsid}" to theClassId  
  
move z"{clsid}" to theInterfaceId  
  
exit method.  
  
end method queryClassInfo.
```

この場合、*clsid* は 128 ビットの *clsid* を表す文字列です。たとえば、{F08FCA68-286C-11D2-831F-B01A09C10000} のようになります。*clsid* 値は変更しないでください。*clsid*は、Windows API により生成されるので、すべてのクラスに固有の *clsid* が指定されています。*clsid* は、クラスとそのインターフェイスの登録の一部としてさまざまな場所で使用されるので、値が一致する必要があります。一致しない場合、予測しない結果が発生します。新しい ProgID を指定した、ヌル値で終了する文字列をデータ項目 *theProgId* に移動すると、クラスの ProgID を変更することができません。バージョン番号は、現在、COBOL ランタイム システムでは無視されています。

"queryLibraryInfo" メソッドは、次のように記述されます。

```
method-id. queryLibraryInfo.
```

```

linkage section.

01 theLibraryName      pic x(512).

01 theMajorVersion    pic x(4) comp-5.

01 theMinorVersion    pic x(4) comp-5.

01 theLibraryId       pic x(39).

01 theLocale          pic x(4) comp-5.

procedure division using by reference theLibraryName

                        by reference theMajorVersion

                        by reference theMinorVersion

                        by reference theLibraryId

                        by reference theLocale. *> 現在は無効です。

move 1 to theMajorVersion

move 0 to theMinorVersion

move z"{clsid}" to theLibraryId

exit method.

end method queryLibraryInfo.

```

"queryClassInfo" については、ライブラリ ID の値を変更しないでください。ただし、theTypeLibrary の値を次のどれかに設定すると、クラスのタイプ ライブラリの場所を指定することができます。

文字列を指定 タイプ ライブラリがクラスの .dll ファイルまたは .exe ファイルにリソースとして埋め込まれます。クラス  
しない スをデフォルトのクラス ウィザードで作成した場合、タイプ ライブラリはこの場所に格納されます。

*n* タイプライブラリがクラスの .dll ファイルまたは .exe ファイルのリソースであることを示す数字を指定  
します。

*path* タイプ ライブラリへの完全パスとファイル名を指定します。

theMajorVersion および theMinorVersion の値を設定することにより、タイプ ライブラリのメジャー バージョン番  
号とマイナー バージョン番号をオプションで設定することもできます。

インプロセス サーバーとアウトオブプロセス サーバーの登録手順を次に示します。

- アウトオブプロセス サーバー
- インプロセス サーバー

## アウトオブプロセス サーバー

アウトオブプロセス サーバーでは、自己登録と自己登録解除を行うことをお勧めします。自己登録は、通常、コマンド行スイッチで制御されています。たとえば、`myserver.exe` を登録するには、エンドユーザーは次のコマンドを入力します。

```
myserver /Register
```

サーバーの登録を解除するには、エンドユーザーは次のコマンドを入力します。

```
myserver /Unregister
```

`olesup` クラスには、2 つのメソッド、`"registerServer"` と `"unRegisterServer"` があり、この操作を簡単に行うことができます。登録と登録解除のためにコマンド行でスイッチを検出するサーバーの起動プログラムに、コードを追加します。

起動プログラムが登録スイッチを検出したときに、次のコードを実行します。

```
invoke olesup "registerServer" using theClass commandline
    returning error-code
```

`olesup` の `"registerServer"` メソッドは、`"queryClassInfo"` メソッドと `"queryLibraryInfo"` メソッドを使用して、`theClass` にクラスとタイプ ライブラリの情報を問い合わせます。

起動プログラムが登録解除スイッチを検出したときに、次のコードを実行します。

```
invoke olesup "unregisterServer" using theClass
    returning error-code
```

| データ項目              | データ型             | 説明   |
|--------------------|------------------|--|
| <i>theClass</i>    | OBJECT REFERENCE | 登録するクラス  |
| <i>commandline</i> | PIC X(256)       | サーバーを起動するコマンド行。サーバーへのパスを含む必要があります。               |
| <i>error-code</i>  | PIC X(4) COMP-5  | 正しく登録できた場合は 0 を返します。登録に失敗した場合は OLE エラー コードを返します。 |



## インプロセス サーバー

Microsoft regsvr32 ユーティリティでは、コマンド行からインプロセス サーバー（.dll ファイルとしてビルドされたサーバー）を登録または登録解除することができます。サーバーを登録するには、次のように記述します。

```
regsvr32 server.dll
```

サーバーを登録解除するには、次のように記述します。

```
regsvr32 -u server.dll
```

クラス ライブラリ ウィザードにより自動的に生成された "queryClassInfo" メソッドと "queryLibraryInfo" メソッドに加えて、.dll ファイルには、regsvr32 ユーティリティによる登録が機能するように DllOleLoadClasses という名前のエントリ ポイントを含める必要があります。クラス ライブラリ ウィザードで、クラスの起動プログラムを生成するオプションを選択すると、DllOleLoadClasses が自動生成されます。

次のコードは、DllOleLoadClasses の例を示します。

\*> OLE サーバー トリガ（インプロセス サーバーの場合のみ）

\*>警告： このファイルには、これ以上エントリ ポイントを追加しないでください。

```
$set case
```

```
linkage section.
```

```
01 loadReason    pic x(4) comp-5.
```

```
procedure division.
```

```
entry "DllOleLoadClasses" using by value loadReason.
```

\*> OCWIZARD - クラスを起動します。

\*> クラスを呼び出すと、OLE クライアントが使用できるクラスとして登録されます。

```
    call "myserver"
```

\*> OCWIZARD - クラスを終了します。

```
    exit program.
```

.

上記の例は、ファイル名 myserver で呼び出された 1 つのクラスだけを登録します。

DllOleLoadClasses に固有のコードを追加することができます。パラメータ loadReason には、COBOL ランタイム シ

システムがこのエントリ ポイントを呼び出した理由が返されます。

- 1 サーバーを読み込み中です。
- 2 クラスを登録中です。
- 3 サーバーを登録解除中です。

### 4.3.3 終了オプションの設定

デフォルトでは、アウトオブプロセス サーバーは、最後のクライアントがログアウトしたときに自動的に終了します。ただし、このような終了方法以外の方法が必要になることがあります。たとえば、サーバーがデスクトップのウィンドウを表示している場合などです。この設定は、クラス OLESup で "setOLETerminateOption" メソッドを使用すると、変更できます。

```
invoke olesup "setOLETerminateOption" using by value option
```

この場合、*option* は、次のどちらかを指します。

- 0 ウィンドウが表示されていない場合だけに、サーバーを終了することができます。
- 1 コンソール ウィンドウ以外のウィンドウが表示されていない場合だけに、サーバーを終了できます。

(Object COBOL の GUI アプリケーションで DISPLAY 文を使用している場合、このアプリケーションによりコンソール ウィンドウが開かれます)。

- 2 開いているウィンドウに関係なく、サーバーを終了できます (これはデフォルトの動作です)。

### 4.3.4 ActiveX オブジェクトのデバッグ

NetExpress を使用して ActiveX オブジェクトをデバッグします。インプロセス ActiveX オブジェクトとアウトオブプロセス ActiveX オブジェクトでは、手順がやや異なります。

#### アウトオブプロセスのデバッグ

OLE クライアントとサーバーが別のプロセスとして実行されている場合、サーバーをデバッグするには、NetExpress をコピーする必要があります。これには、2 つの方法があります。

- NetExpress デバッガを通してサーバー トリガを起動する。
- "CBL\_DEBUGBREAK" 呼び出しを追加する。

トリガを起動し、デバッグするには、次の手順にしたがいます。

1. NetExpress を起動し、サーバー アプリケーションの起動プログラムのアニメートを開始します。

2. 次のどちらかの文が表示されるまで、起動プログラムをステップ実行します。

```
invoke olesup "becomeServer"
```

または

```
invoke anEventManager "run"
```

3. この文をステップ実行する場合、NetExpress アニメータは応答を停止します。

サーバーは、OLE クライアントからメッセージを受け取るためにイベント ループで待機します。この段階で、クライアント アプリケーションの実行とデバッグが可能になります。クライアントがサーバーにメッセージを送信すると、呼び出された OLE サーバー メソッドの最初の文にある実行ポイントで NetExpress アニメータが起動します。

この段階で、サーバー メソッドをデバッグすることができます。EXIT METHOD 文をステップ実行すると、実行結果はイベント ループに戻り、サーバーが別のメッセージを受け取るまで、NetExpress アニメータが中断されます。

呼び出しを追加し、デバッグするには、次の手順にしたがいます。

1. ソースコードでデバッグを開始する場所を決定します。
2. 次の文を追加します。

```
CALL "CBL_DEBUGBREAK"
```
3. サーバーをリビルドします。
4. クライアント プログラムを実行します。

実行時に、サーバーで "CBL\_DEBUGBREAK" の呼び出しに到達すると、アニメートを開始するかどうかをたずねるメッセージ ボックスが表示されます。

クライアントを通常どおり実行すると、デバッグされるのはサーバーだけになります。また、デバッガでもクライアントを起動すると、クライアントからサーバーへのメッセージ送信をステップ実行する場合に、2 つのデバッガの間で制御が切り替えられます。クライアントが Object COBOL で作成されている場合、クライアントをデバッグするには、NetExpress の 2 つ目のインスタンスを起動する必要があります。クライアントが別の言語で作成されている場合には、その言語のベンダーが提供するツールを使用してクライアントをデバッグします。

## インプロセス サーバー

インプロセス サーバーは、クライアントと同じアドレス空間に読み込まれます。クライアントが Object COBOL で作成されている場合、NetExpress でデバッグだけを行います。メッセージをサーバーに送信する文をステップ実行すると、NetExpress アニメータにサーバー コードが表示されます。

クライアントが別の言語で作成されている場合には、クラスのメソッド "new" を Object COBOL の OLE オートメーション サーバーに追加します。次の文を "new" メソッドに含めます。

