VisiBroker for C++ 開発者ガイド

Borland VisiBroker® 7.0

Borland®
Excellence Endures™

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

ライセンス規定および限定付き保証にしたがって配布が可能なファイルについては，deploy.html ファイルを参照してください。

Borland Software Corporation は，本書に記載されているアプリケーションに対する特許を取得または申請している場合があります。該当する特許のリストについては，製品 CD または［バージョン情報］ダイアログボックスをご覧ください。本書の提供は，これらの特許に関する権利を付与することを意味するものではありません。

Copyright 1992-2006 Borland Software Corporation. All rights reserved. すべての Borland のブランド名および製品名は，米国およびその他の国における Borland Software Corporation の商標または登録商標です。その他のブランドまたは製品名は，その版権所有者の商標または登録商標です。

Microsoft，.NET ロゴおよび Visual Studio は，Microsoft Corporation の米国およびその他の国における商標または登録商標です。

サードパーティの条項と免責事項については，製品 CD に収録されているリリースノートを参照してください。

2006 年 5 月 11 日初版発行
著者：Borland Software Corporation
発行：ボーランド株式会社
PDF

目次

第 1 章
Borland VisiBroker の概要 1
VisiBroker の概要 1
VisiBroker の機能 2
VisiBroker のマニュアル 2
スタンドアロンヘルプビューアからの VisiBroker オンラインヘルプトピックへのアクセス 3
VisiBroker コンソールからの VisiBroker オンラインヘルプトピックへのアクセス 3
マニュアルの表記規則 4
プラットフォームの表記 4
Borland サポートへの連絡 4
オンラインリソース 5
Web サイト 5
Borland ニュースグループ 5

第 2 章
CORBA モデルの概要 7
CORBA の概要 7
VisiBroker の概要 8
VisiBroker の機能 9
VisiBroker のスマートエージェント（osagent）アーキテクチャ 9
ロケーションサービスを使った高度なオブジェクト検索機能 9
インプリメンテーションとオブジェクトのアクティブ化のサポート 9
堅牢なスレッドと接続の管理 9
IDL コンパイラ 10
DII と DSI による動的起動 10
インターフェースとインプリメンテーションリポジトリ 10
サーバー側の可搬性 10
インターセプタとオブジェクトラッパーを使った ORB のカスタマイズ 11
イベントキュー .11
ネーミングサービスのバックストア 11
GateKeeper 11
VisiBroker CORBA 準拠 11
VisiBroker 開発環境 11
プログラマツール 12
CORBA サービスツール 12
管理ツール 12
VisiBroker との相互運用性 12
ほかの ORB 製品 13
IDL から C++ へのマッピング 13

第 3 章
VisiBroker を使ったサンプルアプリケーションの開発 15
開発手順 15
ステップ 1 : オブジェクトインターフェースの定義 . . . 17
IDL を使った Account インターフェースの記述 . . 17
ステップ 2 : クライアントスタブとサーバーサーバントの生成 17
IDL コンパイラが生成するファイル 17
ステップ 3 : クライアントの実装 18
Client.C 18
AccountManager オブジェクトのへのバインド . .19
Account オブジェクトの取得 19
残高の取得 19
ステップ 4 : サーバーの実装 19
サーバープログラム 19
Account クラスの階層 20
ステップ 5 : サンプルのビルド 20
サンプルのコンパイル 21
ステップ 6 : サーバーの起動とサンプルの実行 21
スマートエージェントの起動 21
サーバーの起動 .21
クライアントの実行 22
VisiBroker を使ったアプリケーションの配布 22
VisiBroker アプリケーション 23
アプリケーションの配布 23
環境変数 23
サポートサービスの有効性 23
アプリケーションの実行 23
クライアントアプリケーションの実行 24

第 4 章
C++ 対応プログラマツール 25
VisiBroker for C++ のヘッダーファイル用スイッチ . . .25
_VIS_STD 25
_VIS_NOLIB 25
引数／オプション .26
共通オプション .26
一般情報 26
idl2cpp 26
idl2ir 29
ir2idl 30
idl2wsc 30
idl2wsc の使用 30
idl2wsc の制限 31

第 5 章
IDL から C++ へのマッピング 33
プリミティブデータ型 33
文字列 34
String_var クラス 34
定数 35
定数に関する特殊なケース 35
列挙体 36
型定義 36
モジュール 37
複合データ型 38
構造体 38
固定長の構造体 .38
可変長の構造体 .39
構造体のメモリ管理 39
共用体 40
共用体に対して管理される型 41
共用体のメモリ管理 41
シーケンス 41
シーケンスに対して管理される型 43
シーケンスのメモリ管理 43

配列 . . . 44
配列スライス . . . 44
配列に対して管理される型 . . . 44
タイプセーフ配列 . . . 45
配列のメモリ管理 . . . 45
プリンシパル . . . 46
valuetype . . . 46
値ボックス . . . 49
抽象インターフェース . . . 49

第 6 章
VisiBroker のプロパティ 51

スマートエージェントおよびスマートエージェント通信のプロパティ . . . 51
VisiBroker ORB のプロパティ . . . 53
サーバーマネージャのプロパティ . . . 56
追加プロパティ . . . 56
サーバー側のリソース使用量関連のプロパティ . . . 56
クライアント側のリソース使用量関連のプロパティ 57
スマートエージェント関連のプロパティ . . . 57
その他のプロパティ . . . 57
ロケーションサービスのプロパティ . . . 58
イベントサービスのプロパティ . . . 58
ネーミングサービス（VisiNaming）のプロパティ . . . 58
取り替え可能なバックストアプロパティ . . . 61
すべてのアダプタに共通するデフォルトのプロパティ 62
JDBC アダプタのプロパティ . . . 62
DataExpress アダプタのプロパティ . . . 63
JNDI アダプタのプロパティ . . . 64
VisiNaming サービスのセキュリティ関連プロパティ . . . 64
OAD のプロパティ . . . 65
インターフェースリポジトリのプロパティ . . . 65
TypeCode のプロパティ . . . 66
クライアント側 LIOP 接続のプロパティ . . . 66
クライアント側 IIOP 接続のプロパティ . . . 67
QoS 関連のプロパティ . . . 68
クライアント側インプロセス接続のプロパティ . . . 68
サーバー側サーバーエンジンのプロパティ . . . 69
サーバー側スレッドセッション IIOP_TS/IIOP_TS 接続のプロパティ . . . 69
サーバー側スレッドセッション BOA_TS/BOA_TS 接続のプロパティ . . . 70
サーバー側スレッドプール IIOP_TP/IIOP_TP 接続のプロパティ . . . 70
サーバー側スレッドプール BOA_TP/BOA_TP 接続のプロパティ . . . 72
サーバー側スレッドプール LIOP_TP/LIOP_TP 接続のプロパティ . . . 72
サーバー側スレッドプール BOA_LTP/BOA_LTP 接続のプロパティ . . . 73
双方向通信をサポートするプロパティ . . . 74
デバッグログのプロパティ . . . 74
例 . . . 77
例 . . . 77
Web サービスランタイムのプロパティ . . . 78
ランタイムを有効にする . . . 78
Web サービス HTTP リスナープロパティ . . . 78
Web サービス接続マネージャのプロパティ . . . 79
SOAP 要求ディスパッチャのプロパティ . . . 79

第 7 章
例外処理 81

CORBA モデルにおける例外 . . . 81
システム例外 . . . 81
SystemException クラス . . . 82
完了状態の取得 . . . 83
マイナーコードの取得と設定 . . . 83
システム例外の種類の判定 . . . 83
システム例外のキャッチ . . . 83
例外をシステム例外にダウンキャスト . . . 84
特定のシステム例外のキャッチ . . . 84
ユーザー例外 . . . 85
ユーザー例外の定義 . . . 85
例外を生成するためのオブジェクトの変更 . . . 86
ユーザー例外のキャッチ . . . 86
ユーザー例外へのフィールドの追加 . . . 86

第 8 章
サーバーの基礎 87

概要 . . . 87
VisiBroker ORB の初期化 . . . 87
POA の作成 . . . 87
ルート POA へのリファレンスの取得 . . . 88
子 POA の作成 . . . 88
サーバントメソッドの実装 . . . 89
サーバントの作成およびアクティブ化 . . . 89
POA のアクティブ化 . . . 90
オブジェクトのアクティブ化 . . . 90
クライアント要求の待機 . . . 90
完全なサンプルコード . . . 90

第 9 章
POA の使い方 93

ポータブルオブジェクトアダプタの概要 . . . 93
POA の用語 . . . 94
POA の作成と使用の手順 . . . 95
POA ポリシー . . . 95
POA の作成 . . . 97
POA の命名規則 . . . 97
ルート POA の取得 . . . 97
POA ポリシーの設定 . . . 98
POA の作成およびアクティブ化 . . . 98
オブジェクトのアクティブ化 . . . 98
オブジェクトの明示的なアクティブ化 . . . 99
オンデマンドのオブジェクトのアクティブ化 . . . 99
オブジェクトの暗黙的なアクティブ化 . . . 100
デフォルトサーバントによるアクティブ化 . . . 100
オブジェクトの非アクティブ化 . . . 101
サーバントとサーバントマネージャの使い方 . . . 102
ServantActivators . . . 103
ServantLocators . . . 105
POA マネージャによる POA の管理 . . . 107
現在の状態の取得 . . . 108
停止状態 . . . 108
アクティブ状態 . . . 108
破棄状態 . . . 109
非アクティブ状態 . . . 109
リスナーとディスパッチャ：サーバーエンジン，サーバー接続マネージャ，およびそれらのプロパティ . . . 109

サーバーエンジンと POA 110
POA とサーバーエンジンの関連付け 111
サーバーエンジンのエンドポイントのホストの定義 112
サーバー接続マネージャ 112
マネージャ 112
リスナー 113
ディスパッチャ 113
以上のプロパティを使用するタイミング 114
アダプタアクティベータ 115
要求の処理 116

第 10 章

スレッドと接続の管理 117

スレッドの使い方 117
リスナースレッド，ディスパッチャスレッド，および作業スレッド 118
スレッドポリシー 118
スレッドプールポリシー 119
セッションごとのスレッドポリシー 122
接続管理 124
ServerEngines 124
ServerEngine のプロパティ 125
ディスパッチのポリシーとプロパティの設定 125
スレッドプールディスパッチポリシー 125
セッションごとのスレッドのディスパッチポリシー . 126
コーディングにおける留意点 126
接続管理プロパティの設定 127
適用できるプロパティの有効値 128
プロパティの変更の影響 128
動的に変更できるプロパティ 128
プロパティ値の変更が有効かどうかの確認 129
プロパティ値の変更による影響 129
ガベージコレクション 129

第 11 章

tie メカニズムの使い方 131

tie メカニズムのしくみ 131
サンプルプログラム 132
tie メカニズムを使用するサンプルプログラムの場所 132
tie テンプレートの概要 132
_tie_account クラスを使用するためのサーバーの変更 133
tie サンプルのビルド 134

第 12 章

クライアントの基礎 135

VisiBroker ORB の初期化 135
オブジェクトへのバインド 135
バインド処理中に実行される動作 136
オブジェクトのオペレーションの呼び出し 137
オブジェクトリファレンスの操作 137
nil リファレンスをチェックする 137
nil リファレンスを取得する 137
オブジェクトリファレンスを複製する 137
オブジェクトリファレンスを解放する 138
リファレンスカウントを取得する 138
リファレンスを文字列に変換する 139
オブジェクト名とインターフェース名を取得する . . 139
オブジェクトリファレンスの型を判定する 139
バインドされたオブジェクトの場所と状態を判定する . 140
存在しないオブジェクトをチェックする 140
オブジェクトリファレンスをナローイングする . . . 140
オブジェクトリファレンスをワイドニングする . . . 141
Quality of Service（QoS）の使用 141
Quality of Service（QoS）の概要 141
ポリシーオーバーライドと有効なポリシー 141
QoS のインターフェース 141
CORBA::Object 142
CORBA::Object 142
CORBA::PolicyManager 142
QoSExt::DeferBindPolicy 142
QoSExt::RelativeConnectionTimeoutPolicy . . 143
Messaging::RebindPolicy 143
Messaging::RelativeRequestTimeoutPolicy . . 144
Messaging::RelativeRoundTripTimeoutPolicy 145
Messaging::SyncScopePolicy 145
例外 145

第 13 章

IDL の使い方 147

IDL の概要 147
IDL コンパイラでコードを生成する方法 147
IDL 仕様のサンプル 148
クライアント用に生成されたコードの検討 148
IDL コンパイラが生成するメソッド（スタブ） 148
ポインタ型 <interface_name>_ptr の定義 149
自動メモリ管理 <interface_name>_var クラス . . . 149
サーバー用に生成されたコードの検討 150
IDL コンパイラが生成するメソッド（スケルトン） . 150
IDL コンパイラが生成するクラステンプレート . . . 150
IDL のインターフェース属性の定義 151
戻り値がない一方向メソッドの指定 151
別のインターフェースを継承する IDL インターフェースの指定 152

第 14 章

スマートエージェントの使い方 153

スマートエージェントの概要 153
スマートエージェントの設定と同期に関する推奨事項 153
全般的なガイドライン 154
負荷分散とフォールトトレランスのガイドライン 154
ロケーションサービスのガイドライン 155
スマートエージェントを使用しない場合 155
スマートエージェントの検索 155
スマートエージェント間の協力によるオブジェクトの検索 155
OAD との協力によるオブジェクトへの接続 156
スマートエージェント（osagent）の起動 156
詳細出力 157
エージェントを無効にする 157
スマートエージェントの有効性の確認 157
クライアントの確認 158
VisiBroker ORB ドメインの操作 158
異なるローカルネットワーク上のスマートエージェントの接続 159
スマートエージェントが互いを検出する方法 159
マルチホームホストのしくみ 160
スマートエージェントのインターフェースの用法の指定 161
ポイントツーポイント通信の使い方 162

実行時パラメータとしてのホストの指定 162
環境変数による IP アドレスの指定 162
agentaddr ファイルによるホストの指定 162
オブジェクトの有効性の確認 163
状態を保持しないオブジェクトのメソッドの呼び出し 163
状態を保持するオブジェクトのフォールトトレランスの実現 163
OAD に登録されたオブジェクトの複製 163
ホスト間のオブジェクトの移行 164
状態を保持するオブジェクトの移行 164
インスタンス化されたオブジェクトの移行 164
OAD に登録されたオブジェクトの移行 164
すべてのオブジェクトとサービスのレポート 165
オブジェクトへのバインド 165

第 15 章
ロケーションサービスの使い方 167

ロケーションサービスの概要 167
ロケーションサービスのコンポーネント 169
ロケーションサービスエージェントの概要 169
スマートエージェントが動作するすべてのホストのアドレスを取得する 170
アクセス可能なすべてのインターフェースを検索する 170
1 つのインターフェースの複数のインスタンスへのリファレンスを取得する 170
1 つのインターフェースの名前が同じ複数のインスタンスへのリファレンスを取得する 170
トリガーの概要 171
トリガーメソッド 171
トリガーの作成 171
トリガーによって検出される最初のインスタンス 172
エージェントの照会 172
1 つのインターフェースのすべてのインスタンスを検索する 172
スマートエージェントに既知のインターフェースとインスタンスを検索する 173
トリガーハンドラの書き込みと登録 175

第 16 章
VisiNaming サービスの使い方 179

概要 179
名前空間の理解 181
ネーミングコンテキスト 181
ネーミングコンテキストファクトリ 181
Name と NameComponent 182
名前の解決 182
文字列化された名前 182
単純な名前と複雑な名前 183
VisiNaming サービスの実行 183
VisiNaming サービスのインストール 183
VisiNaming サービスの設定 183
VisiNaming サービスの起動 184
コマンドラインからの VisiNaming サービスの起動 . . 185
nsutil の設定 185
nsutil の実行 185
nsutil を使用して VisiNaming サービスを閉じる . 186
VisiNaming サービスのブートストラップ 186
resolve_initial_references の呼び出し 186
-DSVCnameroot の使用 187
-ORBInitRef の使用 187
corbaloc URL の使用 187
corbaname URL の使用 187
-ORBDefaultInitRef 187
-ORBDefaultInitRef での corbaloc URL の使用 188
-ORBDefaultInitRef での corbaname の使用 . .188
NamingContext 188
NamingContextExt .189
デフォルトネーミングコンテキスト 189
デフォルトコンテキストの取得 189
ネーミングコンテキストファクトリの取得 189
VisiNaming サービスのプロパティ 190
取り替え可能なバックストア 193
バックストアの種類 193
インメモリアダプタ 193
JDBC アダプタ .193
DataExpress アダプタ 194
JNDI アダプタ .194
設定と使用 194
プロパティファイル 195
JDBC アダプタのプロパティ 195
DataExpress アダプタのプロパティ 197
JNDI アダプタのプロパティ 197
OpenLDAP の設定 197
キャッシング機能 197
キャッシング機能に関する重要事項 198
クラスタ 199
クラスタリングの基準 199
Cluster と ClusterManager インターフェース . . .199
Cluster インターフェースの IDL 仕様 199
ClusterMangager インターフェースの IDL 仕様 200
NamingContextExtExtended インターフェースの IDL 仕様 200
クラスタの作成 .201
明示的クラスタと暗黙的クラスタ 201
負荷分散 202
オブジェクトのフェイルオーバー 202
VisiNaming オブジェクトクラスタ内の無効なオブジェクトリファレンスの削除 203
VisiNaming サービスクラスタによるフェイルオーバーと負荷分散 203
VisiNaming サービスクラスタの設定 204
マスター／スレーブモードでの VisiNaming サービスの設定 205
多数のクライアントが接続する環境での起動 205
VisiNaming サービスフェデレーション 206
VisiNaming サービスのセキュリティ 207
ネーミングクライアント認証 208
SSL を使用するように VisiNaming を設定する . . .208
メソッドレベル承認 209
プログラムのコンパイルとリンク 210
サンプルプログラム 210
VisiNaming で使用する JdataStore HA の設定 211
プライマリミラーの DB を作成する 211
各リスニング接続について JdsServer を呼び出す . .211
JDataStore HA を設定する 212
VisiNaming の明示的クラスタリングの例を実行する 212
VisiNaming のネーミングフェイルオーバーの例を実行する 214

第 17 章
イベントサービスの使い方 217
概要 217
プロキシコンシューマおよびプロキシサプライヤ . . 218
OMG コモンオブジェクトサービス仕様 219
通信モデル 219
プッシュモデル 220
プルモデル 220
イベントチャネルの使い方 221
イベントチャネルの作成 222
プッシュサプライヤおよびコンシューマのサンプル . . . 222
プッシュサプライヤ／コンシューマサンプル 222
PushSupplier クラスの派生 222
PushSupplier の実装 223
サンプルプッシュサプライヤの完全なインプリメンテーション 225
PushConsumer クラスの派生 228
PushConsumer の実装 229
キューの長さの設定 231
プログラムのコンパイルとリンク 231

第 18 章
VisiBroker サーバーマネージャの使用 233
サーバーマネージャの概要 233
サーバーでのサーバーマネージャの有効化 233
サーバーマネージャリファレンスの取得 234
コンテナの使用 234
Storage インターフェース 234
Container インターフェース 235
Container のメソッド 235
プロパティの操作とクエリーに関連するメソッド 235
オペレーションに関連するメソッド 236
子コンテナに関連するメソッド 236
ストレージに関連するメソッド 236
Storage インターフェース 236
Storage インターフェースのメソッド 237
サーバーマネージャに対するアクセスの制限 237
サーバーマネージャ IDL 237
サーバーマネージャの例 239
最上位コンテナへのリファレンスの取得 240
すべてのコンテナとそれらのプロパティの取得 . . . 240
プロパティの取得と設定，およびファイルへの保存 . 240
Container でのオペレーションの呼び出し 241
カスタムコンテナ 241

第 19 章
VisiBroker ネイティブメッセージングの使用 243
はじめに 243
2 フェーズ呼び出し（2PI） 243
ポーリング／プルモデルとコールバックモデル . . . 244
非ネイティブメッセージングと IDL の変形 244
ネイティブメッセージングソリューション 244
リクエストエージェント 245
ネイティブメッセージングの Current オブジェクト . 245
コアオペレーション 246
StockManager サンプル 246
ポーリング／プルモデル 247
コールバックモデル 248
高度な項目 251
グループポーリング 251
応答受信者における Cookie と応答逆多重化 252
2 フェーズ呼び出しへの展開 253
応答ドロップ 255
コレクションの手動破棄 255
非抑制早期リターンモード 256
コールバックモデルでのポーラー生成の抑制 257
ネイティブメッセージングの API 仕様 257
RequestAgentEx インターフェース 257
create_request_proxy() 258
destroy_request() 258
RequestProxy インターフェース 259
the_receiver 259
poll() 259
destroy() 260
ローカルインターフェースの Current オブジェクト 260
suppress_mode() 260
wait_timeout 260
the_cookie 260
request_tag 260
the_poller 261
reply_not_available 261
ReplyRecipient インターフェース 262
reply_available() 262
コアオペレーションのセマンティクス 263
ネイティブメッセージングの相互運用性仕様 263
ネイティブメッセージングはネイティブ GIOP を使用 263
ネイティブメッセージングのサービスコンテキスト . 264
NativeMessaging タグ付きコンポーネント 265
Borland ネイティブメッセージングの使用 265
リクエストエージェントとクライアントモデルの使用 265
Borland リクエストエージェントの起動 265
Borland リクエストエージェントの URL 265
Borland ネイティブメッセージングクライアントモデルの使用 266
Borland リクエストエージェントの vbroker プロパティ 266
vbroker.requestagent.maxThreads 266
vbroker.requestagent.maxOutstandingRequests 266
vbroker.requestagent.blockingTimeout 266
vbroker.requestagent.router.ior 266
vbroker.requestagent.listener.port 266
vbroker.requestagent.requestTimeout 266
CORBA メッセージング との相互運用性 267
以前のバージョンの VisiBroker ネイティブメッセージングからの移行 267
以前のバージョンの VisiBroker ネイティブメッセージングからの移行 268

第 20 章
オブジェクトアクティベーションデーモン（OAD）の使い方 269
オブジェクトとサーバーの自動アクティブ化 269
インプリメンテーションリポジトリデータの検索 . . 269
サーバーのアクティブ化 270
OAD の使い方 270
OAD の起動 270
OAD ユーティリティの使い方 271
インターフェース名をリポジトリ ID に変換する . . 271

oadutil リストによるオブジェクトの一覧表示 . . . 272
oadutil によるオブジェクトの登録 273
例：リポジトリ ID の指定 274
例：IDL インターフェース名の指定 274
OAD にリモートで登録する 274
スマートエージェントを使用しない OAD の使用 275
ネーミングサービスによる OAD の使用 275
オブジェクトの複数のインスタンスの区別 276
CreationImplDef クラスによるアクティブ化プロパティの設定 276
ORB インプリメンテーションを動的に変更する . . 276
OAD::reg_implementation による OAD 登録 . . 277
OAD で渡す引数 277
オブジェクトの登録解除 277
oadutil ツールによるオブジェクトの登録解除 . . . 278
登録解除のサンプル 278
OAD からの登録解除の操作 278
インプリメンテーションリポジトリの内容の表示 . . 279
OAD の IDL インターフェース 279

第 21 章

インターフェースリポジトリの使い方 281

インターフェースリポジトリの概要 281
インターフェースリポジトリの内容 282
作成できるインターフェースリポジトリの数 282
irep を使ったインターフェースリポジトリの作成と表示 282
irep を使ったインターフェースリポジトリの作成 . 283
インターフェースリポジトリの内容の表示 283
idl2ir を使ったインターフェースリポジトリの更新 . . 284
インターフェースリポジトリの構造体の概要 284
インターフェースリポジトリ内のオブジェクトの識別 285
インターフェースリポジトリに保存できるオブジェクトの型 285
継承元のインターフェース 286
インターフェースリポジトリへのアクセス 286
インターフェースリポジトリのサンプルプログラム . . 287

第 22 章

動的起動インターフェースの使い方 289

動的起動インターフェースの概要 289
重要な DII の概念 290
Request オブジェクトの使用 290
Any 型を使った引数のカプセル化 291
要求の送信オプション 291
応答の受信オプション 292
オブジェクトのオペレーションを動的に呼び出すための手順 292
DII を使用するサンプルプログラム 292
共通オブジェクトリファレンスの取得 292
要求の作成と初期化 293
Request クラス 293
DII 要求を作成および初期化する方法 294
create_request メソッドの使い方 294
_request メソッドの使い方 294
Request オブジェクトを作成するサンプルコード . 294
要求のコンテキストの設定 295
要求に引数を設定する方法 295
NVList クラスを使用して，引数リストを実装する 295
NamedValue クラスを使用して，入力引数と出力引数を設定する 296
Any クラスを使ってタイプセーフに引数を渡す . . .296
TypeCode クラスを使用して，引数または属性の型を表す 297
DII 要求の送信と結果の受信 298
要求を呼び出す .298
send_deferred メソッドを使用して，遅延 DII 要求を送信する 299
send_oneway メソッドを使用して，非同期 DII 要求を送信する 299
複数の要求を送信する 299
複数の要求を受信する 300
DII によるインターフェースリポジトリの使い方 301

第 23 章

動的スケルトンインターフェースの使い方 305

動的スケルトンインターフェースの概要 305
オブジェクトインプリメンテーションを動的に作成するための手順 306
DSI を使用するサンプルプログラム 306
DynamicImplementation クラスの拡張 306
動的要求のオブジェクトを設計するサンプル 306
リポジトリ ID の指定 308
ServerRequest クラスについて 308
Account オブジェクトの実装 309
AccountManager オブジェクトの実装 309
入力パラメータの処理 309
戻り値の設定 310
サーバーインプリメンテーション 310

第 24 章

ポータブルインターセプタの使い方 313

ポータブルインターセプタの概要 313
インターセプタの種類 314
ポータブルインターセプタの種類 314
ポータブルインターセプタと情報インターフェース . . .314
Interceptor クラス 314
リクエストインターセプタ 315
ClientRequestInterceptor 315
クライアント側の規則 316
ServerRequestInterceptor 316
サーバー側の規則 317
IOR インターセプタ 317
ポータブルインターセプタ（PI）Current 318
Codec 318
CodecFactory 318
ポータブルインターセプタの作成 318
例：PortableInterceptor の作成 319
ポータブルインターセプタの登録 319
ORBInitializer の登録 320
例：ORBInitializer の登録 320
VisiBroker によるポータブルインターセプタの拡張機能 321
POA スコープ付きサーバーリクエストインターセプタ 321
VisiBroker ポータブルインターセプタのインプリメンテーションの制限 321
ClientRequestInfo の制限 321

ServerRequestInfo の制限 322
ポータブルインターセプタの概要 322
例 : client_server 322
サンプルの目的 322
必要なパッケージのインポート 322
クライアント側リクエストインターセプタの初期化と ORB への登録 323
サーバー側インターセプタの ORBInitializer の実装 325
クライアント側またはサーバー側のリクエストインターセプタの RequestInterceptor の実装 327
クライアントの ClientRequestInterceptor 実装 . . 327
public void send_request(ClientRequestInfo ri) インターフェースのインプリメンテーション . . 328
void send_poll(ClientRequestInfo ri) インターフェースのインプリメンテーション 328
void receive_reply(ClientRequestInfo ri) インターフェースのインプリメンテーション 328
void receive_exception(ClientRequestInfo ri) インターフェースのインプリメンテーション 328
void receive_request_service_contexts (ServerRequestInfo ri) インターフェースのインプリメンテーション 331
void receive_request (ServerRequestInfo ri) インターフェースのインプリメンテーション 331
void receive_reply (ServerRequestInfo ri) インターフェースのインプリメンテーション 331
void receive_exception (ServerRequestInfo ri) インターフェースのインプリメンテーション . . . 331
void receive_other (ServerRequestInfo ri) インターフェースのインプリメンテーション 332
クライアントおよびサーバーアプリケーションの開発 . 334
クライアントアプリケーションのインプリメンテーション 334
サーバーアプリケーションのインプリメンテーション 335
コンパイルの手順 336
クライアントアプリケーションとサーバーアプリケーションの実行と配布 337

第 25 章 VisiBroker インターセプタの使い方 339

インターセプタの概要 339
インターセプタインターフェースとインターセプタマネージャ 340
クライアントインターセプタ 340
BindInterceptor 340
ClientRequestInterceptor 341
サーバーインターセプタ 341
POALifeCycleInterceptor 341
ActiveObjectLifeCycleInterceptor 342
ServerRequestInterceptor 342
IORCreationInterceptor 342
サービスリゾルバインターセプタ 343
VisiBroker ORB によるインターセプタの登録 . . . 343
インターセプタオブジェクトの作成 344
インターセプタのロード 344
サンプルインターセプタ 344
サンプルコード 344
クライアント／サーバーインターセプタのサンプル 344
コードリスト 346
SampleServerLoader 346
SamplePOALifeCycleInterceptor 346
SampleServerInterceptor 347
SampleClientInterceptor 348
SampleClientLoader 349
SampleBindInterceptor 350
インターセプタ間の情報の受け渡し 351
ポータブルインターセプタと VisiBroker インターセプタの同時使用 351
インターセプタの呼び出しポイントの順序 351
クライアント側インターセプタ 351
サーバー側インターセプタ 352
POA 作成中の ORB イベントの順序 352
POA リファレンス作成中の ORB イベントの順序 . 352

第 26 章 オブジェクトラッパーの使い方 355

オブジェクトラッパーの概要 355
型付きと型なしのオブジェクトラッパー 356
idl2cpp の特殊な要件 356
Object ラッパーのサンプルアプリケーション 356
型なしオブジェクトラッパー 356
複数の型なしオブジェクトラッパーの使い方 357
pre_method 呼び出しの順序 357
post_method 呼び出しの順序 357
型なしオブジェクトラッパーの使い方 358
型なしオブジェクトラッパーファクトリの実装 . . . 358
型なしオブジェクトラッパーのインプリメンテーション 359
pre_method メソッドと post_method メソッドに共通の引数 359
型なしオブジェクトラッパーファクトリの作成と登録 . 360
型なしオブジェクトラッパーの削除 361
型付きオブジェクトラッパー 361
複数の型付きオブジェクトラッパーの使い方 362
呼び出しの順序 362
同じ場所にあるクライアント／サーバーの型付きオブジェクトラッパー 363
型付きオブジェクトラッパーの使い方 363
型付きオブジェクトラッパーの実装 363
クライアント向け型付きオブジェクトラッパーの登録 364
サーバー向けの型付きオブジェクトラッパーの登録 . 365
型付きオブジェクトラッパーの削除 366
型なしラッパーと型付きラッパーの複合的な使い方 . . . 366
型付きラッパーのコマンドライン引数 366
型付きラッパーのイニシャライザ 366
型なしラッパーのコマンドライン引数 368
型なしラッパーのイニシャライザ 368
サンプルアプリケーションの実行 369
トレースおよび時間測定のオブジェクトラッパーをオンにする 369
キャッシュとセキュリティのオブジェクトラッパーをオンにする 369
型付きラッパーと型なしラッパーをオンにする . . 369
共用クライアント／サーバーを実行する 370

第 27 章 イベントキュー 371

イベントタイプ 371
接続イベント 371
イベントリスナー 371

IDL 定義 371
ConnInfo 構造体 372
EventListener インターフェース 372
ConnEventListeners インターフェース 372
EventQueueManager インターフェース 373
EventQueueManager を返す方法 373
イベントキューのサンプルコード 373
EventListener の登録 373
EventListener の実装 374