```
call "CBL_DEBUGBREAK"
```

クライアントがサーバーを起動すると、NetExpress が自動的に起動され、サーバー コードのデバッグが可能になります。

### 4.3.5 タイプ ライブラリの操作

NetExpress クラス ウィザードで生成されるタイプ ライブラリには、次の情報が含まれます。

- クラスの各メソッドの名前、ディスパッチ ID、パラメータのデータ型
- 各プロパティの名前とデータ型

次の項では、サーバー プログラムの最新の変更をタイプ ライブラリに反映する方法について説明します。

- プロパティとメソッドの追加
- プロパティとメソッドの削除
- パラメータのデータ型の変更

NetExpress Ocreg ユーティリティを使って、既存の OLE オートメーション クラスのためにいつでもタイプ ライブラリを生成することもできます。NetExpress [ヘルプ] メニューの [ヘルプ トピック] をクリックし、詳細は、[インデックス] タブの「タイプ ライブラリ」を参照してください。

#### 4.3.5.1 プロパティとメソッドの追加

すべてのプロパティとメソッドを追加するには、メソッド ウィザードを使用する必要があります。メソッド ウィザードを起動するたびに、タイプ ライブラリが新しいインターフェイス情報に自動更新されます。

#### 4.3.5.2 プロパティとメソッドの削除

タイプ ライブラリからメソッドまたはプロパティを削除するには、次の手順にしたがいます。

1. クラスのソース コードからメソッドを削除します。
2. テキスト ウィンドウで *filename\_if.idl* ファイルを開きます (NetExpress の「プロジェクト」ウィンドウで名前をダブルクリックします)。
3. *filename\_if.idl* でメソッドを検索します。プロパティ取得操作とプロパティ設定操作は、それぞれ別のメソッドとして記述されています。すべてのメソッドは、次のようにコメントの間に記述されています。

```
//OCXWIZARD - メソッドを起動します。
```

...

//OCXWIZARD - メソッドを終了します。

メソッドは次のように記述します。

```
[id(DISPID_CLASSNAME_METHODNAME)] HRESULT MethodName (  
    parameter_descriptions);
```

プロパティは、次のように記述します。

```
[id(DISPID_CLASSNAME_SETPROPERTYNAME), propput] HRESULT PropertyName (  
    parameter_descriptions);
```

```
[id(DISPID_CLASSNAME_GETPROPERTYNAME), propget] HRESULT PropertyName  
(parameter_descriptions);
```

パラメータの内容は、次のとおりです。

|                               |                               |
|-------------------------------|-------------------------------|
| <i>CLASSNAME</i>              | クラスの名前                        |
| <i>METHODNAME</i>             | メソッドの名前                       |
| <i>property_name</i>          | プロパティの名前                      |
| <i>parameter_descriptions</i> | メソッドで使用されるパラメータを記述する行 (1 行以上) |

4. メソッドの ID (DISPID\_CLASSNAME\_METHODNAME) を書き留め、メソッドを削除します。
5. テキスト ウィンドウで *filename\_if.h* を開き、メソッドの ID を検索して、削除します。
6. クラス、タイプ ライブラリ (.tlbファイル) および、クラスのリソース ファイル (*filename.res*) を、個々にリビルドするか、または、NetExpress プロジェクトで包括的にリビルドします。
7. タイプ ライブラリを使用するクライアントを再リンクするか、参照し直します。

#### 4.3.5.3 パラメータのデータ型の変更

メソッド、またはプロパティの設定や取得によりパラメータのデータ型を変更する手順を次に示します。

1) データ型をソースで変更します。 2) *filename\_if.idl* ファイルでメソッドを検索し、新しい COBOL データ型と一致するように IDL 型を変更します。(この変更を行うには、テスト用クラスを作成し、このクラスにメソッドを追加して、ウィザードの使用結果を確認することをお勧めします) 3) .TLB とサーバーを再コンパイルします。 4) タイプ ライブラリを使用するクライアントを再リンクするか、参照し直します。

1. クラスのソース コードでメソッドを検索し、COBOL データ型のパラメータを変更します。

プロパティのデータ型を変更する場合は、データ型は `get` メソッドや `set` メソッドと一致する必要があります。

2. テキスト ウィンドウで `filename_if.idl` ファイルを開きます (NetExpress の「プロジェクト」ウィンドウで名前をダブルクリックします)。
3. `filename_if.idl` でメソッドを検索します。プロパティの取得と設定も個別のメソッドとして格納されています。メソッドを識別する方法については、前の項を参照してください。
4. COBOL ソース コードの変更を反映するようにメソッドの IDL データ型を変更します。

IDL データ型と COBOL データ型を一致させる方法がわからない場合には、クラス ウィザードを使用してダミーの OLE オートメーション サーバーを作成します。次にメソッド ウィザードに必要なデータ型のメソッドを追加します。その結果、ダミー クラスのタイプ ライブラリから、使用中のタイプ ライブラリにパラメータ情報をコピーすることができます。

5. クラス、タイプ ライブラリ (`.tlb`ファイル)、およびクラスのリソース ファイル (`filename.res`) を、個々にリビルドするか、または、NetExpress プロジェクトで包括的にリビルドします。
6. タイプ ライブラリを使用するクライアントを再リンクするか、参照し直します。

#### 4.3.6 スレッド オプションの設定

マルチスレッド サーバーをビルドし、実行する場合、この項の説明だけで十分対応できます。デフォルトでは、Object COBOL サーバーは、シングル スレッド サーバーとしてビルドされます。シングル スレッドの COBOL プログラムは、シングル スレッド用の COM スレッド オプション以外を実行することはできません。

NetExpress でマルチスレッド プログラムをビルドする方法については、NetExpress の [ヘルプ] メニューで [ヘルプ トピック] をクリックしてから、[目次] タブで [プログラミング]、[マルチスレッド プログラミング] の順に選択します。

COM には、次のようなサーバー専用のスレッド オプションがあります。

- `single-threaded`
- `apartment` - 各オブジェクトはシングル スレッドで実行されます。
- `free` - 完全なマルチスレッド

オブジェクトは、どのスレッドのメソッドからも呼び出すことができるため、スレッドにローカルなデータをもつことはできません。

- `both` - アpartmentまたはマルチスレッド(インプロセス サーバーのみ)

COM は、アウトオブプロセス サーバーとインプロセス サーバーで異なるスレッドを管理します。

#### 4.3.6.1 アウトオブプロセス サーバー

アウトオブプロセス サーバーは、シングル スレッド用にビルドされるとシングル スレッドとして実行され (COBOL プログラムのデフォルト)、マルチスレッド用にビルドされるとフリー スレッドとして実行されます。

マルチスレッドのアウトオブプロセス サーバーをアパートメント スレッドとして実行するように設定するには、クラスを実行する前に、サーバーの起動コードに次の文を含めてください。

```
invoke olesup "singleThread" returning aBoolean
```

ここで aBoolean は、PIC X(4) COMP-5で、正常な場合は 1 にセットされます。

#### 4.3.6.2 インプロセス サーバー

デフォルトでは、COM はビルド オプションに関係なく、常にインプロセス サーバーをシングル スレッド サーバーとして実行します。インプロセス サーバーのスレッド オプションを変更する唯一の方法は、レジストリ エントリを編集することです。

1. サーバーのレジストリ エントリで InProcServer32 キーを検索します。

このキーへの完全なレジストリ パスは、[HKEY\_CLASSES\_ROOT¥{clsid}¥InProcServer32] です。この場合、clsid は、サーバーの CLSID です。

2. このキーの次にある ThreadingModel という名前付きの値を次のどれかに変更します。

- Single (ThreadingModel が設定されない場合のデフォルト値)
- Apartment
- Free
- Both

---

#### 注記

NetExpress により生成されたレジストリ ファイルには、名前付きの値 ThreadingModel がありません。regsvr32 を実行することにより生成されたレジストリ エントリにも、値 ThreadingModel は含まれません。

NetExpress により生成されたレジストリ ファイルにこのキーを追加するには、レジストリ ファイルを手動で編集するか、サーバーが登録されたときに Windows API を使用してレジストリを編集するコードを作成します。このコードは、クラス ウィザードにより生成されたインプロセス起動プログラムのエントリ ポイント DllLoadClasses に追加することができます。詳細については、「インプロセス サーバー」を参照してください。

---

## 4.4 DCOM の使用

DCOM (Distributed Component Object Model) は、ネットワークのさまざまなマシンに、ActiveX クライアントと ActiveX オブジェクトを分散させることができます。クライアント マシンとサーバー マシンの正しいレジストリ情報を提供することにより、Object COBOL ActiveX オブジェクトをネットワーク全体で実行することができます。サーバーが存在するリモート マシンは、クライアント マシンのレジストリ エントリにより検索できます。

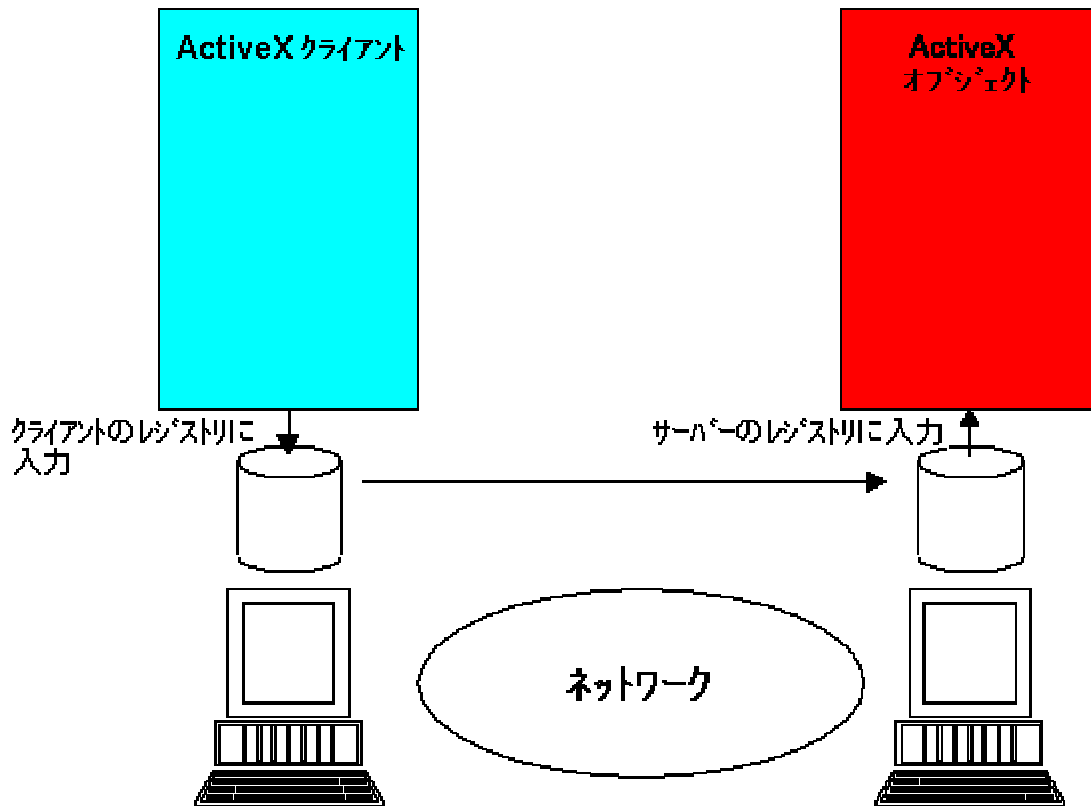


図 4-5 リモートにある ActiveX オブジェクトの検索

ActiveX オブジェクトに対しては、2 つの .reg ファイルを生成する必要があります。1 つは、クライアント マシンで登録するファイルで、もう 1 つは、リモートサーバー マシンで登録するファイルです。OLE オートメーション サーバー クラスの作成時にこのクラスをリモートとして指定すると、これらのファイルは NetExpress クラス ウィザードにより自動的に作成されます。ただし、ActiveX オブジェクトをローカル サーバーとして開発し、完成時にリモート サーバーとして再配置する方法が主流となっています。

レジストリ ファイルを作成する手順は、次のとおりです。

1. NetExpress の [ツール] メニューで [OLE レジストリ ファイル ジェネレータ] をクリックします。
2. 「サーバー情報」グループ ボックスで、次のように操作します。
  - クラスが .dll プログラムとしてビルドされている場合、「プロセス内」をクリックします。また、クラスが .exe プログラムとしてビルドされている場合、「簡易」をクリックします。



- 「分散」チェック ボックスを選択します。
- サーバーのマシン名がわかっている場合には、「リモート サーバー」フィールドに入力します。

3. 「クラス」グループ ボックスでは、次の操作を行います。

- 「プログラムID」フィールドにクラス ID を入力します。
- 「コメント」フィールドに説明を入力します。
- インプロセス サーバーの場合には、「DLL名」フィールドに .dll ファイル名を入力します。
- 簡易サーバーの場合は、「コマンド行」フィールドに .exe ファイル名を入力します。

リモート マシンの %windows%system ディレクトリにサーバーをインストールする場合以外は、ファイル名の入力時に完全パスも指定します。

サーバーにアクセスするすべてのクライアント マシンでこのサーバーを登録するには、クライアントのレジストリ ファイルを使用します。サーバー マシンでサーバーを登録するには、サーバーのレジストリ ファイルを使用します。また、サーバー マシンには、NetExpress または Application Server をインストールする必要があります。

#### 4.4.1 DCOM とセキュリティ

Windows NT のセキュリティ機能により、分散サーバーの起動、実行、アクセスの過程で問題が生じることがあります。この項では、DCOM と Windows NT を使用するときを確認する必要がある主な点を列挙します。

- クライアントがサーバーを使用できるように、サーバーに起動権限とアクセス権の両方を設定する必要があります。

DCOM では、これらの規則が強制的に適用されます。Microsoft の DCOMCNFG ユーティリティを使用すると、各サーバーの値を設定することができます。すべてが正常に機能していることを確認するには、起動権限とアクセス権をカスタマイズし、各リストに "Everyone" を追加して、サーバーと同じドメインにあるクライアントを使用します。その後で、セキュリティを調整します。

- サーバーを実行するためのユーザー ID も設定する必要があります。この場合も、Microsoft の DCOMCNFG ユーティリティを使用してください。

次のどちらかを選択することができます。

|         |  |
|---------|--|
| 対話型ユーザー | 現在サーバーにログインしているユーザー アカウントでサーバーを起動します。マルチユーザー サーバーを使用する場合は、サーバーを 1 つ起動するだけでかまいません。サーバーは、画面への自動アクセス権を持っています。 |
|---------|--|

|        |  |
|--------|--|
| 起動ユーザー | サーバーは、クライアントアプリケーションが実行されているマシンアカウントで起動されます。マルチユーザーサーバーを使用する場合、サーバーは、新しくログインする各クライアントによって起動されます。 |
| 指定ユーザー | サーバーは、特定のユーザーアカウントで起動されます。マルチユーザーサーバーを使用する場合は、サーバーを1つ起動するだけではありません。                              |

セキュリティの詳細については、Microsoft のマニュアルを参照してください。

## 4.5 レジストリ エントリの編集

NetExpress は、ActiveX オブジェクトを簡単に登録するための .reg ファイルを生成します。この項では、サーバーへのパスを変更する必要がある場合に備え、.reg ファイルの形式に関して説明します。.reg ファイルは、次の形式の ASCII ファイルです。

REGEDIT

[HKEY\_CLASSES\_ROOT¥*classname*]

@ = *description*

[HKEY\_CLASSES\_ROOT¥*classname*¥Clsid]

@ = *uuid*

[HKEY\_CLASSES\_ROOT¥CLSID¥*uuid*]

@ = *description*

[HKEY\_CLASSES\_ROOT¥CLSID¥*uuid*¥ProgID]

@ = *classname*

[HKEY\_CLASSES\_ROOT¥CLSID¥*uuid*¥*servertype*]

@ = *startup*

パラメータの内容は、次のとおりです。

|                    |  |
|--------------------|--|
| <i>classname</i>   | ProgID。クライアントは、この名前で OLE オートメーションサーバーを識別します。ProgID は、通常、基本ファイル名です。 |
| <i>description</i> | サーバーについて説明する簡単なコメント。   |

*uuid* Windows API の `theCreateGUID()` 関数により作成された固有の数字による識別子。UUIDGEN ツールを実行して、これらを作成することもできます (詳細は、後述します)。

*startup* サーバーを実行するためのトリガ。トリガが `.exe` ファイルとしてコンパイルされる場合、トリガ名だけを記述します。`.int` ファイルまたは `.gnt` ファイルとしてコンパイルした場合には、COBOL 実行コマンドを使用する必要があります。

このパラメータには、`.exe` ファイル名、バッチ ファイル名、COBOL 実行コマンドが代入されます。たとえば、次のように記述します。

```
run mytrigger
```

`.exe` ファイルまたはバッチ ファイルの場合、*startup* パラメータの一部として、システム パスを指定するか (PATH 環境変数により設定されます)、明示的なパスを設定する必要があります。

*servertype* OLE サーバーの型。一般的な型は、`LocalServer32` と `InProcServer32` です。

UUIDGEN ツールは、Microsoft Win32 SDK の一部です。UUIDGEN を実行するたびに、固有の 16 進数が生成されます。

Windows レジストリに情報を入力するには、コマンド プロンプトで次のように入力してください。