第 28 章

動的に管理される型の使い方 377

DynAny インターフェースの概要 377
DynAny サンプル 377
DynAny 型 377
DynAny 使用上の制限 378
DynAny の作成 378
DynAny の値の初期化とアクセス 378
構造データ型 379
構造データ型内の複数のコンポーネント間の移動 . . 379
DynEnum 379
DynStruct 379
DynUnion 379
DynSequence と DynArray 379
DynAny サンプル IDL 380
DynAny サンプルクライアントアプリケーション . . . 380
DynAny サンプルサーバーアプリケーション 381

第 29 章

valuetype の使い方 387

valuetype について 387
valuetype IDL サンプルコード 387
具象 valuetype 387
valuetype の派生 388
共有セマンティクス 388
null セマンティクス 388
ファクトリ 388
抽象 valuetype 388
valuetype の実装 388
valuetype の定義 389
IDL ファイルのコンパイル 389
valuetype 基底クラスの継承 389
Factory クラスの実装 390
ファクトリを VisiBroker ORB に登録 390
ファクトリの実装 390
ファクトリと valuetype 391
valuetype の登録 391
ボックス化 valuetype 391
抽象インターフェース 391
custom valuetype 392
truncatable valuetype 392

第 30 章

双方向通信 395

双方向 IIOP の使用 395
双方向 VisiBroker ORB のプロパティ 395
双方向サンプルについて 396
既存のアプリケーションで双方向 IIOP を有効にする . 397
双方向 IIOP を明示的に有効にする 397
一方向または双方向接続 398
POA で双方向 IIOP を有効にする 398
セキュリティに関する注意 399

第 31 章

VisiBroker における BOA の使い方 401

VisiBroker を使った BOA コードのコンパイル 401
BOA オプションのサポート 401
オブジェクトアクティベータの使い方 401
BOA の下でのネーミングオブジェクト 402
オブジェクト名 .402

第 32 章

オブジェクトアクティベータの使い方 403

オブジェクトのアクティブ化の遅延 403
Activator インターフェース 403
サービスのアクティブ化の使い方 404
サービスアクティベータを使ってオブジェクトのアクティブ化を遅延する 404
サービスを使ってオブジェクトのアクティブ化を遅延するサンプル .405
odb.idl インターフェース 405
サービスアクティブ化オブジェクトの実装 406
サービスアクティベータの実装 406
サービスアクティベータのインスタンス化 407
サービスアクティベータを使ったオブジェクトのアクティブ化 .407
サービスアクティブ化オブジェクトインプリメンテーションの非アクティブ化 408

第 33 章

リアルタイム CORBA 拡張 411

概要 411
リアルタイム CORBA 拡張の使用 412
リアルタイム CORBA ORB 413
リアルタイムオブジェクトアダプタ 414
リアルタイム CORBA 優先順位 415
優先順位マッピング .416
優先順位マッピングの種類 417
優先順位マッピングの規則 418
デフォルトの優先順位マッピング 418
デフォルト以外の優先順位マッピングの使用 420
VisiBroker アプリケーションコードでのネイティブ優先順位の使用 421
スレッドプール 422
スレッドプール API 422
スレッドプールの作成および設定 423
オブジェクトアダプタとスレッドプールの関連付け .423
汎用スレッドプール 424
スレッドプールの破棄 425
リアルタイム CORBA Current 425
リアルタイム CORBA 優先順位モデル 427
オブジェクトレベルでの優先順位の設定 428
リアルタイム CORBA ミューテックス API 429
内部 ORB スレッド優先順位の制御 429
個別内部 ORB スレッドの優先順位の設定 430
内部 ORB スレッドの優先順位範囲の制限 431

第 34 章

CORBA 例外 433

CORBA 例外の説明 433
OMG 指定のヒューリスティックな例外 437
OMG 指定のその他の例外 438

第 35 章

VisiBroker プラグイン可能トランスポートインターフェース 441

プラグイン可能トランスポートインターフェースファイル 441
トランスポート層の要件 442
プロトコルプラグインに必要なユーザー提供コード . . . 442
固有のプロファイル ID タグ 443
サンプルコード 443
新しいトランスポートの実装 444
VISPTransConnection と VISPTransConnectionFactory 444
VISPTransListener と VISPTransListenerFactory . 445
VISPTransProfileBase と VISPTransProfileFactory . 446
追加クラス - VISPTransBridge と VISPTransRegistrar . 446

第 36 章

VisiBroker ログ 449

ログの概要 449
ロガーマネージャ 451
ログ 451
フィルタリング 452
予約名 453
カスタマイズ 453
設定 455
ログマネージャの設定 455
アペンダとレイアウトの登録設定 456
ロガーでのアペンダとレイアウトの設定 456
フィルタ設定 457
プロパティの設定 457

第 37 章

Web サービスの概要 459

Web サービスアーキテクチャ 459
標準 Web サービスアーキテクチャ 460
VisiBroker Web サービスアーキテクチャ 460
Web サービス関連ファイル 460
Web サービスランタイム 461
Web サービスとしての CORBA オブジェクトの公開 . . 463
開発 464
IDL からの WSDL の生成 464
C++ インターフェース型固有のブリッジの生成 . . . 464
配布 465
配布 WSDD の作成 465
作成された WSDD を使用して配布します。 465
axiscpp.conf ファイルのサンプル 465
Web サービスランタイムの設定 466
WSDD リファレンス 466
制限 467
SOAP/WSDL の互換性 467

索引 469

第 1 章

Borland VisiBroker の概要

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

VisiBroker の概要

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

VisiBroker は次のコンポーネントからなります。

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

VisiBroker の機能

VisiBroker には次の機能があります。

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

VisiBroker のマニュアル

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

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

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

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

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

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

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

Windows

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

```
help
```

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

```
help
```

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

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

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

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

マニュアルの表記規則

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

表 1.1　マニュアルの表記規則

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

プラットフォームの表記

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

表 1.2　プラットフォームの表記

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

Borland サポートへの連絡

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

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

ボーランド社のサポートへの連絡にあたっては，次の情報を用意してください。

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

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

オンラインリソース

ネットワーク上の次のサイトから情報を得ることができます。

Web サイト http://www.borland.com/jp/

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

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

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

Web サイト

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

特に，次の URL をチェックすることをお勧めします。

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

Borland ニュースグループ

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

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

第 2 章

CORBA モデルの概要

ここでは VisiBroker を紹介します。これは，VisiBroker for C++ と VisiBroker for Java の両方の ORB で構成されます。どちらも CORBA 2.6 仕様の完全なインプリメンテーションです。この章では，VisiBroker の機能とコンポーネントについて説明します。

CORBA の概要

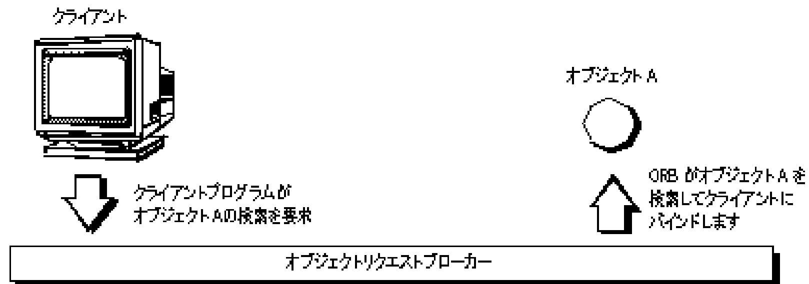
コモンオブジェクトリクエストブローカーキテクチャ（Common Object Request Broker Architecture，CORBA）を利用すると，記述された言語や存在する場所に関係なく，分散アプリケーションどうしの相互運用が可能になります。

CORBA 仕様は，分散オブジェクトアプリケーション開発の複雑さとコストの低減を目的として，オブジェクトマネージメントグループ（Object Management Group）によって採用されました。CORBA では，オブジェクト指向手法で，アプリケーション間の再利用と共有が可能なソフトウェアコンポーネントを作成します。各オブジェクトは内部の詳細機能をカプセル化し，明確に定義されたインターフェースを提示します。このインターフェースを利用することで，アプリケーションの複雑さが緩和されます。インターフェース自体も，標準のインターフェース定義言語（IDL）で記述されています。このインターフェースにより，アプリケーションの複雑さが緩和されます。いったんオブジェクトを実装してテストすれば，そのオブジェクトを繰り返し使用できるため，アプリケーションの開発コストも節約できます。

オブジェクトリクエストブローカー（ORB）の役割は，これらのインターフェースを追跡および管理し，インターフェース間の通信を円滑化し，インターフェースを利用するアプリケーションにサービスを提供することです。ORB 自体は独立したプロセスではなく，エンドユーザーのアプリケーション内に統合されたライブラリとネットワークリソースの集合です。クライアントアプリケーションは，この ORB を利用してさまざまなオブジェクトを検索して使用します。

次の図のオブジェクトリクエストブローカーは，クライアントアプリケーションをそのアプリケーションが使用するオブジェクトに接続します。クライアントアプリケーションにとって，探しているオブジェクトが同じコンピュータ上に存在するのか，ネットワーク上のリモートコンピュータ上に存在するのかを知る必要がありません。クライアントアプリケーション側に必要な情報は，オブジェクトの名前とオブジェクトのインターフェースの使い方だけです。オブジェクトの検索，要求の転送，結果の返信などの詳細は，すべて ORB によって処理されます。

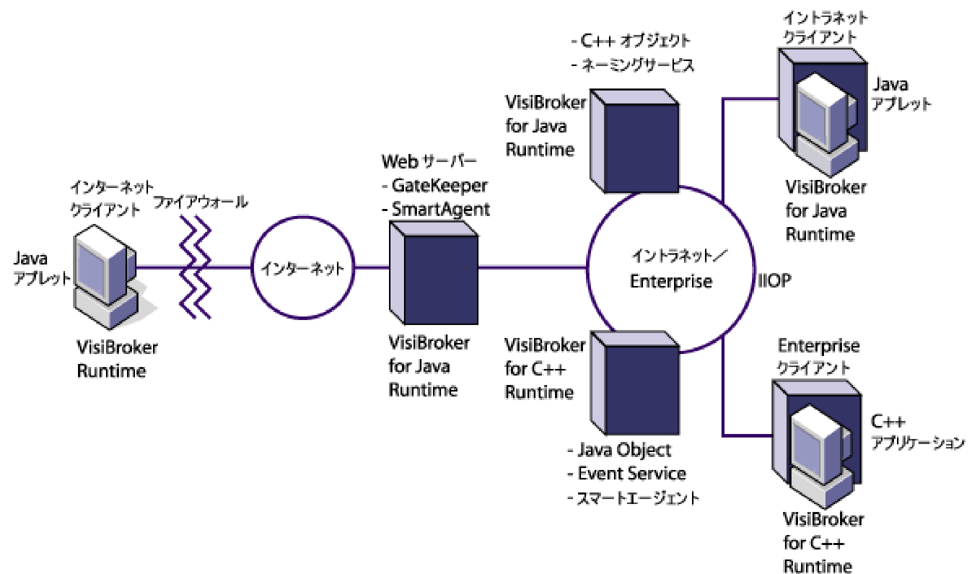
図 2.1　オブジェクトを操作するクライアントプログラム



VisiBroker の概要

VisiBroker は，完全な CORBA 2.6 ORB 実行時環境を提供します。また，オープンで柔軟性があり相互運用が可能な C++ と Java に対して，分散したアプリケーションを構築，配布，および管理する開発環境をサポートします。VisiBroker で構築したオブジェクトは，Internet Inter-ORB プロトコル（IIOP）標準で通信する Web ベースアプリケーションから簡単にアクセスできます。IIOP は，インターネットやローカルイントラネットを介して分散オブジェクト間で通信するためのプロトコルです。VisiBroker は IIOP のビルトインインプリメンテーションを搭載しており，高いパフォーマンスと相互運用性を保証します。

図 2.2　VisiBroker のアーキテクチャ



VisiBroker の機能

次に，VisiBroker の主な機能について説明します。

VisiBroker のスマートエージェント（osagent）アーキテクチャ

VisiBroker のスマートエージェント（osagent）は，動的な分散ディレクトリサービスであり，クライアントプログラムとオブジェクトインプリメンテーションにネーミング機能を提供します。ネットワークにある複数のスマートエージェントは協調して機能し，クライアントからサーバーオブジェクトへのアクセスの負荷を分散し，可用性を高めます。スマートエージェントは，ネットワーク上で利用できるオブジェクトを追跡し，クライアントアプリケーションからのオブジェクトの呼び出しに応じてオブジェクトを検索します。VisiBroker は，（サーバーのクラッシュやネットワーク障害などのエラーによって）クライアントアプリケーションとサーバーオブジェクトの間の接続が失われていないかどうかを判定できます。エラーが検出されると，設定によっては異なるホスト上の別のサーバーにクライアントを接続します。スマートエージェントの詳細については，第 14 章「スマートエージェントの使い方」と「Quality of Service（QoS）の使用」を参照してください。

ロケーションサービスを使った高度なオブジェクト検索機能

VisiBroker には CORBA 仕様を拡張した強力なロケーションサービスが搭載されています。この機能では，複数のスマートエージェントの情報を利用できます。ロケーションサービスはネットワーク上の複数のスマートエージェントと連係し，クライアントがバインドできる有効なオブジェクトのインスタンスをすべて参照します。クライアントアプリケーションはコールバックメカニズムである**トリガー**を使用して，オブジェクト可用性の変化を知ることができます。ロケーションサービスを**インターセプタ**と組み合わせて使用することで，クライアントからサーバーオブジェクトへの要求をパワフルに負荷分散できる機能を開発できます。第 15 章「ロケーションサービスの使い方」を参照してください。

インプリメンテーションとオブジェクトのアクティブ化のサポート

オブジェクトアクティベーションデーモン（OAD）は，VisiBroker によるインプリメンテーションリポジトリのインプリメンテーションです。OAD では，クライアントがオブジェクトを使用する必要が発生すると自動的にオブジェクトインプリメンテーションを起動できます。また，VisiBroker にはクライアント要求が受信されるまでオブジェクトのアクティブ化を遅らせる機能も搭載されており，特定オブジェクトや，サーバー上にあるオブジェクトの全クラスのアクティブ化を遅らせることができます。

堅牢なスレッドと接続の管理

VisiBroker では，シングルスレッドとマルチスレッドのスレッド管理をネイティブサポートしています。VisiBroker のセッション単位モデルでは，各クライアントへのサーバー接続ごとにスレッドが自動的に割り当てられ，複数の要求をサービスします。各接続が終了するとスレッドも終了します。スレッドプーリングモデルでは，サーバーオブジェクトへの／からの要求トラフィックの量に基づいてスレッドが割り当てられます。つまり，ビジーなクライアントには，要求をスピーディに実行できるように複数のスレッドが用意され，あまりビジーでないクライアントは別のクライアントとスレッドを共有します。スレッドを共有しても，要求はすぐに処理されます。

VisiBroker の接続管理は，クライアントからサーバーへの接続数を最小限に抑えます。同じサーバー上のオブジェクトに対するすべてのクライアント要求は，異なるスレッドから生じたものであっても，同じ接続上に多重化されます。また，解放されたクライアント接

続は後で同じサーバーに再接続する際に再利用できるため，クライアントが同じサーバーへ新しく接続するためのオーバーヘッドをカットできます。

スレッドと接続の動作は詳細に設定できます。VisiBroker によるスレッドと接続の管理方法の詳細については，第 10 章「スレッドと接続の管理」を参照してください。

IDL コンパイラ

VisiBroker には，オブジェクト開発を支援するため，3 つの IDL コンパイラが組み込まれています。

- idl2java：このコンパイラは，IDL ファイルを入力として，必要なクライアントスタブとサーバースケルトンを Java で生成します。
- idl2cpp：このコンパイラは，IDL ファイルを入力として，必要なクライアントスタブとサーバースケルトンを C++ で生成します。
- idl2ir：このコンパイラは，IDL ファイルを入力として，インターフェースリポジトリに内容を挿入します。上の 2 つのコンパイラとは異なり，idl2ir は，C++ ORB と Java ORB の両方で機能します。