```
regedit registryfile
```

この場合、*registryfile* は、レジストリ エントリのあるファイルの名前です。

別の Windows システムにオートメーション サーバーをインストールするたびに、オートメーション サーバーを登録する必要があります。毎回、同じレジストリ ファイルを使用することができます。ただし、レジストリ ファイルがマシン固有のパスを含む場合は、それを編集する必要があります。

注記

レジストリ ファイルの形式と構造体は、さまざまです。詳細については、Win 32 SDK のマニュアルを参照してください。

## 4.6 詳細情報

OLE は、大きなテーマなので、このような短い章では詳しく説明することができません。OLE についてより深く理解しておく、役に立つことがあるかもしれません。OLE の詳細については、次の情報源を使用してください。

- Win32 SDK のマニュアルの『OLE プログラマーズ リファレンス』

これには、OLE オートメーションに関する説明が含まれています。Win32 SDK とそのマニュアルは、NetExpress の CD-ROM に収録されています。

- 『*Inside OLE2*』 Kraig Brockschmidt 著 Microsoft Press 発行
- 『*Inside COM*』 Dale Rogerson 著 Microsoft Press 発行

Microsoft の Web サイトにある次のページも参照してください。

- OLE Development

OLE、COM、DCOM の開発者のための最新ニュースと技術記事です。

- COM Security Frequently Asked Questions

OLE オートメーションを使用した分散システムをビルドするのに有効な情報です。

## 第5章 OLE データ型

OLE オートメーションは、OLE クライアントと OLE サーバー間で通信するためのデータ型を定義します。この章では、COBOL データ型を OLE データ型にマップする方法を説明します。

### 5.1 概要

OLE オートメーションは、ある OLE オブジェクトから別の OLE オブジェクトにデータを渡すための独自のデータ型を定義しています。Object COBOL は、次の図で示すように、OLE メッセージの送受信時に、COBOL データ型と OLE データ型の間で自動的に変換を行います。

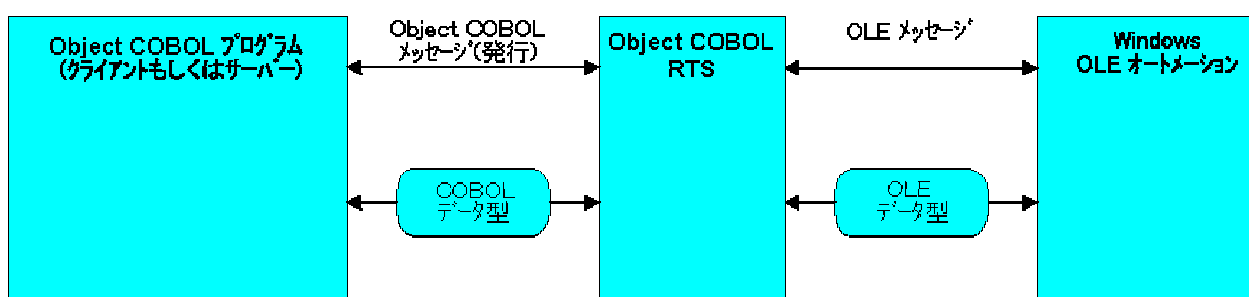


図 5-1 OLE オートメーションによるデータ送信

Object COBOL の OLE オートメーションでは、次のデータ型をサポートします。

- 2 バイトと 4 バイトの整数型
- 4 バイトと 8 バイトの浮動小数点数型
- 文字列型
- OLE オブジェクト型
- OLE VARIANT 型
- OLE SafeArrays 型

OLE SafeArrays 型と OLE Variants 型は、Object COBOL クラスである OLESafeArray と OLEVariant のインスタンスとして Object COBOL に渡される複雑なデータ型です。他のデータ型は直接、同じ COBOL データ型に変換されます。

これから説明する項目は、次のとおりです。

- OLE データ型指定規則

COBOL プログラムと OLE の間でデータを送受信する場合にランタイム システムで行われる変換につい

て説明します。

- オブジェクト リファレンス

オブジェクト リファレンスを COBOL と OLE の間で転送する場合に発生する状況の追加情報について説明します。

- Variant

Variant データ型と、これを操作するための Object COBOL OLEVariant クラスの使用方法を説明します。

- SafeArrays

SafeArrays 型と、これを操作するための Object COBOL OLESafeArray クラスの使用方法を説明します。

## 5.2 OLE データ型指定規則

OLE オブジェクトに COBOL データを送信する場合、適切な OLE データ型が指定されます。同様に、COBOL プログラムが OLE オブジェクトからデータを受け取る場合に、COBOL 型が指定されます。次の表は、OLE オートメーションを通じてデータを Object COBOL プログラムと受け渡しするときに行われる変換を示します。

| OLE データ型    | COBOL データ型  | 説明  |
|-------------|---|---|
| VT_I2       | PIC X(2) COMP-5   | 2 バイトの整数  |
| VT_I4       | PIC X(4) COMP-5   | 4 バイトの整数  |
| VT_DISPATCH | OBJECT REFERENCE -<br>OLEBase のインスタンス (オブジェクト リファレンスが不要になった場合、受信側プログラムで処理します) | OLE オブジェクト ハンドル。「オブジェクト リファレンス」の項を参照してください。     |
| VT_R4       | COMP-1  | 4 バイトの浮動小数点                                     |
| VT_R8       | COMP-2  | 8 バイトの浮動小数点                                     |
| VT_DATE     | COMP-2  | バイナリ形式の日付。詳細については、Microsoft OLE マニュアルを参照してください。 |
| VT_BOOL     | pic x(2) comp-5   | ブール値  |

| OLE データ型   | COBOL データ型   | 説明   |
|--|--|--|
| <p>SafeArray</p> <p>SafeArray の OLE データ型は VT_datatype の VT_ARRAY ORed です。VT_datatype は、SafeArray によりサポートされているデータ型ならどれでも格納できます。</p>       | <p>OBJECT REFERENCE - OLESafeArray のインスタンス</p>   | <p><math>n</math> 次元の固定サイズの配列。<br/>「SafeArrays」の項を参照してください。</p>  |
| <p>Object COBOL メソッドに渡す VT_VARIANT</p>   | <p>OBJECT REFERENCE - OLEVariant のインスタンス</p>   | <p>Variant データ。型情報とデータを含みます。Variant 型データには、この表に記載されている他のデータ型を 1 つ含めることができます。「Variant」の項を参照してください。</p> <p>注記:</p> <p>これは、メソッドの起動時にメソッドに渡される variant だけに適用されます。起動済みのメソッドは、VT_VARIANT を返しません。</p> |
| <p>Variant の内容に基づいて、次のどれかに変換されます。</p> <ul style="list-style-type: none"> <li>• IDispatch</li> <li>• SafeArray</li> <li>• BSTR</li> </ul> | <p>OBJECT REFERENCE - Object COBOL メソッドから返された OLEVariant のインスタンス</p>   | <p>これは、OLE オートメーションを通して起動された Object COBOL メソッドが OLEVariant のインスタンスを返そうとした場合に適用されます。かわりに variant データ型の内容が返されます。</p>   |
| <p>VT_BSTR</p>   | <p>PIC X(<math>n</math>) または CharacterArray のインスタンス</p> <p>ランタイム システムは、使用中の COBOL データ項目が PIC X(<math>n</math>) と OBJECT REFERENCE のどちらで宣言されるかによって、これらの 2 つの型のどちらかを使用して自動的に変換します。</p> | <p>データ長を先頭につけた文字列</p>  |

OLE オートメーションは、これらのさまざまなデータ型を識別するために数字コードを使用します。たとえば、

VARIANT データ型では、型情報を格納しています。これらの数字コードは、すべてコピー ファイル mfole.cpy のレベル 78 のデータ項目として定義されます。このコピー ファイルは、¥netexpress¥base¥source¥ ディレクトリにあります。

### 5.3 オブジェクト リファレンス

Object COBOL プログラムが OLE オブジェクトに渡されるたびに、ランタイム システムは、OLE ハンドルを (VT\_DISPATCH) プロキシ Object COBOL ハンドル (OBJECT REFERENCE) に変換します。プロキシ オブジェクトは、OLEbase のインスタンスです。Object COBOL ハンドルを使用して OLE オブジェクトにメッセージを送信し、パラメータとして別の OLE オブジェクトにハンドルを渡します。

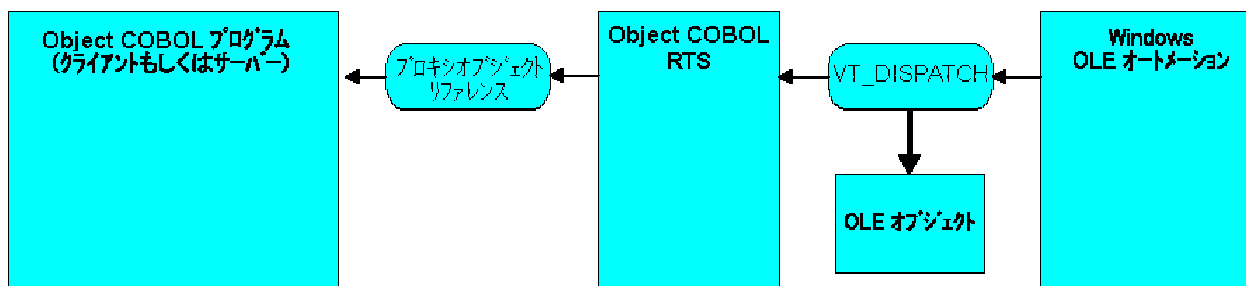


図 5-2 VT\_DISPATCH のプロキシ リファレンスへの変換

オブジェクト ハンドルを別の OLE オブジェクトに送信する場合に、ランタイム システムは次の規則にしたがってオブジェクト ハンドルを変換しようとします。

- オブジェクト ハンドルがプロキシ オブジェクトのハンドルである場合、元の VT\_DISPATCH 値に変換されます。
- オブジェクト ハンドルが CharacterArray のハンドルである場合、VT\_BSTR に変換されます。
- オブジェクト ハンドルが OLEVariant のハンドルである場合、VT\_VARIANT に変換されます。
- オブジェクト ハンドルが SafeArray のハンドルである場合、VT\_ARRAY に変換されます。
- オブジェクト ハンドルが上記以外ののハンドルである場合、「型が不明です。」という例外メッセージが表示されます。

"getClass" メッセージを olesup に送信すると、オブジェクト ハンドルのクラスを検索することができます。返される結果は、クラスのオブジェクト ハンドルです。プログラムの Class-Control の段落で宣言したクラス名と比較することによって、クラスを調べることができます。たとえば、次のように記述します。

```
class-control.
```

```
CharacterArray is class "chararray"
```

```
olesup is class "olesup"
```



```
SafeArray is class "olesafea"
```

```
OLEVariant is class "olevar"
```

```
...
```

```
.
```

```
working-storage section.
```

```
01 anObject          object reference.
```

```
01 aClass            object reference.
```

```
...
```

```
procedure division.
```

```
...
```

```
invoke olesup "getClass" using anObject
```

```
                returning aClass
```

```
if aClass = CharacterArray
```

```
    display "CharacterArray 型"
```

```
else
```

```
    if aClass = olebase
```

```
        display "OLE オブジェクト"
```

```
    else
```

```
        if aClass = SafeArray
```

```
            display "SafeArray 型"
```

```
        else
```

```
            if aClass = OLEVariant
```

```
                display "Variant 型"
```

```
            else
```

```
                display "OLE オートメーションではサポートされていません。"
```

```
end-if  
  
end-if  
  
end-if  
  
end-if  
  
...
```

## 5.4 Variant 型

OLE Variant は、データの型情報とデータの断片をラップする OLE データ型です。OLE プログラミングでは、Variant の使用方法は数多くあります。次に例を示します。

- パラメータとして渡されるデータの型が正確にわからない場合に、メソッドや関数を作成します。
- さまざまなデータ型を 1 つの SafeArray に格納します。

SafeArray の要素は、すべて同じサイズである必要があります。Variant を処理するために SafeArray を作成した場合、この配列の各要素は、Variant にラップされた別の型でもかまいません。

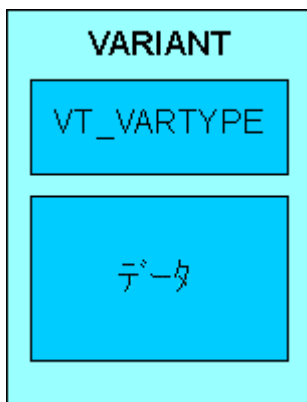


図 5-3 型情報とデータをラップする OLE Variant

コピー ファイル mfole.cpy は、新しい COBOL データ型 VARIANT を定義します。VARIANT は、OLE Variant を記述するデータ構造体を定義します。また、このコピー ファイルは、レベル 78 のデータ項目セットも定義し、VARIANT が保持できるさまざまなデータ型に対応する値を指定します。次のコード例では、VARIANT データ項目の VARIANT-vartype フィールドを調べ、mfole.cpy のレベル 78 データ項目と比較して、variant が整数型か文字列型のデータを含むかどうかを調べます。

```
working-storage section.  
  
COPY "MFOLE.CPY".  
  
01 aVariant          usage VARIANT.
```

```
01 vtype          pic x(2) comp-5.
```

```
...
```

```
procedure division.
```

```
...
```

\*> VARIANT-vartype は、型情報を含む

\*> VARIANT データ項目のフィールドです。

```
move VARIANT-vartype of aVariant to vType
```

```
evaluate vType
```

```
when VT-I2
```

```
display "2 バイトの整数型"
```

```
when VT-I4
```

```
display "4 バイトの整数型"
```

```
when VT-BSTR
```

```
display "文字列型"
```

```
when other
```

```
display "その他の型"
```

```
end-evaluate
```

Object COBOL の OLE オートメーション サポートには、VARIANT データの受け渡しを可能にする OLEVariant クラスがあります。OLEVariant クラスには、ネイティブの Object COBOL データ型として文字列またはオブジェクトリファレンスを含む VARIANT データにアクセスするメソッドが含まれています。他の型の VARIANT データを処理するには、VARIANT データ項目を宣言し、OLEVariant オブジェクトからのデータをデータ項目に読み込みます。

mfole.cpy で指定された VARIANT 型定義を使用して VARIANT データ項目の構造体を COBOL で直接表現できるのに、なぜ OLEVariant クラスが必要なのか疑問に思うことがあるかもしれません。その理由は、OLE オートメーションが VARIANT の割り当て、操作、割り当て解除に Windows API 関数セットを使用しており、OLEVariant クラスがこれらの関数に COBOL からの単純なインターフェイスを提供しているためです。

次の項では、OLEVariant クラスの使用方法を説明します。

- OleVariant クラスを使用する前に
- OLEVariant インスタンスの作成
- OLEVariant からのデータ読み取り

これらの項では、OLEVariant クラスのメソッドの一部をリストします。OLEVariant メソッドの完全なリストについては、NetExpress ヘルプの「OLE オートメーション クラス ライブラリ」を参照してください。NetExpress の [ヘルプ トピック] メニューの [ヘルプ] をクリックしてから、[目次] タブの [リファレンス]、[OO クラス ライブラリ] をクリックし、次にクラス ライブラリ リファレンスのショートカット ボタンをクリックします。

---

注記: OLE VARIANT データ型の詳細説明は、Win 32 SDK ヘルプの「OLE プログラマーズ リファレンス」を参照してください。Win 32 SDK は、NetExpress に含まれています。

---

#### 5.4.1 OleVariant クラスを使用する前に

OleVariant を使用する Object COBOL クラスは Class-Control の段落で Object COBOL クラス OleVariant を宣言し、mfole.cpy を作業場所節にコピーする必要があります。

```
class-control.          ...      class OleVariant is class "olevar".