以上のコンパイラの詳細については，第 13 章「IDL の使い方」と第 21 章「インターフェースリポジトリの使い方」を参照してください。

DII と DSI による動的起動

VisiBroker は，動的起動インターフェース（DII）および動的起動のための動的スケルトンインターフェース（DSI）の両方のインプリメンテーションを提供します。DII を使用すると，コンパイル時に定義されてなかったオブジェクトに対する要求をクライアントアプリケーションで動的に作成できます。DSI を使用すると，コンパイル時に定義されていなかったオブジェクトに対するクライアントオペレーションリクエストをサーバーでディスパッチできます。詳細については，第 22 章「動的起動インターフェースの使い方」と第 23 章「動的スケルトンインターフェースの使い方」を参照してください。

インターフェースとインプリメンテーションリポジトリ

インターフェースリポジトリ（IR）は，ORB オブジェクトに関するメタ情報のオンラインデータベースです。オブジェクトに保存されるメタ情報には，モジュール，インターフェース，処理，属性，および例外があります。インターフェースリポジトリインスタンスの開始方法，IDL ファイルからインターフェースリポジトリに情報を追加する方法，およびインターフェースリポジトリから情報を抽出する方法については，第 21 章「インターフェースリポジトリの使い方」を参照してください。

オブジェクトアクティベーションデーモン（OAD）は，インプリメンテーションリポジトリに対する VisiBroker のインターフェースです。クライアントがオブジェクトを参照するとき，そのインプリメンテーションを自動的にアクティブ化します。詳細については，第 20 章「オブジェクトアクティベーションデーモン（OAD）の使い方」を参照してください。

サーバー側の可搬性

VisiBroker は，基本オブジェクトアダプタ（BOA）のかわりとして CORBA ポータブルオブジェクトアダプタ（POA）をサポートしています。POA は，オブジェクトのアクティブ化，一時的または永続的オブジェクトのサポートなど，BOA と同じ機能を一部共有しています。また，POA には POA マネージャやサーバントマネージャなどの追加機能も搭載されています。これらはオブジェクトのインスタンスを作成および管理します。詳細については，第 9 章「POA の使い方」を参照してください。

インターセプタとオブジェクトラッパーを使った ORB のカスタマイズ

VisiBroker のインターセプタを使用して，開発者はクライアントとサーバー間がバックグラウンドで行っている通信を監視できます。VisiBroker のインターセプタは，Borland 独自のインターセプタです。インターセプタを使用すると，クライアントとサーバーのコードをカスタマイズして，VisiBroker ORB を拡張できます。それらのコードでは，負荷分散，監視，セキュリティなどの機能によって分散アプリケーションの特殊な要求に応えることができます。第 24 章「ポータブルインターセプタの使い方」を参照してください。

VisiBroker には OMG の標準機能をベースにしたポータブルインターセプタも搭載されているため，移植性のあるインターセプタコードを記述して異なるベンダーの ORB でコードを使用できます。詳細については，「*COBRA 2.6 仕様*」を参照してください。

VisiBroker のオブジェクトラッパー機能を使用すると，バインドしたオブジェクトのメソッドをクライアントアプリケーションが呼び出す際，またはサーバーアプリケーションがオペレーションリクエストを受け取る際に呼び出すメソッドを定義できます。第 26 章「オブジェクトラッパーの使い方」を参照してください。

イベントキュー

イベントキューは，サーバー側だけの機能として設計されています。サーバーは，サーバーが必要とするイベントタイプに基づいてリスナーをイベントキューに登録しておき，サーバーは必要なときにそのイベントを処理することができます。詳細については，第 27 章「イベントキュー」を参照してください。

ネーミングサービスのバックストア

新機能の相互運用可能なネーミングサービスは，取り替え可能なバックストアを統合して，その状態を永続化します。これにより，ネーミングサービスにおけるフォールトトレランスとフェイルオーバーが容易になります。詳細については，第 16 章「VisiNaming サービスの使い方」を参照してください。

GateKeeper

GateKeeper を使用すると，クライアントプログラムは Web ブラウザによるセキュリティ制約の下で，Web サーバー上のオブジェクトにオペレーションリクエストを発行し，それらのオブジェクトからコールバックを受け取ることができます。また，Gatekeeper はファイアウォールを介した通信も処理でき，HTTP デーモンとしても使用できます。Gatekeeper は，OMG CORBA ファイアウォール仕様に完全に準拠しています。詳細については，『VisiBroker Gatekeeper の概要』を参照してください。

VisiBroker CORBA 準拠

VisiBroker は，オブジェクトマネージメントグループ（Object Management Group）の CORBA 仕様（バージョン 2.6）に完全に準拠しています。詳細については，http://www.omg.org/ の CORBA 仕様を参照してください。

VisiBroker 開発環境

VisiBroker は，開発と配布の両方に使用できます。開発環境には，次のコンポーネントがあります。

- 管理ツールとプログラミングツール

- VisiBroker ORB

プログラマツール

開発段階では，次のツールを使用します。

ツール	使途
idl2ir	このツールでは，VisiBroker for Java と VisiBroker for C++ の両方の IDL ファイルで定義されたインターフェースにインターフェースリポジトリの内容を代入できます。
idl2cpp	IDL ファイルから C++ スタブとスケルトンを生成します。
idl2java	IDL ファイルから Java スタブとスケルトンを生成します。
java2iiop	Java ファイルから Java スタブとスケルトンを生成します。このツールを利用すると，IDL ではなく，Java でインターフェースを定義できます。
java2idl	Java バイトコードを含むファイルから IDL ファイルを生成します。

CORBA サービスツール

次のツールでは，開発時の VisiBroker ORB を管理します。

ツール	使途
irep	インターフェースリポジトリの管理に使用します。第 21 章「インターフェースリポジトリの使い方」を参照してください。
oad	オブジェクトアクティベーションデーモン（OAD）の管理に使用します。第 20 章「オブジェクトアクティベーションデーモン（OAD）の使い方」を参照してください。
nameserv	ネーミングサービスのインスタンスの起動に使用します。第 16 章「VisiNaming サービスの使い方」を参照してください。

管理ツール

次のツールでは，開発時の VisiBroker ORB を管理します。

ツール	使途
oadutil list	OAD に登録されている VisiBroker ORB オブジェクトインプリメンテーションをリストします。
oadutil reg	OAD に VisiBroker ORB オブジェクトインプリメンテーションを登録します。
oadutil unreg	OAD の VisiBroker ORB オブジェクトインプリメンテーションを登録解除します。
osagent	スマートエージェントの管理に使用します。第 14 章「スマートエージェントの使い方」を参照してください。
osfind	特定のネットワークで実行中のオブジェクトを報告します。

VisiBroker との相互運用性

VisiBroker for Java で作成したアプリケーションは，VisiBroker for C++ で開発したオブジェクトインプリメンテーションと通信できます。同様に，VisiBroker for C++ で作成したアプリケーションは，VisiBroker for Java で開発したオブジェクトインプリメンテーションと通信できます。たとえば，VisiBroker for C++ で Java アプリケーションを使用する場合，VisiBroker for C++ に付属する IDL コンパイラに Java アプリケーションの開発に使用するものと同じ IDL を入力します。次に，生成した C++ スケルトンを使ってオブジェクトインプリメンテーションを開発します。VisiBroker for Java 上で C++ アプリケーションを使用する場合も，この処理を繰り返しますが，VisiBroker for Java のかわりに VisiBroker IDL コンパイラをかわりに使用します。

また，VisiBroker for Java で記述したオブジェクトインプリメンテーションは，VisiBroker for C++ で記述したクライアントに使用できます。実際には，VisiBroker for Java で記述したサーバーは CORBA 準拠の任意のクライアントと相互機能し，VisiBroker for Java で記述したクライアントは CORBA 準拠の任意のサーバーと相互機能します。これは，VisiBroker for C++ のどのオブジェクトインプリメンテーションにも当てはまります。

ほかの ORB 製品

CORBA 準拠のソフトウェアオブジェクトどうしは Internet Inter-ORB プロトコル（IIOP）通信により，完全に相互運用できます。これらのソフトウェアオブジェクトは，互いのインプリメンテーションの知識がないベンダーによって開発されたものであっても問題ありません。VisiBroker は IIOP を使用しているため，VisiBroker で開発したクライアントアプリケーションとサーバーアプリケーションは，ほかのベンダーのさまざまな ORB 製品と相互運用できます。

IDL から C++ へのマッピング

VisiBroker は，『*OMG IDL/C++ Language Mapping Specification*』に準拠しています。`idl2cpp` コンパイラに実装されている VisiBroker の現在の IDL/C++ 言語マッピングの概要については，『*VisiBroker プログラマーズリファレンス*』を参照してください。IDL の各構造ごとに対応する C++ の構造がサンプルコードとともに記載されています。

マッピング仕様の詳細については，『*OMG IDL/C++ Language Mapping Specification*』を参照してください。

第 3 章

VisiBroker を使ったサンプルアプリケーションの開発

この節では，サンプルアプリケーションを使用して，Java と C++ 両方のオブジェクトベースの分散アプリケーションを開発するプロセスについて説明します。

サンプルアプリケーションのコードは，bank_agent.html ファイルに用意されています。このファイルは次の場所にあります。

```
<install_dir>/examples/vbe/basic/bank_agent/
```

開発手順

VisiBroker を使って分散アプリケーションを開発する場合は，まずアプリケーションに必要なオブジェクトを識別します。下の図は，サンプルの bank アプリケーションを開発する手順を示しています。次に，各プロセスを簡単にまとめます。

1. Interface Definition Language（IDL）を使って各オブジェクトの仕様を記述します。

   IDL は，オブジェクトが提供するオペレーションとオブジェクトを呼び出す方法を指定する言語です。このサンプルでは，balance() メソッドを持つ Account インターフェースと，open() メソッドを持つ AccountManager インターフェースを IDL で定義します。

2. IDL コンパイラを使用して，クライアントスタブコードとサーバー POA サーバントコードを生成します。

   ステップ 1 で説明したインターフェース仕様と idl2java または idl2cpp コンパイラを使用して，リモートオブジェクトのインプリメンテーション用のクライアント側スタブとサーバー側クラスを生成します。

3. クライアントプログラムのコードを記述します。

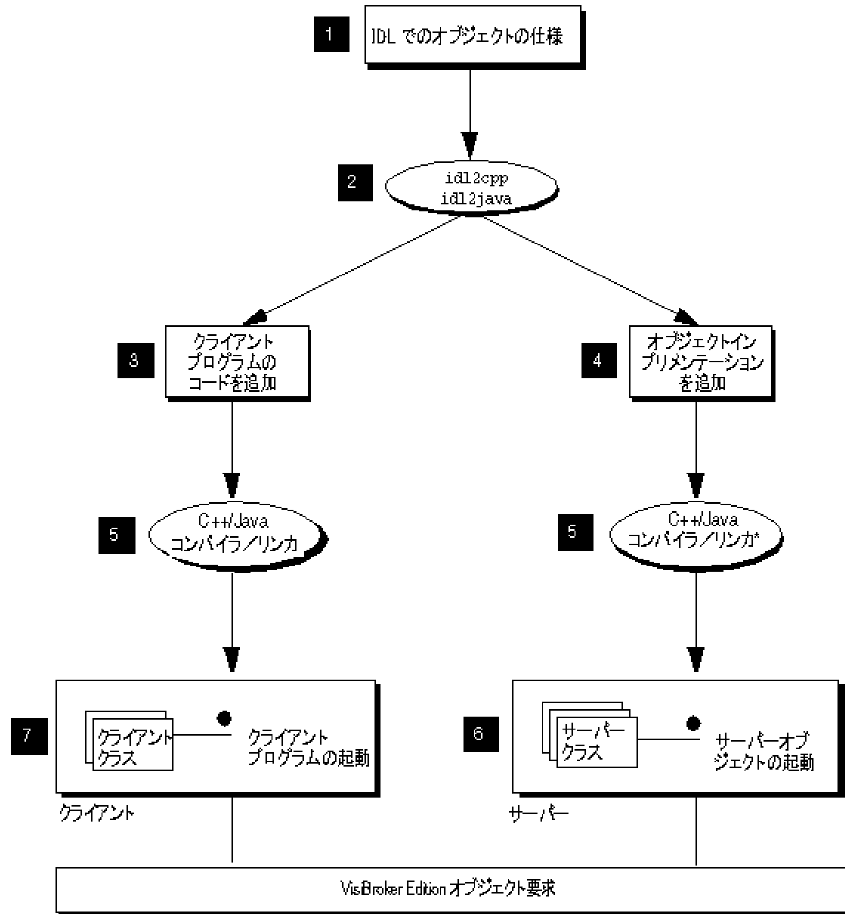
   クライアントプログラムのインプリメンテーションを完成するには VisiBroker ORB を初期化し，Account および AccountManager オブジェクトをバインドし，これらのオブジェクトのメソッドを呼び出して残高を出力するコードを記述します。

4. サーバーオブジェクトのコードを記述します。

サーバーオブジェクトのコードのインプリメンテーションを完成するには，AccountPOA クラスと AccountManagerPOA クラスからクラスを派生し，それらのインターフェースのメソッドにインプリメンテーションを提供し，サーバーの main ルーチンを実装します。

5 適切なスタブとスケルトンを使用して，クライアントコードとサーバーコードをコンパイルします。
6 サーバーを起動します。
7 クライアントプログラムを実行します。

図 3.1 サンプル bank アプリケーションの開発



ステップ 1 : オブジェクトインターフェースの定義

VisiBroker を使ってアプリケーションを作成するための最初の手順は，OMG の Interface Definition Language（IDL）を使用して，目的のオブジェクトおよびそれらのインターフェースをすべて指定することです。IDL は，さまざまなプログラミング言語にマッピングされます。

次に，idl2cpp コンパイラを使用して，IDL 仕様に準拠したスタブルーチンとサーバントコードを生成します。スタブルーチンは，クライアントプログラムがオブジェクトのオペレーションを呼び出すために使用します。記述したコードとともにサーバントコードを使用して，オブジェクトを実装するサーバーを作成します。

IDL を使った Account インターフェースの記述

IDL は C++ と似た構文を持っており，モジュール，インターフェース，データ構造などの定義に使用します。

次のサンプルは，bank_agent サンプルの Bank.idl ファイルの内容を示します。Account インターフェースは，現在の残高（balance）を取得するためのメンバー関数を 1 つだけ提供します。AccountManager インターフェースは，ユーザーの口座（Account）がまだ存在しない場合に，口座を作成します。

```
module Bank{
   interface Account {
      float balance();
   };
   interface AccountManager {
      Account open(in string name);
   };
};
```

ステップ 2 : クライアントスタブとサーバーサーバントの生成

IDL で作成したインターフェース仕様を使用して，VisiBroker の idl2cpp がクライアントプログラムの C++ スタブルーチンとオブジェクトインプリメンテーションのスケルトンコードを生成します。

クライアントプログラムは，すべてのメンバー関数の呼び出しでスタブルーチンを使用します。

記述したコードとともにスケルトンコードを使用して，オブジェクトを実装するサーバーを作成します。

クライアントプログラムとサーバーオブジェクトのコードが完成したら，それらのコードを C++ コンパイラとリンカへの入力として使用して，クライアントとサーバーを生成します。

Bank.idl ファイルには特別な処理が必要ないため，次のコマンドを使ってコンパイルできます。

```
prompt> idl2cpp Bank.idl
```

idl2cpp コンパイラのコマンドラインオプションの詳細については，第 13 章「IDL の使い方」を参照してください。

IDL コンパイラが生成するファイル

idl2cpp コンパイラは，Bank.idl ファイルから次の 4 つのファイルを生成します。

- Bank_c.hh : Account クラスと AccountManager クラスの定義を保持します。

- `Bank_c.cc`：クライアントが使用する内部スタブルーチンを保持します。
- `Bank_s.hh`：`AccountPOA` クラスと `AccountManagerPOA` servant クラスの定義を保持します。
- `Bank_s.cpp`：サーバーが使用する内部ルーチンを保持します。

クライアントアプリケーションを構築するには，`Bank_c.hh` ファイルと `Bank_c.cpp` ファイルを使用します。`Bank_s.hh` ファイルと `Bank_s.cpp` ファイルは，サーバーオブジェクトを構築するためのファイルです。生成されたすべてのファイルには `.cpp` か `.hh` のどちらかの拡張子が付き，ソースファイルと区別できます。

Windows： `idl2cpp` コンパイラが生成するファイルのデフォルトの拡張子は `.cpp` です。ただし，VisiBroker のサンプルアプリケーションに関連付けられているメイクファイルでは，`-src` サフィックスを使って出力先のファイルの拡張子を指定しています。

注意： `idl2cpp` コンパイラが生成したファイルの内容は変更しないでください。

ステップ 3：クライアントの実装

前のサンプルで示したように，Bank クライアントの実装で使用される多くのクラスは，`idl2cpp` コンパイラによって生成される `Bank` コード内に存在します。

`Client.C` ファイルは，このサンプルを説明するものです。このファイルは，`bank_agent` ディレクトリに置かれています。通常は，プログラマがこのファイルを作成します。

Client.C

`Client` プログラムは，現在の銀行口座の残高を取得するクライアントアプリケーションを実装します。Bank のクライアントプログラムは，次の手順を実行します。

1. VisiBroker ORB を初期化する。
2. `AccountManager` オブジェクトにバインドする。
3. `bind()` から返されるオブジェクトリファレンスを使用して，`Account` を取得する。
4. `Account` オブジェクトの `balance` を呼び出して，残高を取得する。

```
#include "Bank_c.hh"
int main(int argc, char* const* argv) {
   try {
      // ORB を初期化します。
      CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
      // マネージャの ID を取得します。
      PortableServer::ObjectId_var managerId =
            PortableServer::string_to_ObjectId("BankManager");
      // AccountManager を検索します。完全な POA 名とサーバント ID を指定します。
      Bank::AccountManager_ptr manager =
      Bank::AccountManager::_bind("/bank_agent_poa", managerId);
      // 口座名として argv[1]，またはデフォルトを使用します。
      const char* name = argc > 1 ? argv[1] : "Jack B. Quick";
      // アカウントマネージャに指定した口座を開くように要求します。
      Bank::Account_ptr account = manager->open(name);
      // 口座の残高を取得します。
      float balance = account->balance();
      // 残高を印刷します。
      cout << "The balance in " << name << "'s account is $" << balance << endl;
   } catch(const CORBA::Exception& e) {
      cerr << e << endl;
   }
}
```

AccountManager オブジェクトのへのバインド

クライアントプログラムは，open(String name) メンバー関数を呼び出す前に，bind() メンバー関数を使用して，AccountManager オブジェクトを実装するサーバーへの接続を確立する必要があります。

bind() メンバー関数のインプリメンテーションは，idl2cpp によって自動的に実装されます。bind() メンバー関数は，サーバーを検索して接続を確立するように VisiBroker ORB に要求します。

正しくサーバーが見つかり，接続が確立されると，サーバーの AccountManagerPOA オブジェクトを表すプロキシオブジェクトが作成されます。クライアントプログラムには，ポインタが返されます。

Account オブジェクトの取得

C++：次に，クライアントプログラムは AccountManager オブジェクトの open() メンバー関数を呼び出して，指定された顧客名に対応する Account オブジェクトへのポインタを取得する必要があります。

残高の取得

クライアントプログラムが Account オブジェクトとの接続を確立すると，balance() メンバー関数を使って残高を取得できます。クライアント側の balance() メンバー関数は，実際には idl2cpp コンパイラによって生成されたスタブです。このスタブが，要求に必要なすべてのデータを集め，その要求をサーバーオブジェクトに送信します。

このほかにも，クライアントプログラムが AccountManager のオブジェクトリファレンスを操作するためのメンバー関数がいくつかあります。

ステップ 4：サーバーの実装

クライアントの場合と同様に，Bank サーバーの実装で使用される多くのクラスは，idl2cpp コンパイラによって生成される Bank のヘッダーファイルに格納されています。Server.C ファイルは，このサンプルのために用意されているサーバーインプリメンテーションです。通常は，プログラマがこのファイルを作成します。

サーバープログラム

このファイルは，Bank サンプルのサーバー側の Server クラスを実装します。次のサンプルコードは，サーバー側プログラムの例です。サーバープログラムは次の処理を実行します。

- オブジェクトリクエストブローカー（ORB）を初期化する。
- 必要なポリシーを使ってポータブルオブジェクトアダプタ（POA）を作成する。
- AccountManager サーバントオブジェクトを作成する。
- サーバントオブジェクトをアクティブ化する。
- POA マネージャ（および POA）をアクティブ化する。
- 要求の受信を待機する。

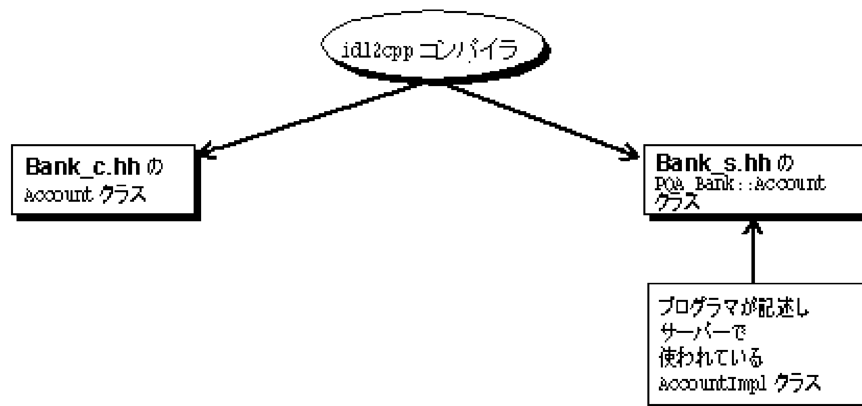
```
#include "BankImpl.h"
int main(int argc, char* const* argv) {
   try {
      // ORB を初期化します。
      CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
      // ルート POA へのリファレンスを取得します。
      CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
      PortableServer::POA_var rootPOA = PortableServer::POA::_narrow(obj);
```

```
            CORBA::PolicyList policies;
            policies.length(1);
            policies[(CORBA::ULong)0] =
                  rootPOA->create_lifespan_policy(PortableServer::PERSISTENT);
            // POA マネージャを取得します。
            PortableServer::POAManager_var poa_manager = rootPOA->the_POAManager();
            // 適切なポリシーで myPOA を作成します。
            PortableServer::POA_var myPOA = rootPOA->create_POA("bank_agent_poa",
                  poa_manager, policies);
            // サーバントを作成します。
            AccountManagerImpl managerServant;
            // サーバントの ID を決定します。
            PortableServer::ObjectId_var managerId =
                  PortableServer::string_to_ObjectId("BankManager");
            // その ID を使って myPOA でサーバントをアクティブ化します。
            myPOA->activate_object_with_id(managerId, &managerServant);
            // POA マネージャをアクティブ化します。
            poa_manager->activate();
            cout << myPOA->servant_to_reference(&managerServant) << " is ready" << endl;
            // 着信要求を待機します。
            orb->run();
         } catch(const CORBA::Exception& e) {
            cerr << e << endl;
            return 1;
         }
         return 0;
      }
```

Account クラスの階層

実装する Account クラスは，idl2cpp コンパイラによって生成された POA_Bank::Account クラスから派生されます。Bank_c.hh ファイルにある POA_Bank::Account クラスの定義をよく見ると，それが Account クラスから派生していることがわかります。次の図に，このクラス階層を示します。

図 3.2　AccountImpl インターフェースのクラス階層



ステップ 5：サンプルのビルド

VisiBroker リリースの examples ディレクトリには，このサンプルとほかの VisiBroker サンプル用の Makefile.cpp があります。

作成した Client.C ファイルと生成された Bank_c.cc ファイルを一緒にコンパイルおよびリンクして，クライアントプログラムを作成します。作成した Server.C ファイルと，生成された Bank_s.cpp ファイルおよび Bank_c.cpp ファイルを一緒にコンパイルおよびリンクし

て，Bank Account サーバーを作成します。クライアントプログラムとサーバーは，どちらも VisiBroker ORB ライブラリとリンクする必要があります。

また，examples ディレクトリには stdmk（UNIX）や stdmk_nt（Windows NT）という名前のファイルもあります。使用するファイルの場所と変数の設定は，Makefile で定義します。

使用しているコンパイラが特定のフラグをサポートしていない場合は，stdmk ファイルまたは stdmk_nt ファイルのカスタマイズが必要な場合があります。

サンプルのコンパイル

Windows： VisiBroker が C:¥vbroker にインストールされているとします。サンプルをコンパイルするには，次のように入力します。

```
prompt> C:
prompt> cd vbroker¥examples¥basic¥bank_agent
prompt> nmake -f Makefile.cpp
```

Visual C++ の nmake コマンドは，idl2cpp コンパイラを実行してから，各ファイルをコンパイルします。

make の実行中に問題が発生した場合は，path 環境変数に VisiBroker ソフトウェアがインストールされている bin ディレクトリが含まれているかどうかを確認してください。

また，VisiBroker ソフトウェアがインストールされているディレクトリを VBROKERDIR 環境変数に設定してみてください。

UNIX： VisiBroker が /usr/local にインストールされているとします。サンプルをコンパイルするには，次のように入力します。

```
prompt> cd /usr/local/vbroker/examples/basic/bank_agent
prompt> make cpp
```

このサンプルの make は，標準の UNIX 機能です。PATH に make がない場合は，システム管理者に問い合わせてください。

ステップ 6：サーバーの起動とサンプルの実行

クライアントプログラムとサーバーインプリメンテーションのコンパイルが完了しました。これで最初の VisiBroker アプリケーションを実行できます。

スマートエージェントの起動

VisiBroker のクライアントプログラムまたはサーバーインプリメンテーションを実行するには，まず，ローカルネットワークの少なくとも 1 つのホストでスマートエージェントを起動する必要があります。

次は，スマートエージェントを起動するための基本のコマンドです。

```
prompt> osagent
```

スマートエージェントについては，第 14 章「スマートエージェントの使い方」を参照してください。

サーバーの起動

Windows： DOS プロンプトウィンドウを開き，次の DOS コマンドを使ってサーバーを起動します。

```
prompt> start Server
```

UNIX： Account サーバーを起動するには，次のコマンドを入力します。

```
prompt> Server&
```

クライアントの実行

Windows： 別の DOS プロンプトウィンドウを開き，次の DOS コマンドを使ってクライアントを起動します。

```
prompt> Client
```

UNIX： クライアントプログラムを開始するには，次のコマンドを入力します。

```
prompt> Client
```

次のような出力が表示されます。口座残高はランダムに計算されます。

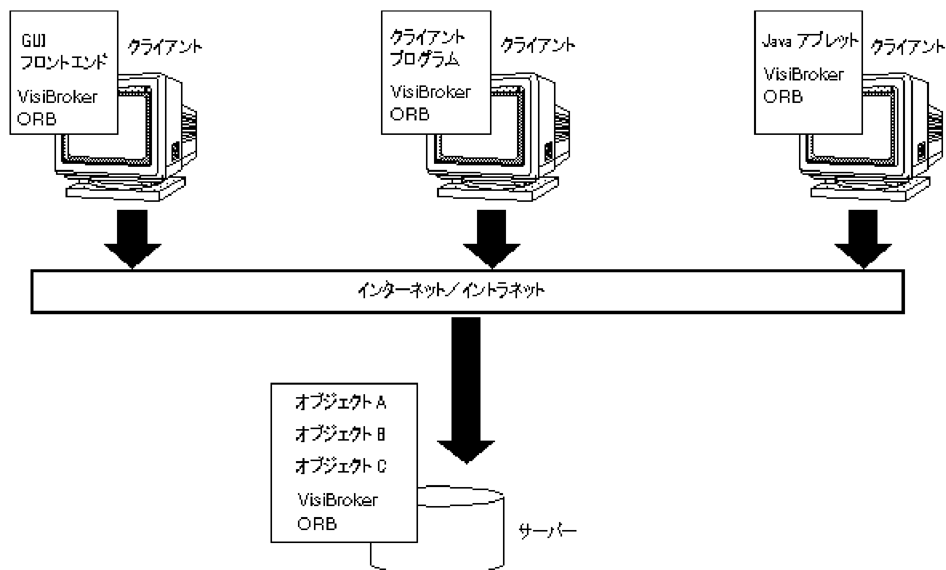
```
The balance in the account in $168.38.
```

VisiBroker を使ったアプリケーションの配布

VisiBroker は配布段階でも使用します。これは，開発者がクライアントプログラムやサーバーアプリケーションを開発し，テストを行って本稼動の準備が整った段階です。この段階で，システム管理者は，エンドユーザーのデスクトップにクライアントプログラムを配布したり，サーバークラスのマシンにサーバーアプリケーションを配布するための準備を行います。

VisiBroker ORB は，配布のためにフロントエンドでクライアントプログラムをサポートします。クライアントプログラムを実行する各マシンに VisiBroker ORB をインストールする必要があります。VisiBroker ORB を利用する同じホスト上のクライアントは，VisiBroker ORB を共有します。VisiBroker ORB は，中間層でサーバーアプリケーションもサポートします。サーバーアプリケーションを実行する各マシンで VisiBroker ORB をインストールする必要があります。VisiBroker ORB を利用する同じサーバーマシン上のサーバーアプリケーションやオブジェクトは，VisiBroker ORB を共有します。クライアントは，GUI フロントエンド，アプレット，またはクライアントプログラムになります。サーバーインプリメンテーションは，中間層でビジネスロジックを保持します。

図 3.3 VisiBroker ORB を使って配布されるクライアントプログラムとサーバープログラム



VisiBroker アプリケーション

アプリケーションの配布

VisiBroker を使って開発されたアプリケーションを配布するには，最初に，そのアプリケーションが実行されるホストで実行時環境を設定し，必要なサポートサービスがローカルネットワークで利用できるかどうかを確認する必要があります。

VisiBroker for C++ を使って開発されたアプリケーションには，次の実行時環境が必要です。

- VisiBroker ライブラリ。これは，製品をインストールしたディレクトリの bin サブディレクトリにあります。
- アプリケーションから要求されるサポートサービスの有効性。

配布するアプリケーションが実行されるホストに，VisiBroker ORB ライブラリをインストールする必要があります。また，これらのライブラリの場所をアプリケーションの環境の PATH に入れる必要があります。

環境変数

配布したアプリケーションが特定のホストのスマートエージェント（osagent）を使用するように指定するには，そのアプリケーションを実行する前に，OSAGENT_ADDR 環境変数を設定する必要があります。ORBagentAddr プロパティをコマンドライン引数として使用して，ホスト名または IP アドレスを指定できます。23 ページの「サポートサービスの有効性」に，必要なコマンドライン引数をリストします。

配布したアプリケーションがスマートエージェントと通信するときに，特定の UDP ポートを使用するように指定するには，そのアプリケーションを実行する前に，OSAGENT_PORT 環境変数を設定する必要があります。

IP ポート番号は，ORBagentPort（C++）コマンドライン引数を使って指定できます。

環境変数の詳細については，『Borland VisiBroker インストールガイド』を参照してください。

サポートサービスの有効性

配布したアプリケーションが実行されるネットワーク上のどこかで，スマートエージェントが実行されている必要があります。配布するアプリケーションに必要な条件に基づいて，ほかの VisiBroker 実行時サポートサービスが利用できるかどうかも確認してください。次のようなサービスがあります。