...

working-storage section.

    copy "mfole.cpy".

...
```

コピー ファイル mfole.cpy には、OLEVariant を操作する場合に必要なデータ構造体の COBOL 型定義が含まれません。

#### 5.4.2 OLEVariant インスタンスの作成

OLEVariant インスタンスのデータを初期化する方法は、格納されているデータの型によって異なります。数字データを VARIANT データ項目に直接入力し、これを使用して OLEVariant インスタンスを初期化することができます。文字列は、Windows の API を呼び出して作成される OLE BSTRINGS として格納されます。OLE オブジェクトは、Object COBOL プログラムで使用できる Object COBOL プロキシ リファレンスではなく、VT\_DISPATCH リファレンスを使用して保存します。OLEVariant クラスには、BSTRING を自動的に作成し、これを格納する前に自動的に

プロキシ リファレンスを VT\_DISPATCH にマップし直すメソッドがあります。

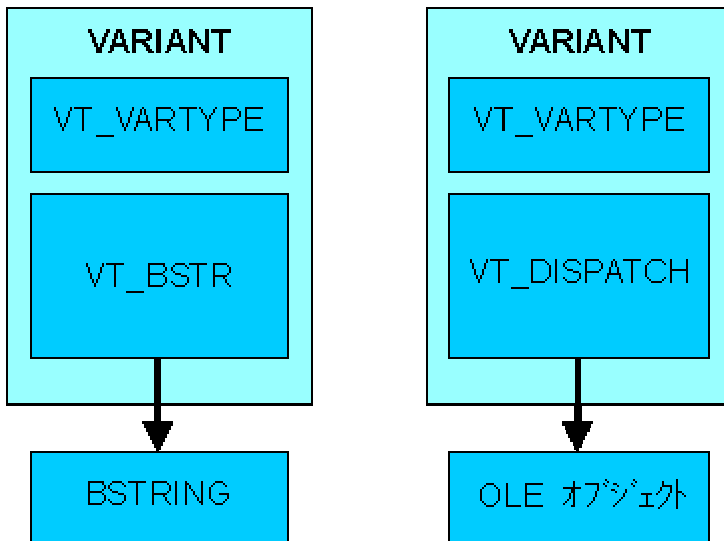


図 5-4 文字列とオブジェクト用の Windows の Variant

OLEVariant オブジェクトを作成するためのメソッドは、他に 2 通りあります。文字列と OLE オブジェクトのどちらかを保持するために OLEVariant を作成する場合と、他の variant データ型を 1 つ保持するために OLEVariant を作成する場合で、使用するメソッドは異なります。

文字列または OLEVariant オブジェクトを保持するために、OLEVariant インスタンスを作成する場合、次の手順にしたがいます。

1. プログラムの作業場所節にコピー ファイル mfole.cpy を記述します。
2. OLEVariant クラスに "new" メッセージを送信します。その結果、初期化されていない OLEVariant インスタンスが返されます。
3. PIC X(n) の文字列、Object COBOL CharacterArray の文字列、または OLE オブジェクトのどれを格納するかに応じて、インスタンスに "setString"、"setChararry"、または "setOLEObject" メッセージのどれかを送信します。

他の型のデータを保持するために、OLEVariant インスタンスを作成する場合、次の手順にしたがいます。

1. プログラムの作業場所節にコピー ファイル mfole.cpy を含めます。
2. VARIANT データ項目を宣言し、初期化します。
  - a. データ型識別子を、保持するデータ型の VARIANT データ項目で VARIANT-VARTYPE フィールドに移動します。データ型識別子は mfole.cpy で レベル 78 として定義され、すべての前に "VT-" が付けられます。
  - b. データを VARIANT データ項目の該当するフィールドに移動します。VARIANT データ フィールド

ドには、VARIANT-VT-I2 のような名前が付けられます 詳細なリストについては、コピー ファイル mfole.cpy の VARIANT 型定義を参照してください。VARIANT-VT-BSTR のようなデータ型では、保存データは、データ構造体へのポインタとなります。

3. "newWithData" メッセージを OLEVariant クラスに送信します。その際に VARIANT データ項目をパラメータとして渡します。

次の例では、2 つの OLEVariant を作成しています。一方は文字列型を格納し、もう一方は 4 バイトの整数型を格納します。

```
working-storage section.  
  
copy "mfole.cpy".  
  
01 aCharArray          object reference.  
  
01 aVariantinstance   object reference.  
  
01 variantinstance1   object reference.  
  
01 variantinstance2   object reference.  
  
01 vType              pic 9(4) comp-5.  
  
01 strLength          pic x(4) comp-5.  
  
01 winStatus          pic x(4) comp-5.  
  
01 aNumber            pic s9(9) comp-5.  
  
01 aString            pic x(12) value "I'm a string".  
  
01 vData1             VARIANT.  
  
01 vData2             VARIANT.  
  
01 vDataDisplay       VARIANT.  
  
...  
  
procedure division.  
  
...
```

\*>---OLEVariant インスタンスを作成し、文字列を格納します。

\*> まず、空の variant インスタンスを作成します。

```
invoke OLEVariant "new" using vData1  
  
        returning variantInstance1
```

\*>---文字列型データを設定します。

```
move length of aString to strLength  
  
invoke variantInstance1 "setString"using  
  
        by value strLength  
  
        by reference aString  
  
        returning winStatus  
  
...
```

\*>---OLEVariant インスタンスを作成し、4 バイトの整数を格納します。

```
move vt-I4 to variant-vartype of vData2  
  
move 99 to variant-vt-i4 of vData2  
  
invoke OLEVariant "newWithData" using vData2  
  
        returning variantInstance2  
  
...
```

### 5.4.3 OLEVariant からのデータ読み取り

OLEVariant インスタンスからデータを読み取る手順は、次のとおりです。

1. インスタンスから情報を検索するために VARIANT 型のデータ項目を宣言します。
2. "getVariant" メッセージを OLEVariant インスタンスに送信して、OLEVariant インスタンスに含まれる VARIANT 構造体のコピーを取得します。
3. データ型を調べるために VARIANT-VARTYPE フィールドをテストします。
4. データ型が OLE BSTRING である場合、"getString" メソッドまたは "getCharArray" メソッドを使用して、データを COBOL PIC X(n) または CharacterArray として検索する必要があります。データ型が OLE VT\_DISPATCH (オブジェクト リファレンス) である場合、"getOLEObject" メソッドを使用してプロキシ COBOL オブジェクト リファレンスを検索します。

他のデータ型については、VARIANT データ項目の適切なフィールドからデータを直接読み込むことができ

ます。

型を検索するための別の方法では、"getType" メッセージを OLEVariant インスタンスに送ります。詳細については、オンラインヘルプの「OLE オートメーション クラス ライブラリ リファレンス」を参照してください。

次のコード例は、VARIANT データ項目からデータを読み取る方法を示します。

```
working-storage section.  
  
copy "mfole.cpy".  
  
01 aCharArray          object reference.  
  
01 aVariantinstance   object reference.  
  
01 strLength          pic x(4) comp-5.  
  
01 winStatus          pic x(4) comp-5.  
  
01 aNumber            pic s9(9) comp-5.  
  
01 vDataDisplay       VARIANT.  
  
...  
  
procedure division.  
  
...  
  
evaluate variant-vartype of vDataDisplay *> data-type  
  
when vt-bstr *> これらの定数は、MFOLE.CPY で宣言されます。  
  
    invoke aVariantInstance "getCharArray"  
  
        using aCharArray  
  
        returning winStatus  
  
    invoke aCharArray "display"  
  
when vt-I4  
  
    move variant-vt-i2 of vDataDisplay to aNumber  
  
    display aNumber  
  
when vt-I2
```



```

move variant-vt-i4 of vDataDisplay to aNumber

display aNumber

when other

display "文字列型、2 バイト整数型、4 バイト整数型のどれにも該当しません。"

end-evaluate

...

```

## 5.5 SafeArray 型

OLE SafeArray は、プロセスの境界を安全に通ることができる固定された  $n$  次元の配列です。OLE は、SafeArray を操作するための API を装備しているので、SafeArray の作成や破棄、SafeArray に含まれるデータの操作が可能です。Object COBOL から OleSafeArray クラスを通して SafeArray を操作することができます。OleSafeArray は、Object COBOL の OLE オートメーション サポートの 1 つです。

1 つの次元で最初にある要素の位置は、その次元の下限になります。次元の下限は、SafeArray を定義するとき任意の整数として定義することができます。たとえば、下図は、 $6 \times 7$  の要素を持つ 2 次元の SafeArray を表しますが、次元 1 と次元 2 の下限を 0 に設定すると、赤いセルのアドレスは「2、1」となります。次元 1 と次元 2 の下限を 1 に設定すると、赤いセルのアドレスは「3、2」となります。

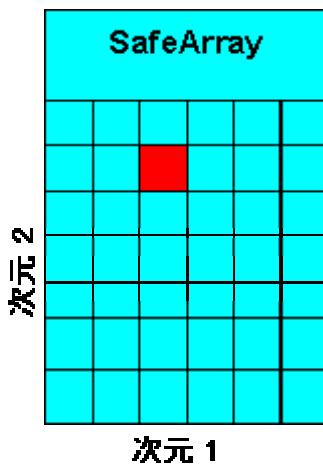


図 5-5 2 次元の SafeArray

次の項では、SafeArray を作成し、OLE オブジェクトにデータを渡す方法と、プログラムに渡された SafeArray を問い合わせる方法について説明します。

- SafeArray を使用する前に
- SafeArray の作成

- SafeArray に関する情報検索
- SafeArray 要素の読み書き
- SafeArray データへの直接アクセス

この章では、OLESafeArray クラスのメソッドをすべてリストしているわけではありません。OLESafeArray メソッドの詳細なリストについては、NetExpress ヘルプの「OLE オートメーション クラス ライブラリ」を参照してください。NetExpress の [ヘルプ トピック] メニューの [ヘルプ] をクリックしてから、[目次] タブの [リファレンス]、[OO クラス ライブラリ] をクリックし、次にクラス ライブラリ リファレンスのショートカット ボタンをクリックします。

### 5.5.1 SafeArray を使用する前に

SafeArray を使用する Object COBOL クラスは、Class-Control の段落で Object COBOL クラス OleSafeArray を宣言し、olesafea.cpy を作業場所節にコピーする必要があります。

```
class-control.

...

class OleSafeArray is class "olesafea".

...

working-storage section.

copy "olesafea.cpy".