サポートサービス	必要な場合：
オブジェクトアクティベーションデーモン（oad）	オンデマンドで起動する必要があるオブジェクトを実装するサーバーアプリケーションを配布する場合。
インターフェースリポジトリ（irep）	動的スケルトンインターフェースまたは動的インプリメンテーションインターフェースのどちらかを使用するアプリケーションを配布する場合。これらのインターフェースの詳細については，第 21 章「インターフェースリポジトリの使い方」を参照してください。
GateKeeper	ネットワークセキュリティとしてファイアウォールを使用する環境で実行するアプリケーションを配布する場合。

アプリケーションの実行

VisiBroker のクライアントプログラムまたはサーバーインプリメンテーションを実行するには，まず，ローカルネットワークの少なくとも 1 つのホストでスマートエージェントを起動する必要があります。スマートエージェントについては，「スマートエージェントの起動」で詳しく説明しています。

クライアントアプリケーションの実行

クライアントアプリケーションは，VisiBroker ORB オブジェクトを使用しますが，自分からほかのクライアントアプリケーションには VisiBroker ORB オブジェクトを提供しないアプリケーションです。

次の表は，クライアントアプリケーションに指定できるコマンドライン引数をまとめたものです。これらの引数はサーバーに対しても適用されます。

表 3.1　C++ クライアントアプリケーションのコマンドライン引数

オプション	説明
-ORBagentAddr <hostname|ip_address>	このクライアントが使用するスマートエージェントが実行されているホストのホスト名，または IP アドレスを指定します。指定したアドレスにスマートエージェントが見つからない場合，またはこのオプションが指定されていない場合は，ブロードキャストメッセージを使ってスマートエージェントを検索します。
-ORBagentPort <port_number>	スマートエージェントのポート番号を指定します。このオプションは，複数の VisiBroker ORB ドメインが必要な場合に便利です。指定しない場合は，デフォルトのポート番号 14000 が使用されます。
-ORBbackcompat <0|1>	1 に設定された場合は，VisiBroker for C++ バージョン 2.0 との下位互換性の提供を指定します。デフォルトは 0 です。
-ORBbackdii <0|1>	1 に設定された場合は，1.0 IDL-to-C++ マッピングの提供を指定します。0 に設定されるか値が設定されなかった場合は，新しい 1.1 マッピングが使用されます。デフォルトの設定値は 0 です。-ORBbackcompat を 1 に設定した場合，このオプションは自動的に 1 に設定されます。
-ORBir_name <ir_name>	オブジェクトインプリメンテーションで Object::get_interface() メソッドが呼び出される場合にアクセスするインターフェースリポジトリの名前を指定します。
-ORBir_ior <ior_string>	オブジェクトインプリメンテーションで Object::get_interface() メソッドが呼び出される場合にアクセスするインターフェースリポジトリの IOR を指定します。
-ORBnullstring <0|1>	1 に設定された場合，ORB は，C++ の NULL 文字列をストリーム化できます。NULL 文字列は，長さ 0 の文字列としてマーシャリングされます。一方，空文字列（""）は，長さ 1 の文字列（1 文字 ¥0）としてマーシャリングされます。0 に設定された場合，NULL 文字列をアンマーシャリングしようとすると，CORBA::BAD_PARAM が生成されます。NULL をアンマーシャリングしようとすると，CORBA::MARSHAL が生成されます。デフォルトの設定値は 0 です。-ORBbackcompat を 1 に設定した場合，このオプションは自動的に 1 に設定されます。
-ORBrcvbufsize <buffer_size>	応答の受信に使用する TCP バッファのサイズ（バイト単位）を指定します。指定しなかった場合は，デフォルトのバッファサイズが使用されます。この引数を使用して，パフォーマンスまたはベンチマークの結果に著しい影響を与えることができます。
-ORBsendbufsize <buffer_size>	クライアント要求の送信に使用する TCP バッファのサイズ（バイト単位）を指定します。指定しなかった場合は，デフォルトのバッファサイズが使用されます。この引数を使用して，パフォーマンスまたはベンチマークの結果に著しい影響を与えることができます。
-ORBshmsize <size>	共有メモリ内の送信セグメントと受信セグメントのサイズ（バイト単位）を指定します。クライアントプログラムとオブジェクトインプリメンテーションが共有メモリを介して通信する場合は，このオプションを調整するとパフォーマンスが向上することがあります。このオプションは Windows プラットフォームでしかサポートされていません。
-ORBtcpnodelay <0|1>	1 に設定された場合は，すべてのソケットが要求をただちに送信するように指定します。デフォルト値の 0 の場合，ソケットはバッファがいっぱいになったときに要求を一括して送信します。この引数を使用して，パフォーマンスまたはベンチマークの結果に著しい影響を与えることができます。

第 4 章

C++ 対応プログラマツール

この章では，VisiBroker for C++ が提供するプログラマツールについて説明します。

VisiBroker for C++ のヘッダーファイル用スイッチ

次のスイッチは，ヘッダーファイルのコンシューマを適切なコードライブラリにポイントするために使用します。

_VIS_STD

VisiBroker for C++ が従来の C++ アプリケーションと標準の C++ アプリケーションの両方の開発をサポートしているプラットフォームでは，`_VIS_STD` を定義すると，VisiBroker for C++ ヘッダーファイルに C++ ライブラリの正しい C++ ヘッダーファイルをインクルードできます。標準の C++ アプリケーションを開発している場合は，コンパイル時に `_VIS_STD` フラグを使用します。従来の C++ アプリケーションを開発している場合は，このフラグを使用しないでください。

_VIS_NOLIB

Windows では，VisiBroker for C++ のヘッダーファイル（`vdef.h`）は，VisiBroker for C++ ライブラリの検索レコードを自動的にオブジェクトファイルに置きます。これは，MSVC コンパイラと BCB コンパイラの両方に `#pragma` コメントを使って行います。`_DEBUG`，`VISDEBUG`，`_VIS_STD` などのその他の定義に応じて，適切なライブラリ検索レコードが選択されます。この動作が不要で，VisiBroker for C++ ライブラリ名をアプリケーションのリンクコマンドで明示的に指定する場合は，`_VIS_NOLIB` を定義する必要があります。デフォルトでは，定義されません。

引数／オプション

引数には，すべての VisiBroker プログラマツールに共通の引数と，各ツールに固有の引数があります。各ツールに固有の引数とオプションは，そのツールの節で説明します。次に，汎用のオプションをリストします。

共通オプション

次のオプションは，すべてのプログラマツールに共通です。

オプション	説明
-J<java option>	java_option ディレクトリを Java 仮想マシンに渡します。
-VBJversion	VisiBroker のバージョンを出力します。
-VBJdebug	VisiBroker のデバッグ情報を出力します。
-VBJclasspath	クラスパスを指定します。クラスパスは CLASSPATH 環境変数より優先します。
-VBJprop <name> [=<value>]	名前／値の組を Java 仮想マシンに渡します。
-VBJjavavm <jvmpath>	Java 仮想マシンのパスを指定します。
-VBJaddJar <jarfile>	Java 仮想マシンを実行する前に，CLASSPATH の末尾に jarfile を追加します。

メモ UNIX プラットフォームの場合，-J オプションは，Solaris 上の VisiBroker Edition for Java でのみ使用可能です。

一般情報

この章で説明している VisiBroker プログラミングツールの構文は，使用している環境が UNIX と Windows のどちらであるかによって異なります。最初に UNIX バージョンの構文を示し，その後で Windows バージョンの構文を示します。

UNIX： UNIX 環境で，コマンドのオプションを表示するには，次のように入力します。

構文	サンプル
command name -¥?	idl2cpp -¥?

Windows： Windows 環境で，コマンドのオプションを表示するには，次のように入力します。

構文	サンプル
command name -?	idl2cpp -?

idl2cpp

このコマンドは，VisiBroker の IDL から C++ へのコンパイラを実装し，IDL ファイルからクライアントスタブとサーバースケルトンコードを生成します。

構文

```
idl2cpp [arguments] infile(s)
```

idl2cpp は IDL ファイルを入力とし，クライアント側とサーバー側のそれぞれの C++ クラス，クライアントスタブ，およびサーバースケルトンコードを生成します。

infile パラメータは，C++ コードの生成元になる IDL ファイルを指定します。arguments は，生成されるコードをさまざまに制御します。

サンプル

```
idl2cpp -hdr_suffix hx -server_ext _serv -no_tie -no_excep-spec bank.idl
```

Windows: idl2cpp が生成したスタブとスケルトンを基にしてインプリメンテーションをリンクする場合は，-DSTRICT プリプロセッサオプションを使用してください。このオプションを使用しないと，リンカが orb.lib にコンストラクタがないというエラーメッセージを表示する可能性があります。

引数	説明
-D, -defined foo[=bar]	プリプロセッサマクロ foo を定義します。値 bar はオプションです。複数のプリプロセッサマクロを指定するには，その数だけ -D オプションを使用します。たとえば，-Dfoo=bar -Dhello=world です。
-H, -list_includes	標準エラー出力にインクルードファイルのフルパスを出力します。デフォルトはオフです。
-I, -include <dir>	#include の検索対象のディレクトリを追加します。複数の #include ディレクトリを指定するには，その数だけ -I オプションを使用します。たとえば，-I/home/include -I /app/include です。
-P, no_line_directives	行番号情報の生成を抑止します。デフォルトはオフです。
-U, -undefine foo	プリプロセッサマクロ foo の定義を解除します。
-client_ext <file_extension>	生成されるクライアントファイルに使用するファイル拡張子を指定します。デフォルトの拡張子は (_c) です。拡張子のないクライアントファイルを生成するには，<file_extension> の値として none を指定します。
-[no_]back_compat_mapping	現在のリリースでは，このオプションは何も行いません。次のリリースで変更される予定です。
_[no_]boa	BOA 互換のコードを生成します。デフォルトでは，このコードは生成されません。
-[no_]comments	コード中にコメントを保持します。デフォルトで，このコメントは生成されたコードに表示されます。
-[no_]idl_strict	IDL ソースの解釈を厳密な OMG 標準に制限します。デフォルトでは，OMG 標準による解釈は使用されません。
-[no_]obj_wrapper	オブジェクトラッパーサポートを使ってスタブ，およびスケルトンを生成します。このオプションはまた，ほかのすべてのオブジェクトラッパーが継承するベースの型付きオブジェクトラッパーと，型なしオブジェクト呼び出しを実行するデフォルトオブジェクトとを生成します。このオプションを設定しなかった場合，idl2cpp はオブジェクトラッパーのためのコードを生成しません。
-[no_]preprocess	解析の前に IDL ファイルをプリプロセスします。デフォルトではオンになっています。
-[no_]preprocess_only	プリプロセス後に IDL ファイルの解析を停止します。このオプションは，コンパイラで stdout のプリプロセスの結果を生成します。デフォルトはオンです。
-[no_]pretty_print	_pretty_print メソッドを生成します。デフォルトではオンになっています。
-[no_]servant	サーバー側コードを生成します。デフォルトでは，サーバントが生成されます。
-[no_]stdstream	クラスのシグニチャ内に，標準の iostream クラスを使ったストリーム演算子を生成します。デフォルトはオンです。
-[no_]tie	_tie テンプレートクラスを生成します。デフォルトで，_tie クラスが生成されます。
-[no_]warn_all	すべての警告をオフにします。デフォルトはオフです。
-[no_]warn_missing_define	前方参照宣言された名前が定義されていない場合に警告を出します。デフォルトはオンです。
-[no_]warn_unrecognized_pragmas	#pragma を認識できない場合に警告を表示します。
-corba_inc <filename>	生成されるコードに，通常の #include <filename> 指示文ではなく，#include <corba.h> 指示文を挿入します。デフォルトでは，#include <corba.h> が生成されたコードに挿入されます。
-[no_]examples	サンプルインプリメンテーションの生成を指定します。デフォルトでは，サンプルインプリメンテーションは生成されません。

引数	説明
-excep_spec	メソッドの例外仕様を生成します。デフォルトでは，例外仕様は生成されません。
Windows: -export <tag>	生成されるすべてのクライアント側宣言（クラス，関数など）に挿入するタグの名前を定義します。idl2cpp の呼び出しで -export _MY_TAG を指定すると， class Bank {...} ではなく， class _MY_TAG Bank{...} のようなクラス定義が生成されます。デフォルトでは，クライアント側の宣言にタグは生成されません。
Windows: -export_skel <tag>	生成されるサーバー側宣言だけに挿入するタグの名前を定義します。idl2cpp の呼び出しで -export _MY_TAG を指定すると， class POA_Bank {...} ではなく， class _MY_TAG POA_Bank{...} のようなクラス定義が生成されます。デフォルトでは，サーバー側の宣言にタグは生成されません。
-gen_include_files	#include ファイルのコードを生成します。デフォルトでは，このコードは生成されません。
-h, -help, -usage, -?	ヘルプ情報を表示します。
-hdr_suffix <string>	ヘッダーファイル名の拡張子を指定します。デフォルトは .hh です。
-impl_inherit	インプリメンテーションの継承を生成します。デフォルトはオフです。
-list_files	コードの生成中，書き込まれるファイルをリストします。デフォルトでは，このリストは作成されません。
-map_keyword <keywrd> <map>	<keywrd> をキーワードとして追加し，指定したマッピングを関連付けます。<keywrd> と競合する IDL 識別子は，C++ の <map> にマッピングされます。これにより，C++ コード内で使用されている名前とキーワードが競合しなくなります。C++ のキーワードはすべてデフォルトのマッピングを持っています。このオプションを使用して，それらのデフォルトのマッピングを指定する必要はありません。
-namespace	モジュールを名前空間として実装します。デフォルトはオフです。
-root_dir <path>	生成されたコードを書き込むディレクトリを指定します。デフォルトでは，コードは現在のディレクトリに書き込まれます。
-server_ext <file_extension>	生成されるサーバーファイルに使用するファイル拡張子を指定します。デフォルトの拡張子は (_s) です。拡張子のないサーバーファイルを生成するには，<file_extension> の値として **none** を指定します。
-src_suffix <string>	ソースファイル名の拡張子を指定します。デフォルトは .cc です。
-target <compiler>	C++ コードの生成に使用するコンパイラを指定します。デフォルトのコンパイラは Solaris です。
-type_code_info	動的起動インターフェースを使用するクライアントプログラムに必要なタイプコード情報を生成します。デフォルトでは，タイプコード情報は生成されません。
-version	VisiBroker のソフトウェアバージョン番号を表示します。
-corba_style	-type_code_info フラグを必要とします。CORBA::Any へのポインタの挿入または抽出を生成します。デフォルトでは，オフです。
-corba_style_tie	-tie フラグを必要とします。同じスコープ内にスケルトンクラスとして tie クラスを生成します。デフォルトでは，オフです。

引数	説明
file1 [file2] ...	処理するファイルを 1 つ以上指定します。標準入力の場合は "-" を指定します。
CPP	orb.idl には，VisiBroker for C++ または VisiBroker for Java に固有の条件付き定義が含まれます。したがって，IDL に orb.idl を入れる場合は，CPP マクロを使って VisiBroker for C++ 固有の定義を有効にする必要があります。たとえば，idl2cpp -D **CPP** test.idl のコードを使用します。または，IDL ファイルの先頭に次の行を記述します。 #define CPP

idl2ir

このコマンドを使用すると，インターフェース定義言語（Interface Definition Language, IDL）ソースファイルの中で定義されているオブジェクトをインターフェースリポジトリ（IR）に記入できます。

構文

```
idl2ir [-ir <IR_name>] [-replace] <filename>.idl [<filename2>.idl ...]
```

サンプル

```
idl2ir -ir my_repository -replace bank/Bank.idl
```

説明 idl2ir コマンドは <filename>.idl を入力とし，インターフェースリポジトリサーバーにバインドして，idl filename 内の IDL 構造をリポジトリに記入します。-replace オプションが指定されている場合や，IDL ファイル内の項目と同じ名前の項目がすでにこのリポジトリにある場合は，古い項目が置き換えられます。

メモ idl2ir コマンドは，匿名の配列やシーケンスを正しく処理しません。この問題を回避するには，すべてのシーケンスと配列に対して typedefs を使用する必要があります。

オプション	説明
-D, -define foo[=bar]	プリプロセッサマクロ foo を定義します。値 bar はオプションです。
-I, -include <dir>	#include の検索対象のディレクトリを追加します。
-P, no_line_directives	行番号情報の生成を抑止します。デフォルトはオフで，行番号の生成は抑止されません。
-H, _list_includes	標準エラー出力にインクルードファイルの名前を出力します。デフォルトはオフです。
-U, -undefine foo	プリプロセッサマクロ foo の定義を解除します。
-[no_]back_compat_mapping	VisiBroker 3.x と下位互換性のあるマッピングを使用します。
-{no_]idl_strict	IDL ソースの解釈を厳密な OMG 標準に制限します。デフォルトでは，OMG 標準による解釈は使用されません。
-[no_]preprocess	解析の前に IDL ファイルをプリプロセスします。デフォルトではオンになっています。
-[no_]preprocess_only	プリプロセス後に IDL ファイルの解析を停止します。このオプションは，コンパイラで stdout のプリプロセスの結果を生成します。デフォルトはオンです。
-[no_]warn_all	すべての警告をオフにします。デフォルトはオフです。
-[no_]warn_unrecognized_pragmas	#pragma を認識できない場合に警告を表示します。
-deep	（シャローマージではなく）ディープマージを指定します。-deep を指定すると，新しい内容と既存の内容の差分だけがマージされます。-shallow マージでは，新しい内容で同じ名前を定義した場合は，すべての既存の内容が新しい内容で置き換えられます。デフォルトは off です。
-h, -help, -usage, -?	使用方法を表示します。
-irep <name>	idl2ir がバインドしようとするインターフェースリポジトリのインスタンス名を指定します。名前を指定しなかった場合，idl2ir は現在のドメイン内で見つかるインターフェースリポジトリサーバーに自分自身をバインドします。現在のドメインは，OSAGENT_PORT 環境変数で定義されます。

オプション	説明
-replace	定義を更新するのではなく，置き換えます。
-version	VisiBroker のソフトウェアバージョン番号を表示します。
file1 [file2] ...	処理するファイルを 1 つ以上指定します。標準入力の場合は "-" を指定します。

ir2idl

このコマンドを使用すると，インターフェースリポジトリ内のオブジェクトからインターフェース定義言語（IDL）ソースファイルを作成できます。

構文

```
ir2idl [options]
```

サンプル

foo という名前の IR の内容を foo.idl という名前のファイルにダンプします。

```
ir2idl -irep foo -o foo.idl
```

説明

ir2idl コマンドは，IR の内容を抽出し，IDL として出力します。

オプション

ir2idl では，次のオプションを使用できます。

オプション	説明
-irep <irep name>	インターフェースリポジトリの名前を指定します。
-o, <file>	出力ファイルの名前を指定します。stdout の場合は，「-」を指定します。
-strict	コードを生成方法を OMG 標準に制限します。デフォルトはオンです。入力 IDL で Borland 固有の構文拡張が検出されると，コンパイラはエラーを表示します。
-version	現在実行している VisiBroker のバージョンを表示または出力します。
-h, -help, -usage, -?	使用方法を表示します。

idl2wsc

idl2wsc は，Axis C++ v1.5 WSDL2Ws サーバー側生成コードに似た C++ コードを生成するとともに，必要な CORBA サーバーへの CORBA 呼び出しも生成します。 これは，C++ Web サービス CORBA ブリッジコードを構成します。

IDL 名が Foo.idl だとすると，idl2wsc ツールは，デフォルトでファイル Foo_ws_s.cc, Foo_ws_c.hh, Foo.wsdl, corba.wsdl, および Foo.wsdd を生成します。「*.cc」,「*.hh」, および「*.wsdl」ファイルは変更しないでください。生成された WSDD ファイルは，C++ Web サービスランタイムライブラリによってロードされるコンパイル済み共有ライブラリを指すように変更できます。

idl2cpp のオプションは，idl2wsc でも使用できます。idl2cpp のオプションに加えて，idl2wsc 独自の次のオプションがあります。

オプション	説明
-encoding_wsi_only	特定の WS-I エンコーディングだけを生成します。デフォルトは OFF です。
-encoding_soap_only	特定の SOAP エンコーディングだけを生成します。デフォルトは OFF です。
-gen_cpp_bridge	VisiBroker for C++ ブリッジコードを生成します。デフォルトは OFF です。

idl2wsc の使用

IDL ファイルを idl2wsc に渡して C++ ブリッジコードを生成する前に，その IDL ファイルを idl2cpp に渡して CORBA スタブコードを生成する必要があります。idl2wsc ツールは，idl2cpp によって生成されたファイルの名前やシグニチャを参照するため，CORBA

スタブコードの生成に使用する idl2cpp のオプションと同じオプションを idl2wsc ツールにも適用する必要があることに注意してください。

idl2cpp で生成されたコードや新しいバージョンの「¥include¥vbws.h」に変更を加えた場合は，idl2wsc で生成されたコードを再コンパイルする必要があります。

idl2wsc の制限

Axis C++ v1.5 WSDL2WS ツールは，複数の portType が定義された WSDL ファイルをサポートしません。定義されている portTypes を 1 つだけを生成します。これは Axis C++ v1.5 の制限であり, そのため, 複数のインターフェースを含む IDL ファイルはサポートされません。

第 5 章

IDL から C++ へのマッピング

ここでは，VisiBroker for C++ の idl2cpp コンパイラが提供する IDL から C++ 言語へのマッピングについて説明します。このコンパイラは，CORBA C++ 言語マッピング仕様に厳密に準拠しています。

プリミティブデータ型

次の表は，インターフェース定義言語（Interface Definition Language，IDL）が提供するプリミティブデータ型をまとめたものです。プラットフォーム間のハードウェアの違いにより，一部の IDL プリミティブデータ型は，「platform dependent（プラットフォームに依存）」と定義されています。64 ビットの整数表現を持つプラットフォームでも，たとえば，g 型は依然として 32 ビットです。特定のプラットフォームにおけるプリミティブデータ型の正確なマッピングについては，インクルードファイル **orbtypes.h** を参照してください。

表 5.1　IDL プリミティブ型のマッピング

IDL 型	VisiBroker の型	C++ 定義
short	CORBA::Short	short
long	CORBA::Long	platform dependent
unsigned short	CORBA::UShort	unsigned short
unsigned long	CORBA::ULong	unsigned long
float	CORBA::Float	float
double	CORBA::Double	double
char	CORBA::Char	char
wchar	CORBA::WChar	wchar_t
boolean	CORBA::Boolean	unsigned char
octet	CORBA::Octet	unsigned char
long long	CORBA::LongLong	platform dependent
ulong long	CORBA::ULongLong	platform dependent

注意：　CORBA 仕様では，IDL の boolean 型は，1 または 0 のいずれかの値だけを持つように定義されます。boolean 型でこれ以外の値を使用すると，未定義の動作が発生します。

文字列

IDL の固定長および可変長の String 型は，どちらも C++ の char * 型にマッピングされます。

メモ　すべての CORBA String 型は NULL で終わります。

アプリケーションで VisiBroker と同じ管理機能を使用するには, 次の関数で文字列を動的に割り当て，解放します。

```
class CORBA
{
   ...
   static char *string_alloc(CORBA::ULong len);
   static void string_free(char *data);
   ...
};
```

CORBA::char *string_alloc(CORBA::ULong len);

文字列を動的に割り当て，文字列へのポインタを返します。割り当てに失敗した場合は，NULL ポインタが返されます。

パラメータ	説明
len	len パラメータで指定する長さに，NULL 終端子を入れる必要はありません。

CORBA::void *string_free(char *data);

CORBA::string_alloc を使って割り当てられた文字列に関連付けられているメモリを解放します。

パラメータ	説明
data	CORBA::string_alloc を使って割り当てられた文字列へのポインタです。

String_var クラス

IDL の string を char * にマッピングするたびに, IDL コンパイラは String_var クラスも生成します。このクラスは，文字列を保持するために割り当てられたメモリへのポインタを格納します。String_var オブジェクトを破棄するか，スコープ外に出ると，その文字列に割り当てられたメモリが自動的に解放されます。

次は，String_var クラスのメンバーとメソッドです。

```
class CORBA      {
   class String_var {
         protected:
               char   *_p;
               ...
         public:
               String_var();
               String_var(char *p);
               ~String_var();
               String_var&      operator=(const char *p);
               String_var&      operator=(char *p);
               String_var&      operator=(const String_var& s);
               operator const char *() const;
               operator char *();
               char &operator[](CORBA::ULong index);
               char operator[](CORBA::ULong index) const;
               friend ostream&         operator<<(ostream&, const String_var&);
               inline friend Boolean         operator==(const String_var& s1,
                           const String_var& s2);
```

```
                ...
        };
        ...
    };
```

_var クラスの詳細については，「<class_name>_var」を参照してください。

定数

インターフェース仕様の外部で定義された IDL の定数は，C++ の定数宣言に直接マッピングされます。次に例を示します。

このサンプルコードは，IDL の最上位での定義を示しています。

```
const string          str_example = "this is an example";

const long            long_example = 100;
const boolean         bool_example = TRUE;
```

このサンプルコードは，定数に対して得られた C++ コードを示しています。

```
const char *              str_example = "this is an example";

const CORBA::Long         long_example = 100;
const CORBA::Boolean      bool_example = 1;
```

インターフェース仕様の内部で定義された IDL の定数は，C++ インクルードファイル内で宣言され，C++ ソースファイル内で値が代入されます。次に例を示します。

このサンプルコードは，example.idl ファイル内の IDL 定義を示しています。

```
interface example {

    const string       str_example = "this is an example";
    const long         long_example = 100;
    const boolean      bool_example = TRUE;
};
```

このサンプルコードは，example_client.hh ファイルに生成された C++ コードを示しています。

```
class example :: public virtual CORBA::Object

{
    ...
    static const char *str_example;  /* これは例を示します。*/
    static const CORBA::Long         long_example; /* 100 */
    static const CORBA::Boolean      bool_example; /* 1 */
    ...
};
```

このサンプルコードは，example_client.cc ファイルに生成された C++ コードを示しています。

```
const char *example::str_example = "this is an example";

const CORBA::Long            example::long_example = 100;
const CORBA::Boolean         example::bool_example = 1;
```

定数に関する特殊なケース

一定の状況下で IDL コンパイラは，IDL 定数の名前ではなく，値を保持する C++ コードを生成する必要があります。たとえば，次のサンプルコードにあるように，C++ コードを正しくコンパイルするには，typedef V に対して定数 length の値を生成する必要があります。

次のサンプルコードは，値を持つ IDL 定数の定義を示しています。

```
// IDL
```

```
interface foo {
   const long length = 10;
   typedef long V[length];
};
```

次のサンプルコードは，C++ コードに生成された IDL 定数の値を示しています。

```
class foo : public virtual CORBA::Object

{
   const CORBA::Long length;
   typedef CORBA::Long V[10];
};
```

列挙体

IDL の列挙体は C++ の列挙体に直接マッピングされます。次に例を示します。

```
// IDL

enum enum_type {
   first,
   second,
   third
};
```

このサンプルコードは，C++ の enum に直接マッピングされた IDL の列挙体を示しています。

```
// C++ コード

enum enum_type {
   first,
   second,
   third
};
```

型定義

IDL の型定義は，C++ の型定義に直接マッピングされます。元の IDL 型定義が複数の C++ 型にマッピングされる場合，IDL コンパイラは，それぞれの型に対応するエリアスを C++ コードに生成します。次に例を示します。

```
// IDL

typedef octet               example_octet;
typedef enum enum_values {
   first,
   second,
   third
} enum_example;
```

次のサンプルコードは，IDL から C++ にマッピングされた単純な型定義の定義を示しています。

```
// C++

typedef octet               example_octet;
enum enum_values {
   first,
   second,
   third
};
typedef enum_values enum_example;
```

次のサンプルコードは，その他の型定義マッピング例を示しています。

次のサンプルコードは，インターフェースの IDL typedef を示しています。

```
// IDL

interface A1;
typedef A1 A2;
```

次のサンプルコードは，C++ にマッピングされた IDL インターフェースの型定義を示しています。

```
// C++

class A1;
typedef A1 *A1_ptr;
typedef A1_ptr A1Ref;
class A1_var;
typedef A1 A2;
typedef A1_ptr A2_ptr;
typedef A1Ref A2Ref;
typedef A1_var A2_var;
```

次のサンプルコードは，シーケンスの IDL typedef を示しています。

```
// IDL

typedef sequence<long> S1;
typedef S1 S2;
```

次のサンプルコードは，C++ にマッピングされた IDL シーケンスの型定義を示しています。

```
// C++

class S1;
typedef S1 *S1_ptr;
typedef S1_ptr S1Ref;
class S1_var;
typedef S1 S2;
typedef S1_ptr S2_ptr;
typedef S1Ref S2Ref;
typedef S1_var S2_var;
```

モジュール

OMG による IDL から C++ への言語マッピング仕様では，IDL の `module` は，同じ名前を持つ C++ の `namespace` へのマッピングが指定されています。ただし，現在のところ `namespaces` をサポートしているコンパイラはほとんどありません。そのため，VisiBroker では現在モジュールからクラスへのマッピングだけをサポートしています。次のサンプルコードは，VisiBroker の IDL コンパイラがどのようにモジュール定義をクラスにマッピングするかを示しています。

このサンプルコードは，IDL モジュール定義を示しています。

```
// IDL

module ABC
{
   ...
};
```

このサンプルコードは，生成された C++ クラスを示しています。

```
// C++

class ABC
{
   ...
};
```

複合データ型

ここでは，複合データ型が IDL から C++ にマッピングされるしくみについて説明します。

- Any 型
- string 型（固定長または可変長）
- sequence 型（固定長または可変長）
- オブジェクトリファレンス
- 可変長のメンバーを保持する別の structure または union
- 可変長の要素を持つ array
- 可変長の要素を持つ typedef

表 5.2　複合データ型の C++ へのマッピング

IDL 型	C++ マッピング
struct（固定長）	struct と _var クラス
struct（可変長）	struct と _var クラス 可変長メンバーは，それぞれ T_var クラスを使って宣言されます。
union	class と _var クラス
sequence	class と _var クラス
array	array，array_slice，array_forany，および array_var

構造体

固定長の構造体

C++ にマッピングされる各 IDL の固定長の構造体に対して，VisiBroker Edition の IDL コンパイラは，その構造体のほかに _var クラスを生成します。次のサンプルコードに，その使用方法を示します。_var クラスの詳細については，「<class_name>_var」を参照してください。

このサンプルコードは，固定長の構造体の定義を示しています。

```
// IDL

struct example {
   short a;
   long b;
};
```

次のサンプルコードは，固定長の IDL 構造体から C++ へのマッピングを示しています。

```
// C++

struct example {
   CORBA::Short a;
   CORBA::Long b;
};
class example_var
{
   ...
   private:
         example *_ptr;
};
```

固定長の構造体の使い方

_var クラスのフィールドにアクセスするには，必ず -> 演算子を使用する必要があります。たとえば，_var クラス ex2 のフィールドにアクセスするには，-> 演算子を使用する必要が

あります。ex2 がスコープの外に出ると，割り当てられていたメモリは自動的に解放されます。

次のサンプルコードは，example 構造体と example_var クラスの使用方法を示しています。

```
// example 構造体を宣言し，フィールドを初期化します。
example ex1 = { 2, 5 };
// _var クラスを宣言し，新しく作成した example 構造体を代入します。
// _var は，未初期化のフィールドを持つ構造体をポイントします。
example_var ex2 = new example;
// ex1 を使って ex2 のフィールドを初期化します。
ex2->a = ex1.b;
```