...
```

### 5.5.2 SafeArray の作成

SafeArray を作成する前に、次の情報を設定する必要があります。

- 配列の次元を表す数

SafeArray の次元には、1 以上のすべての数を設定できます。

- 各次元の境界

境界は、下限として定義されます。境界は、配列の最初にある要素の索引と次元の要素数を示します。SAFEARRAYBOUND 型のデータ項目表を使用して、各データ項目の境界を宣言します。このデータ型は、コピー ファイル olesafea.cpy で定義されています。

- SafeArray に格納するデータの型

SafeArray は、variant の SafeArray を作成し、異なる型を格納する variant オブジェクトを使用できる一方で、SafeArray に格納されているすべてのデータは同じ型である必要があります。SafeArray に格納できる異なるデータ型は、コピー ファイル olesafea.cpy のレベル 78 データ項目として識別されます。

SafeArray を作成するには、次のようなコードを作成します。

```
invoke OleSafeArray "new" using by value vType
                                by value dimensions
                                by reference saBounds(1)
                                returning aSafeArray
```

パラメータの内容は、次のとおりです。

| パラメータ             | COBOL データ型                     | 説明  |
|-------------------|--------------------------------|---|
| <i>dimensions</i> | pic x(4) comp-5                | SafeArray の次元を表す数です。  |
| <i>saBounds</i>   | SAFEARRAYBOUND occurs <i>n</i> | <i>n</i> は、SafeArray の次元を表す数を指します ( <i>dimensions</i> の値)。<br><br>SAFEARRAYBOUND 型のデータ項目には、2 つの要素があります。<br><br>この 2 つの要素とは、llBounds と cElements を指し、次元の下限と次元の要素数を設定することができます。 |
| <i>vType</i>      | PIC X(4) COMP-5                | SafeArray に格納するデータ型に設定します。<br><br>異なる OLE データ型は、コピー ファイル olesafea.cpy のレベル 78 として定義されます。  |

次の例では、3×2 の要素を持つ 2 次元の SafeArray を設定します。

```
program-id. tplolec.
object section.
class-control.
    OleSafeArray is class "olesafea"
.
working-storage section.
```

```
copy "mfole.cpy".  
  
copy "olesafea.cpy".  
  
01 saBound          SAFEARRAYBOUND occurs 2.  
  
01 intSafeArray     object reference.  
  
01 varType          pic 9(4) comp-5.  
  
01 dimensions       pic x(4) comp-5.  
  
procedure division.
```

\*>---データ型を 4 バイトの整数型に設定します。

\*> VT-I4 は、MFOLE.CPY で定義されている

\*> 整数用の OLE データ型です。

```
move VT-I4 to varType
```

\*>---配列を 2 次元として設定します。

```
move 2 to dimensions
```

\*>---これを、下限が 0 で要素数が 3×2 の配列として

\*> 定義します。

\*> (下限は、次元の最初にある要素の索引となります。)

```
move 3 to cElements of saBound(1) *>cElements は、
```

\*>次元の大きさを

\*>設定する

\*>SAFEARRAY 型の

\*>副項目です。

```
move 0 to llBound of saBound(1) *>llBound は、
```

\*>次元の

\*>下限を

\*>設定する

\*>SAFEARRAY 型の

\*>副項目です。

```
move 2 to cElements of saBound(2)

move 0 to llBound of saBound(2)

invoke OleSafeArray "new" using

    by value varType

    by value dimensions

    by reference saBound(1)

    returning intSafeArray
```

### 5.5.3 SafeArray に関する情報の取得

SafeArray を表す OleSafeArray インスタンスを問い合わせ、次の情報を検索することができます。

- 次元の数 (メソッド "getDims")
- 各次元の上限と下限 (メソッド "getLBound" とメソッド "getUBound")

これらのメソッドは、操作が失敗したことを示す 0 以外の値以外に、Windows の状態コードも返します。たとえば、存在しない次元の大きさを検索しようとした場合などに、これらのコードが返されます。

- 配列に格納されているデータの型 (メソッド "getVarType")
- 要素の大きさ (メソッド "getElementSize")

すべてのデータは、PIC X(4) COMP-5 型で返されます。どこか別の場所からプログラムに渡され、大きさがわからない SafeArray を処理する場合、常に、この情報が必要になります。

次のコード例では、SafeArray の次元数と最初の次元の大きさを問い合わせます。

```
working-storage section.

...

01 dimensions          pic x(4) comp-5.

01 dimensionSize      pic x(4) comp-5.
```

```
01 intSafeArray      object reference.
01 lBound            pic x(4) comp-5.
01 uBound            pic x(4) comp-5.
01 hResult           pic x(4) comp-5.
01 varType           pic x(4) comp-5.
```

...

```
procedure division.
```

...

\*>---配列の次元を表す数を検索します。

```
        invoke intSafeArray "getDim" returning dimensions
```

...

```
        move 1 to dimensions
```

\*>---hResult は、SafeArray を問い合わせたときに返される Windows の

\*> 状態コードです。0 は、成功を示します。0 以外は、失敗を示します。

\*> エラー コードは、コピー ファイル MFOLE.CPY で、

\*> レベル 78 データ項目として定義されています。

```
        invoke intSafeArray "getLBound" using by value dimensions
```

```
                                by reference lBound
```

```
                                returning hResult
```

```
        invoke intSafeArray "getUBound" using by value dimensions
```

```
                                by reference uBound
```

```
                                returning hResult
```

\*>---次元の大きさを計算します。

```
        subtract lBound from uBound
```

```
add 1 to uBound giving dimensionSize
```

\*>---配列に格納されているデータの型を検索します。

```
invoke intSafeArray "getVarType" returning varType
```

...

#### 5.5.4 SafeArray 要素の読み書き

OLESafeArray クラスには、格納されているデータの型に応じて、個々の要素を読み取り、書き込むためのさまざまなメソッドがあります。

- 文字列
  - "putCharArray" メソッドと "getCharArray" メソッドでは、Object COBOL CharacterArray を使用して文字列を SafeArray と受け渡しすることができます
  - "putString" メソッドと "getString" メソッドでは、PIC X(n) データ項目を使用して文字列を SafeArray と受け渡しすることができます。"getString" を使用して文字列を受け取るには、データ項目に十分な領域を割り当てる必要があります。領域が足りない場合、メモリが破損します。  
  
これらのメソッドでは、文字列長は PIC X(4) COMP-5 データ項目の値として渡されます。
- OLE オブジェクト:
  - "putOLEObject" メソッドと "getOLEObject" メソッドを使用すると、SafeArray と OLE オブジェクトを受け渡しできます。
- VT\_BSTR 型 と VT\_DISPATCH 型の VARIANT
  - VT\_BSTR の場合、"getCharArrayFromVariant" メソッド と "putCharArrayAsVariant" メソッドを使用して、CharacterArray として、文字列にアクセスすることができます。また、"getStringFromVariant" メソッドと "putStringAsVariant" メソッドを使用すると、PIC X(n) として、文字列にアクセスすることができます。
  - VT\_DISPATCH の場合、"putOLEObjectAsVariant" メソッドと "getOLEObjectFromVariant" メソッドを使用して、OLE オブジェクトにアクセスすることができます。

これらの "get" のつくメソッドは、すべて、アクセス中の要素が正しい型であることを確認し、正しくない場合にエラー コードを返します。

- すべてのデータ型

"getElement" メソッドは、指定された要素のデータを自動的にメモリ領域にコピーします。メソッドには、

POINTER を指定してください。POINTER が、データを受け取るのに十分な大きさのデータ項目を指していることを確認してください。"putElement" メソッドは、指定した POINTER が指す領域からデータをコピーします。"getElementSize" メソッドを使用すると、SafeArray の要素の大きさを調べることができます。

これらのメソッドは、すべて、必要な個々の要素を特定し、エラー コードを返すために、実際のデータのデータ項目 ("getElement" メソッドと "putElement" メソッドの場合は、POINTER) を索引表に渡す必要があるため、酷似しているように見えます。また、文字列にアクセスするメソッドでは、PIC X(4) COMP-5 データ項目を文字列長の値で渡す必要があります。

次のコード例では、"putElement" メソッドを使用して、3×2 の SafeArray に数字データを格納します。また、"getCharArray" メソッドを使用して、VT\_BSTR 型の SafeArray から文字列を検索します。

```
working-storage section.
```

```
copy "olesafea.cpy".
```

```
01 saBound          SAFEARRAYBOUND occurs 2.
```

```
01 intSafeArray     object reference.
```

```
01 strSafeArray     object reference.
```

```
01 iIndex           pic x(4) comp-5 occurs 2.
```

```
01 hIndex           pic x(4) comp-5.
```

```
01 vIndex           pic x(4) comp-5.
```

```
01 iValue           pic x(4) comp-5.
```

```
01 hResult          pic x(4) comp-5.
```

```
01 theData          POINTER.
```

```
...
```

```
procedure division.
```

```
...
```

```
move 9 to iValue
```

```
move 10 to strLength
```

```
set theData to address of iValue *> "putElement" は、アドレス ポインタ
```

\*> からデータを読み取ります。



```

perform varying hIndex from 0 by 1 until hIndex = 3
    perform varying vIndex from 0 by 1 until vIndex = 2
        move hIndex to iIndex(1)
        move vIndex to iIndex(2)
        invoke intSafeArray "putElement" using iIndex(1)
            by value theData
            returning hResult
        subtract 1 from iValue
    end-perform
end-perform ...
perform varying hIndex from 0 by 1 until hIndex = 3
    perform varying vIndex from 0 by 1 until vIndex = 2
        move hIndex to iIndex(1)
        move vIndex to iIndex(2)
        invoke strSafeArray "getCharArray" using iIndex(1)
            aCharArray
            returning hResult
        invoke aCharArray "display"
    end-perform
display " "
end-perform

```

### 5.5.5 SafeArray データへの直接アクセス

OLESafeArray のインスタンスによりラップされた Windows SafeArray のデータ構造体を持つメモリに直接アクセスすることができます。この操作は、SafeArray の内部構造に関して十分知識がない場合は行わないでください。

直接アクセスを行うための手順を次に示します。

- SafeArray データをロックし、"accessData" メソッドを使用して、アドレス ポインタを取得します。
- アドレス ポインタを使用すると、直接 SafeArray を操作することができます。
- 操作の終了後は、"unAccessData" メソッドを使用して、SafeArray のロックを解除します。

"accessData" メソッドと "unAccessData" メソッドの詳細については、「OLE オートメーション クラス ライブラリ リファレンス」を参照してください。NetExpress の [ヘルプ トピック] メニューの [ヘルプ] をクリックし、次に [目次] タブ上の [リファレンス]、[OO クラス ライブラリ] をクリックし、クラス ライブラリ リファレンスのショー トカット ボタンをクリックします。「クラス ライブラリ リファレンス」ウィンドウで、[目次] タブを選択し、「OLESafeArray クラス」を参照してください。

# 第6章 Microsoft Transaction Serverとのインタフェース

Microsoft Transaction Server は、コンポーネントベースのトランザクション処理システムであり、高性能で伸縮自在の強固な、企業向け、インターネット向け、イントラネット向けのサーバー アプリケーションを開発、配置、管理するために使用します。Transaction Server は、コンポーネントベースの分散アプリケーションを開発するためにアプリケーション プログラミング モデルを定義しています。また、これらのアプリケーションを配置し管理するための実行時の基盤も提供します。

Transaction Server を使用すると、トランザクションを個別の関数を実行するコンポーネントに分割することができます。コンポーネントは、Transaction Server に対して、正常に実行されたかどうかを通知します。トランザクションのコンポーネントがすべて正常に終了すると、トランザクションがコミットされます。正常に終了しないものがある場合には、トランザクションはロール バックされます。

この章では、NetExpress を使用して COBOL の Transaction Server コンポーネントを作成する方法について説明します。Transaction Server のセットアップ方法や管理方法、および、プログラマが使用できるより高度な機能については、ここでは説明しません。詳細については、Microsoft Transaction Server に添付されたマニュアルを参照してください。

## 6.1 Transaction Server コンポーネントの作成

Microsoft Transaction Server で使用するコンポーネントは、インプロセス OLE サーバーとしてコンパイルし、リンクする必要があります。この項では、コンポーネントの構造とビルド方法について説明します。NetExpress クラス ウィザードを使用すると、トランザクション サーバーコンポーネントのスケルトンを生成することができます (NetExpress の [ヘルプ] メニューで [ヘルプ トピック] をクリックし、[目次] タブを使用してクラス ウィザードを参照してください)。COBOL での OLE サーバー作成に関する詳細は、「OLE オートメーションと DCOM」の章を参照してください。

### 6.1.1 Transaction Server コンポーネントの構造

Transaction Server で使用されるすべてのコンポーネントは、次のような基本構造になっています。

```
$set ooctrl(+P)

class-id.

    component-name inherits from olebase.

object section.
```

class-control.

Olebase is class "olebase"

objectcontext is class "objectcontext".

object.

method-id. "method-name".

local-storage section.

01 Context                    object reference.

<< メソッドに対してローカルなデータ項目 >>

linkage section.

<< メソッドのパラメータ定義 >>

procedure division using *linkage-section-items*.

- \*    オブジェクトのコンテキストを取得します。
- \*    Transaction Server で制御されていない場合、
- \*    コンテキストは NULL に設定されます。

invoke objectcontext "GetObjectContext" returning Context

<< このコンポーネントに必要な処理を実行します。 >>

- \*    処理が成功した場合、SetComplete の呼び出しを実行します。
- \*    処理が失敗した場合、SetAbort の呼び出しを実行します。

if *everything-ok*

    if Context not = NULL

        invoke Context "setComplete"

    end-if

else

    if Context NOT = NULL

```

        invoke context "setAbort"

    end-if

end-if

*   コンポーネントが正常終了したか確認します。

    if Context not = NULL

        invoke Context "finalize" returning context

    end-if

    exit method.

end method "method-name".

end object.

end class component-name.

```

*component-name*                      コンポーネントの名前。この名前は、OLE prog-id および Microsoft Transaction Server エクスプローラで表示する名前として使用されます。

*method-name*                        コンポーネントで呼び出すメソッドの名前。コンポーネントには複数のメソッドを記述することができます。

### 6.1.2 objectcontext クラス

クラス *objectcontext* を使用すると、Transaction Server が備える機能にアクセスできるようになります。*objectcontext* は、次のような Class-Control 節に記述します。

```

class-control.

    Olebase is class "olebase"

    objectcontext is class "objectcontext".

```

---

注記: クラス *objectcontext* は、objectcontext.dll ファイルに記述されています。このファイルは、アプリケーションを分散させるときに、Transaction Server コンポーネントに含める必要があります。

---

### 6.1.3 コンテキスト オブジェクト

Microsoft Transaction Server の各コンポーネントには、関連付けられたコンテキスト オブジェクトがあります。コンテキスト オブジェクトは、拡張可能な Transaction Server オブジェクトです。これは、トランザクション、アクティビティ、セキュリティなどのプロパティを含めて、インスタンスを実行するためのコンテキストを提供します。Transaction Server コンポーネントが作成されると、Transaction Server は自動的にコンテキスト オブジェクトを作成します。Transaction Server コンポーネントが解放されると、Transaction Server も自動的にコンテキスト オブジェクトを解放します。

コンポーネントのコンテキスト オブジェクトを検索するには、クラス メソッド *GetObjectContext* を使用します。このメソッドは、コンテキスト オブジェクトに対してオブジェクト リファレンスを返します。たとえば、次のように記述します。

```
01 Context object reference.
```

```
invoke objectcontext "GetObjectContext" returning Context
```

このコンテキスト オブジェクトは、コンポーネントに固有なので、他のオブジェクトに渡すことはできません。

コンポーネントが Transaction Server の制御下で実行されていない場合 (Transaction Server パッケージにインポートされていない場合)、NULL 値がオブジェクト コンテキストに返されます。

### 6.1.4 Transaction Server コンポーネントの終了

コンポーネントを終了する前に、オブジェクト コンテキストを解放する必要があります。オブジェクト コンテキストを解放するには、*finalize* メソッドを使用します。たとえば、次のように記述します。

```
invoke Context "finalize"
```

```
returning context
```

このコードにより、コンポーネントのリファレンスが解放されるときに、コンポーネントがメモリから正しく削除されます。

### 6.1.5 Objectcontext のメソッド

次に示すクラス *objectcontext* のメソッドへは、コンポーネントのコンテキストを取得するとアクセスできるようになります。これらのメソッドは、同じ名前の Transaction Server メソッドに直接マップされます。これらのメソッドの機能に関する詳細は、『*Transaction Server プログラマーズ ガイド*』を参照してください。

#### 6.1.5.1 SetComplete

例

```
invoke Context "SetComplete"
```

現在のコンポーネントが処理を終了したことを宣言します。トランザクションの範囲で実行しているオブジェクトについては、オブジェクトのトランザクションによる更新をコミットできます。

#### 6.1.5.2 SetAbort

例

```
invoke Context "SetAbort"
```

オブジェクトが実行しているトランザクションを中断するように宣言します。

#### 6.1.5.3 CreateInstance

例

```
01 NewObject      object reference.  
  
invoke Context "CreateInstance"  
  
using z"Account"  
  
returning NewObject
```

現在のコンテキストを使用して、OLE コンポーネントの新しいインスタンスを作成します。渡される名前は、コンポーネントの prog-id です。これはヌルで終わる必要があります。*CreateInstance* は、新規コンポーネントのオブジェクト リファレンスを返します。新しいインスタンスは、現在のコンポーネントのコンテキストを使用します。

*CreateInstance*を使用してコンポーネントのインスタンスを作成すると、コンポーネントで "new" メソッドを使用してインスタンスを作成した場合と同じように使用することができます。

#### 6.1.5.4 DisableCommit

例

```
invoke Context "DisableCommit"
```

コンポーネントのトランザクションによる更新が一致せず、現在の状態をコミットできないことを宣言します。

#### 6.1.5.5 EnableCommit

例

```
invoke Context "EnableCommit"
```

現在のコンポーネントの処理が必ずしも完了していないこと、その一方でトランザクションによる更新が一致し、現

在の形式でコミットできることを宣言します。

#### 6.1.5.6 IsInTransaction

例

```
01 ReturnValue          pic x(4) comp-5.  
  
    invoke Context "IsInTransaction"  
  
returning ReturnValue
```

トランザクションで現在のコンポーネントが実行されている場合、0 以外の値を返します。値 0 は、コンポーネントがトランザクションで実行されていない場合に返されます。

#### 6.1.5.7 IsCallerInRole

例

```
01 ReturnValue          pic x(4) comp-5.  
  
    invoke Context "IsCallerInRole"  
  
        using z"Managers" ReturnValue
```

コンポーネントの直接の呼び出し者が指定した権限を持っているかどうかを示します (個別にまたはグループで)。呼び出し者が指定した権限を持っていない場合には、0 が返されます。呼び出し者が権限を持っている場合、または、セキュリティを使用できない場合、0 以外の値が返されます。権限はヌルで終わる文字列として指定する必要があります。

#### 6.1.5.8 IsSecurityEnabled

例

```
01 ReturnValue          pic x(4) comp-5.  
  
    invoke Context "IsSecurityEnabled"  
  
        returning ReturnValue
```

このコンポーネントに対してセキュリティが有効でない場合には 0 を返します。また、セキュリティが有効である場合には 0 以外の値を返します。

### 6.1.6 Transaction Server コンポーネントのビルド

NetExpress クラス ウィザード (NetExpress の [ヘルプ] メニューで [ヘルプ トピック] をクリックし、[目次] タブを使用してクラスウィザードを参照してください) を使用すると、Transaction Server コンポーネントのスケルトンを



生成することができます。コンポーネントをビルドするには、「OLE オートメーションと DCOM」の章の「インプロセス サーバーの作成」の項で説明する手順にしたがいます。

## 6.2 Transaction Server コンポーネントの実行とデバッグ

Transaction Server コンポーネントは、Transaction Server の制御下で実行されるため、コンポーネントの実行環境やコンポーネントの実行時にユーザーがログオンしている環境を推測することはできません。

NetExpress をインストールしていない PC でコンポーネントを実行するには、システム パスに COBOL ランタイム ファイルを含むディレクトリを追加する必要があります。このディレクトリを追加すると、コンポーネントは常に COBOL ランタイム ファイルを検出することができます。

コンポーネントをデバッグする場合、デバッグするコンポーネントのメソッドで手続き部の先頭に、次の行を追加する必要があります。

```
call "cbl_debugbreak"
```

このコードにより、コンポーネントの実行時に NetExpress デバッガが起動されます。ソース ファイルとデバッグ ファイルを NetExpress で検出できるようにするには、システム環境で次の環境変数を設定する必要があります。

COBIDY - コンポーネントの .idy ファイルを含むディレクトリを設定します。

COBCPY - コンポーネントのソース ファイルを含むディレクトリを設定します。

## 6.3 完全な Transaction Server コンポーネントの例

NetExpress は、完全な Transaction Server コンポーネントの例を含んでおり、これは、Transaction Server で提供されるアカウントオブジェクトの COBOL での実行です。この例は、account.cblと呼ばれ、¥netexpress¥base¥demo¥mtxディレクトリにこれを見つけることができます。

# 索引

|                   |            |                                |                         |
|-------------------|------------|--------------------------------|-------------------------|
| AccessData メソッド   | 5-22       | DCOM                           | 4-14, 4-32              |
| ActiveX           |            | DllOleLoadClasses              | 4-25                    |
| クライアント            | 4-2, 4-3   | dllserver.obj                  | 4-17                    |
| プロキシ              | 4-4, 4-9   | GetCharArray メソッド              | 5-11, 5-19              |
| プロパティ             | 4-19       | GetElement メソッド                | 5-19                    |
| メッセージ             | 4-6        | GetOLEObject メソッド              | 5-11, 5-19              |
| ActiveX オブジェクト    | 4-16       | GetString メソッド                 | 5-19                    |
| 起動                | 4-4        | GetVariant メソッド                | 5-11                    |
| デバッグ              | 4-26       | GetString メソッド                 | 5-11                    |
| 登録                | 4-20, 4-32 | IDL                            | 2-1, 2-2, 2-4, 2-6, 2-8 |
| メソッド              | 4-18       | 概要                             | 1-4                     |
| リモート              | 4-32       | コンパイラ                          | 2-2                     |
| ActiveX オブジェクトの起動 | 4-4        | サンプル                           | 2-5                     |
| CORBA             | 1-1, 2-2   | Makefiles                      | 2-8                     |
| CORBA と COBOL     |            | Mfole.cpy                      | 5-4, 5-6                |
| 概要                | 1-7        | Microsoft Transaction Server   | 6-1                     |
| CorbaGen          | 2-7, 2-17  | NewWithData メソッド               | 5-10                    |
| makefile          | 2-6, 2-18  | Object Management Architecture |                         |
| アニメーションの準備        | 2-20       | 概要                             | 1-2                     |
| 概要                | 2-1        | Object Management Group (OMG)  |                         |
| 構文                | 2-18       | 概要                             | 1-4                     |
| ラッパー              | 2-18       | Objectcontext メソッド             | 6-4                     |
|                   |            | OLE オートメーション                   |                         |

|                            |            |                           |      |
|----------------------------|------------|---------------------------|------|
| OLE エラー コード .....          | 4-11       | setDispatchType .....     | 4-8  |
| イベント ループ .....             | 4-27       | OLEVariant .....          | 5-6  |
| オブジェクト リファレンス .....        | 5-4        | 作成 .....                  | 5-8  |
| クライアント .....               | 4-3        | 使用する前 .....               | 5-8  |
| サーバー .....                 | 4-13       | データの読み取り .....            | 5-11 |
| サーバー クラス .....             | 4-16       | 例 .....                   | 5-10 |
| サーバーのデバッグ .....            | 4-26       | OLEVariant の作成 .....      | 5-10 |
| データ型 .....                 | 5-1        | OMG .....                 | 2-2  |
| データ型識別子 .....              | 5-9        | OOCTRL(+P)                |      |
| データ型指定 .....               | 5-2        | 指令 .....                  | 4-13 |
| 登録 .....                   | 4-20, 4-32 | ORB                       |      |
| プロパティ .....                | 4-19       | オブジェクト リクエスト ブローカ .....   | 3-1  |
| メッセージ型 .....               | 4-8        | Orbix                     |      |
| 例外 .....                   | 4-9        | CORBA ウィザードの使用 .....      | 2-21 |
| 例外基本番号 .....               | 4-10       | CorbaGen の使用 .....        | 2-20 |
| OLE オートメーション サーバーの登録 ..... | 4-20       | IDL コンパイラ .....           | 2-2  |
| OLE オートメーション サーバーの分散 ..... | 4-32       | アニメート .....               | 2-20 |
| OLE オブジェクト .....           | 5-4        | アプリケーションの開発 .....         | 2-6  |
| SafeArrays .....           | 5-19       | 概要 .....                  | 2-2  |
| ole32lib .....             | 4-17       | 固有のオブジェクト リファレンス .....    | 2-3  |
| Olesafea.cpy .....         | 5-14       | サポート制限事項 .....            | 2-21 |
| olesup .....               | 5-4        | デバッグとアニメート .....          | 2-19 |
| olesup クラス .....           | 4-8, 4-11  | デモンストレーション アプリケーション ..... | 2-6  |
| getBaseOleException .....  | 4-10       | ネーミング サービス .....          | 2-4  |
| getLastSCode .....         | 4-11       | ロケーション透過性 .....           | 2-3  |

|                            |            |                                     |                  |
|----------------------------|------------|-------------------------------------|------------------|
| Orbix アプリケーションの開発.....     | 2-10       | データ型.....                           | 5-17             |
| Orbix の CORBA ウィザード        |            | 文字列.....                            | 5-19             |
| アニメーションの準備.....            | 2-21       | 要素の大きさ.....                         | 5-17             |
| Orbix 用 CORBA ウィザード.....   | 2-7        | 読み取り.....                           | 5-19             |
| PutCharArray メソッド.....     | 5-19       | 例.....                              | 5-15, 5-17, 5-20 |
| PutElement メソッド.....       | 5-20       | SetCharArray メソッド.....              | 5-10             |
| PutOLEObject メソッド.....     | 5-19       | SetOLEObject メソッド.....              | 5-10             |
| putString メソッド.....        | 5-19       | Setstring メソッド.....                 | 5-10             |
| QueryClassInfo メソッド.....   | 4-21       | stubgen ユーティリティ.....                | 2-6              |
| QueryLibraryInfo メソッド..... | 4-21       | Transaction Server (Microsoft)..... | 6-1              |
| Regedit.....               | 4-21, 4-35 | objectcontext クラス.....              | 6-3              |
| RegisterServer メソッド.....   | 4-24       | objectcontext.dll.....              | 6-3              |
| regsvr32.....              | 4-25       | コンテキスト オブジェクト.....                  | 6-4              |
| SafeArray.....             | 5-13       | コンポーネントの構造.....                     | 6-1              |
| OLE オブジェクト.....            | 5-19       | コンポーネントの作成.....                     | 6-1              |
| olesafea.cpy.....          | 5-14       | コンポーネントのビルド.....                    | 6-6              |
| Variant.....               | 5-19       | 実行.....                             | 6-7              |
| 書き込み.....                  | 5-19       | デバッグ.....                           | 6-7              |
| 境界.....                    | 5-17       | 例.....                              | 6-7              |
| 索引.....                    | 5-13       | Trasaction Server (Microsoft).....  | 6-4              |
| 作成.....                    | 5-14       | GetObjectContext.....               | 6-4              |
| 次元.....                    | 5-13, 5-17 | Objectcontext のメソッド.....            | 6-4              |
| 前提条件.....                  | 5-14       | UnAccessData メソッド.....              | 5-22             |
| その他のデータ型.....              | 5-19       | UnRegisterServer メソッド.....          | 4-24             |
| 直接アクセス.....                | 5-21       | UUIDGEN.....                        | 4-35             |

|                             |                    |            |
|-----------------------------|--------------------|------------|
| Variant                     | 概要 .....           | 1-2        |
| SafeArray .....             |                    | 5-19       |
| データ型 .....                  |                    | 5-9        |
| データ構造化 .....                |                    | 5-8        |
| Variant -vartype .....      |                    | 5-9        |
| Variant データ型 .....          |                    | 5-6        |
| Variant-vartype フィールド ..... |                    | 5-6        |
| VT_DISPATCH .....           |                    | 5-4        |
| Windows レジストリ .....         |                    | 4-20, 4-32 |
| エントリの編集 .....               |                    | 4-34       |
| アウトオブプロセス サーバーの変換 .....     |                    | 4-17       |
| アニメート                       |                    |            |
| Orbix .....                 |                    | 2-20       |
| インターフェイス定義言語 .....          |                    | 2-1, 2-2   |
| インプロセス サーバー                 |                    |            |
| ビルド .....                   |                    | 4-17       |
| ウィザード                       |                    |            |
| クラス .....                   |                    | 4-14, 4-16 |
| エラー コード                     |                    |            |
| OLE オートメーション .....          |                    | 4-11       |
| オブジェクト サービス                 |                    |            |
| 概要 .....                    |                    | 1-2        |
| セキュリティ .....                |                    | 2-2        |
| トランザクション .....              |                    | 2-2        |
| オブジェクト リクエスト ブローカ .....     |                    | 3-1        |
|                             | オブジェクト リファレンス      |            |
|                             | OLE オートメーション ..... | 5-4        |
|                             | 型指定 .....          | 3-1        |
|                             | OLE オートメーション ..... | 5-2        |
|                             | クライアント             |            |
|                             | ActiveX .....      | 4-2        |
|                             | クラス ウィザード .....    | 4-14, 4-16 |
|                             | コンパイラ指令            |            |
|                             | OOCTRL(+P) .....   | 4-13       |
|                             | 集約 .....           | 4-13       |
|                             | 終了                 |            |
|                             | アクティブプロキシ .....    | 4-9        |
|                             | スタブ .....          | 2-1, 2-8   |
|                             | クライアント .....       | 2-2        |
|                             | サーバー .....         | 2-2        |
|                             | タイプ ライブラリ .....    | 4-8, 4-28  |
|                             | データ型               |            |
|                             | Variant .....      | 5-6        |
|                             | 識別子 .....          | 5-6        |
|                             | データ型識別子 .....      | 5-9        |
|                             | データ型指定             |            |
|                             | OLE オートメーション ..... | 5-2        |
|                             | データ型の指定 .....      | 3-1        |
|                             | デバッグ               |            |

|                            |          |
|----------------------------|----------|
| ActiveX オブジェクト .....       | 4-26     |
| OLE オートメーション サーバー .....    | 4-26     |
| <b>登録</b>                  |          |
| ActiveX オブジェクト .....       | 4-20     |
| リモートの ActiveX オブジェクト ..... | 4-32     |
| ドメイン .....                 | 3-1      |
| OLE オートメーション .....         | 4-1      |
| クラス .....                  | 3-2      |
| メッセージ .....                | 3-2      |
| <b>ドメイン インターフェイス</b>       |          |
| 概要 .....                   | 1-3      |
| プロキシ オブジェクト .....          | 4-4, 4-9 |
| プロキシ ハンドル .....            | 5-4      |
| プロパティ .....                | 4-19     |
| <b>プロパティ、ActiveX</b>       |          |
| 取得 .....                   | 4-6, 4-8 |
| 設定 .....                   | 4-6, 4-8 |

|                                 |      |
|---------------------------------|------|
| <b>分散オブジェクト</b>                 |      |
| 概要 .....                        | 1-1  |
| <b>分散コンポーネント</b>                |      |
| Orbix .....                     | 2-1  |
| <b>マルチスレッド</b>                  |      |
| OLE オートメーション .....              | 4-30 |
| <b>メソッド</b>                     |      |
| ActiveX オブジェクト .....            | 4-18 |
| <b>ラッパー</b>                     |      |
| 概要 .....                        | 2-1  |
| リモート OLE オートメーション サーバーの登録 ..... | 4-32 |
| <b>例外</b>                       |      |
| OLE オートメーション .....              | 4-9  |
| レガシー アプリケーション .....             | 2-1  |
| 移行 .....                        | 2-8  |
| レジストリ エントリ .....                | 4-17 |
| レジストリ エントリの編集 .....             | 4-34 |