可変長の構造体

構造体に可変長のメンバーが含まれていると，固定長の構造体の場合とは異なる C++ コードが生成されます。たとえば，次のサンプルコードで示すように，example 構造体を可変長に変更した例を示します。ここでは，long メンバーを string に置き換え，またオブジェクトリファレンスを追加しています。

このサンプルコードは，可変長の構造体の定義を示しています。

```
// IDL
interface ABC {
   ...
};
struct vexample {
   short      a;
   ABC      c;
   string      name;
};
```

次のサンプルコードは，可変長の構造体から C++ へのマッピングを示しています。

```
// C++
struct vexample {
   CORBA::Short      a;
   ABC_var      c;
   CORBA::String_var      name;
   vexample&    operator=(const vexample& s);
};
class vexample_var {
   ...
};
```

ABC オブジェクトリファレンスを ABC_var クラスにマッピングする方法に注目してください。同様に，文字列 name は CORBA::String_var クラスにマッピングされます。さらに，可変長の構造体には代入演算子が生成されます。

構造体のメモリ管理

可変長の構造体で _var クラスを使用する場合，可変長のメンバーに割り当てられるメモリは透過的に管理されます。

- 構造体がスコープの外に出ると，可変長のメンバーに関連付けられていたすべてのメモリは自動的に解放されます。
- 構造体が初期化または代入され，その後，再び初期化または代入された場合，元のデータに関連付けられていたメモリは解放されます。

- 可変長のメンバーにオブジェクトリファレンスが代入される場合は，必ずそのオブジェクトリファレンスのコピーが作成されます。可変長のメンバーにポインタが代入される場合，コピーは作成されません。

共用体

各 IDL の union は，データメンバーの値を設定および取得するためのメソッドを持つ C++ クラスにマッピングされます。IDL 共用体の各メンバーは，アクセッサおよびミューテータとして機能する関数のセットにマッピングされます。**ミューテータ**関数はデータメンバーの値を設定します。**アクセッサ**関数はデータメンバーのデータを返します。

discriminant 型の _d という名前の特別な定義済みのデータメンバーも生成されます。共用体が作成されるときにこのディスクリミナントの値は設定されないため，union を使用する前にアプリケーションで値を設定する必要があります。提供されるメソッドの 1 つを使用していずれかのデータメンバーを設定すると，自動的にディスクリミナントが設定されます。ディスクリミナントにアクセスするには，特別なアクセッサ関数 _d() を使用します。

たとえば，次のサンプルコードでは，example_union 共用体から生成される C++ コードを示します。

次のサンプルコードは，構造体を保持する IDL 共用体を示しています。

```
// IDL

struct example_struct
{
   long abc;
};
union example_union switch(long)
{
   case 1: long       x; // プリミティブデータ型
   case 2: string      y; // 単純データ型
   case 3: example_struct       z; // 複合データ型
};
```

次のサンプルコードは，IDL 共用体から C++ クラスへのマッピングを示しています。

```
// C++

struct example_struct
{
        CORBA::Long       abc;
};
class example_union
{
   private:
        CORBA::Long       _disc;
        CORBA::Long       _x;
        CORBA::String_var       _y;
        example_struct       _z;
   public:
        example_union();
        ~example_union();
        example_union(const example_union& obj);
        example_union&    operator=(const example_union& obj);
        void    x(const CORBA::Long val);
        const CORBA::Long    x() const;
        void    y(char *val);
        void    y(const char *val);
        void    y(const CORBA::String_var& val);
        const char    *y() const;
        void z(const example_struct& val);
        const example_struct&    z() const;
        example_struct&    z();
        CORBA::Long _d();
        void _d(CORBA::Long);
```

```
        ...
};
```

次の表では，example_union クラスのメソッドの一部について説明します。

表 5.3　example_union クラスに生成されるメソッド

メソッド	説明
_d()	ディスクリミネータの値を返します。
_d(CORBA::Long)	ディスクリミネータの値を設定するために使用します。この例では，ディスクリミネータは long 型です。ディスクリミネータのデータ型により，入力引数の型が異なることに注意してください。
example_union()	デフォルトコンストラクタは，ディスクリミナントをデフォルト値に設定しますが，その他のデータメンバーは初期化しません。
example_union(const example_union& obj)	コピーコンストラクタは，ソースオブジェクトのディープコピーを実行します。
~example_union()	デストラクタは，共用体が所有しているすべてのメモリを解放します。
operator=(const example_union& obj)	代入演算子は，ディープコピーを実行し，必要に応じて古い格納領域を解放します。

共用体に対して管理される型

次のサンプルコードに示した example_union クラスのほかに，example_union_var も生成されます。_var クラスの詳細については，「<class_name>_var」を参照してください。

共用体のメモリ管理

共用体内の複合データ型のメモリ管理について，次の点に注意してください。

- アクセッサメソッドでデータメンバーに値を設定すると，ディープコピーが実行されます。アクセッサメソッドにパラメータを渡す場合，小さい型に対しては値を，大きい型に対しては定数リファレンスを使用してください。
- アクセッサメソッドでデータメンバーを設定する場合，それまでそのメンバーに関連付けられていたメモリは解放されます。代入されるメンバーがオブジェクトリファレンスの場合，そのオブジェクトのリファレンスカウントは，アクセッサメソッドが戻る前にインクリメントされます。
- char * を受け取るアクセッサメソッドは，渡されたポインタの所有権を取得する前に，格納領域をすべて解放します。
- const char * と String_var を受け取るアクセッサメソッドは，新しいパラメータの格納領域がコピーされる前に，古いメモリをすべて解放します。
- 配列データメンバーのアクセッサメソッドは，配列スライスへのポインタを返します。詳細については，44 ページの「配列スライス」を参照してください。

シーケンス

IDL シーケンスは，固定長と可変長のどちらも，現在の長さと最大の長さを持つ C++ クラスにマッピングされます。固定長シーケンスの最大長は，シーケンスの型によって定義されます。可変長シーケンスは，その C++ コンストラクタが呼び出されたときに，最大長を指定できます。現在の長さは，プログラム的に変更できます。次のサンプルコードに示すように，IDL シーケンスは，アクセッサメソッドを持つ C++ クラスにマッピングされます。

メモ　可変長シーケンスの長さが指定の最大長を超えた場合，VisiBroker は，より大きいバッファを透過的に割り当て，古いバッファを新しいバッファにコピーして古いバッファに割り当てられていたメモリを解放します。ただし，最大長が減少しても，未使用のメモリは解放されません。

このサンプルコードは，IDL の可変長シーケンスを示しています。

```
// IDL
typedef sequence<long> LongSeq;
```

次のサンプルコードは，IDL 可変長シーケンスから C++ クラスへのマッピングを示しています。

```
// C++
class LongSeq
{
   public:
         LongSeq(CORBA::ULong max=0);
         LongSeq(CORBA::ULong max=0, CORBA::ULong length,
                       CORBA::Long *data, CORBA::Boolean release = 0);
         LongSeq(const LongSeq&);
         ~LongSeq();
         LongSeq&          operator=(const LongSeq&);
         CORBA::ULong          maximum() const;
         void         length(CORBA::ULong len);
         CORBA::ULong          length() const;
         const CORBA::ULong&          operator[](CORBA::ULong index) const;
         ...
         static LongSeq          *_duplicate(LongSeq* ptr);
         static void          _release(LongSeq *ptr);
         static CORBA::Long          *allocbuf(CORBA::ULong nelems);
         static void          freebuf(CORBA::Long *data);
   private:
         CORBA::Long      * _contents;
         CORBA::ULong       _count;
         CORBA::ULong       _num_allocated;
         CORBA::Boolean       _release_flag;
         CORBA::Long       _ref_count;
};
```

表 5.4　可変長シーケンスに対して生成されたメソッドの一覧

メソッド	説明
LongSeq(CORBA::ULong max=0)	可変長シーケンスのコンストラクタは，引数として最大長を受け取ります。固定長シーケンスは，定義済みの最大長を持ちます。
LongSeq(CORBA::ULong max=0, CORBA::ULong length, CORBA::Long *data, CORBA::Boolean release=0)	このコンストラクタを使用すると，最大長，現在の長さ，関連付けられるデータバッファへのポインタ，および解放フラグを設定できます。release が 0 以外の場合，VisiBroker は，シーケンスのサイズを増加するときに，データバッファに関連付けられているメモリを解放します。release が 0 の場合，古いデータバッファのメモリは解放されません。固定長シーケンスは，max 以外のパラメータをすべて持ちます。
LongSeq(const LongSeq&)	コピーコンストラクタは，ソースオブジェクトのディープコピーを実行します。
~LongSeq();	デストラクタは，構築時に解放フラグの値が 0 以外の場合にだけ，シーケンスが所有するすべてのメモリを解放します。
operator=(const LongSeq&j)	代入演算子は，ディープコピーを実行し，必要に応じて古い格納領域を解放します。
maximum()	シーケンスのサイズを返します。
length()	シーケンスの長さを設定および返すための 2 つのメソッドが定義されます。
operator[]()	シーケンス内の要素にアクセスするための 2 つの添え字演算子が提供されます。一方の演算子を使用すると，要素を変更できます。もう一方の演算子を使用すると，読み取り専用で要素にアクセスできます。

表 5.4　可変長シーケンスに対して生成されたメソッドの一覧（続き）

メソッド	説明
_release()	シーケンスを解放します。シーケンスの要素の型が文字列またはオブジェクトリファレンスであり，オブジェクトの作成時にコンストラクタの解放フラグが 0 以外であった場合は，バッファが解放される前に，各要素が解放されます。
allocbuf() freebuf()	シーケンスが使用するメモリを割り当てたり，解放する場合は，この 2 つの static メソッドを使用する必要があります。

シーケンスに対して管理される型

次のサンプルコードに示した LongSeq クラスのほかに，LongSeq_var も生成されます。クラスの詳細については，「<class_name>_var」を参照してください。通常のメソッドのほか，シーケンスには 2 つの添え字演算子が定義されます。

```
CORBA::Long& operator[](CORBA::ULong index);

const CORBA::Long& operator[](CORBA::ULong index) const;
```

シーケンスのメモリ管理

次にリストアップしているメモリ管理に関する注意点をよくお読みください。このサンプルコードは，このリストで挙げた点に関する C++ コードを示しています。

- シーケンスの作成時に解放フラグを 0 以外の値に設定した場合は，シーケンスがユーザーのメモリを管理します。要素が代入されると，シーケンスは，式の右辺のメモリの所有権を取得する前に，古いメモリを解放します。
- 文字列またはオブジェクトリファレンスを保持するシーケンスの作成時に，解放フラグを 0 以外の値に設定した場合は，シーケンス内容のバッファが解放され，オブジェクトが破棄される前に，各要素が解放されます。
- 解放フラグを 1 に設定しなかった場合，[] 演算子を使ってシーケンスの要素を代入すると，メモリ管理に関するエラーが発生する可能性があります。
- 解放フラグを 0 に設定して作成したシーケンスを入力／出力パラメータとして使用してはなりません。オブジェクトサーバーでメモリ管理に関するエラーが発生する可能性があります。
- シーケンスで使用される格納領域を作成および解放する場合は，必ず allocbuf と freebuf を使用してください。

このサンプルコードは，可変長シーケンスの IDL 仕様を示しています。

```
// IDL

typedef sequence<string, 3> String_seq;
```

次のサンプルコードは，2 つの固定長シーケンスを使ったメモリ管理の例です。

```
// C++

char *static_array[] = {"1", "2", "3"};
char *dynamic_array = StringSeq::allocbuf(3);
```

// Create a sequence, release flag is set to **FALSE** by default

```
StringSeq static_seq(3, static_array);
// 別のシーケンスを作成し，解放フラグを TRUE に設定します。
StringSeq dynamic_seq(3, dynamic_array, 1);

static_seq[1] = "1"; // old memory not freed, no copying occurschar *str =
string_alloc(2);

dynamic_seq[1] = str;              // 古いメモリが解放され，コピーはされません
```

配列

IDL の配列は，C++ の配列にマッピングされます。C++ 配列は静的に初期化できます。配列要素が文字列またはオブジェクトリファレンスである場合，C++ 配列の要素の型は _var になります。次の表は，要素の型が異なる 3 つの配列を示します。

このサンプルコードは，IDL 配列定義を示しています。

```
// IDL

interface Intf
{
   ...
};
typedef long L[10];
typedef string S[10];
typedef Intf A[10];
```

次のサンプルコードは，IDL 配列から C++ 配列へのマッピングを示しています。

```
// C++

typedef CORBA::Long L[10];
typedef CORBA::String_var S[10];
typedef Intf_var A[10];
```

文字列とオブジェクトリファレンスに対して管理される型 _var を使用すると，配列要素の代入時にメモリを透過的に管理できます。

配列スライス

配列 _slice 型は，多次元配列をパラメータとして渡す場合に使用されます。VisiBroker の IDL コンパイラは，配列に対して，その最初の次元を除いたすべての次元を持つ _slice 型も生成します。パラメータを渡したり，返すには，配列の _slice 型を使用すると便利です。次のサンプルコードは，_slice 型の例を 2 つ示しています。

このサンプルコードは，多次元配列の IDL 定義を示しています。

```
// IDL

typedef long L[10];
typedef string str[1][2][3];
```

次のサンプルコードは，_slice 型の生成を示しています。

```
// C++
typedef CORBA::Long L_slice;
typedef CORBA::String_var str_slice[2][3];
```

配列に対して管理される型

VisiBroker の IDL コンパイラは，IDL 配列に対応する C++ 配列を生成するほかに，_var クラスも生成します。このクラスは，配列に次の追加機能を提供します。

- 配列要素をわかりやすくアクセスできるように，operator[] がオーバーロードされます。
- 引数として配列の _slice オブジェクトへのポインタを受け取るコンストラクタと代入演算子が提供されます。

次のサンプルコードは，配列の IDL 定義を示しています。

```
// IDL

typedef long L[10];
```

このサンプルコードは，配列の _var クラス生成を示しています。

```
// C++

class L_var
{
```

```
    public:
        L_var();
        L_var(L_slice *slice);
        L_var(const L_var& var);
        ~L_var();
        L_var&    operator=(L_slice *slice);
        L_var&    operator=(const L_var& var);
        CORBA::Long&    operator[](CORBA::ULong index);
        operator L_slice *();
        operator L &() const;
        ...
    private:
        L_slice   *_ptr;
};
```

タイプセーフ配列

any 型にマッピングされる要素を持つ配列を処理するために，特別な _forany クラスが生成されます。_var クラスの場合と同様に，_forany クラスを使用すると，基底の配列型にアクセスできます。_any 型がメモリの所有権を維持するため，_forany クラスは，破棄時にメモリを解放しません。オーバーロードが正しく機能するには，ほかの型と区別できなければならないため，_forany クラスは typedef としては実装されません。

このサンプルコードは，IDL 配列定義を示しています。

```
// IDL
typedef long L[10];
```

このサンプルコードは，IDL 配列の _for クラス生成を示しています。

```
// C++
class L_forany
{
    public:
        L_forany();
        L_forany(L_slice *slice);
        ~L_forany();
        CORBA::Long&    operator[](CORBA::ULong index);
        const CORBA::Long&    operator[](CORBA::ULong index) const;
        operator L_slice *();
        operator L &() const;
        operator const L & () const;
        operator const L& () const;
        L_forany& operator=(const L_forany obj);
        ...
    private:
        L_slice   *_ptr;
};
```

配列のメモリ管理

VisiBroker の IDL コンパイラは，配列に関連付けられたメモリを割り当て，複製，コピー，および解放するための 4 つの関数を生成します。これらの関数を使用すると，ORB によってメモリが管理され，new 演算子と delete 演算子をオーバーライドする必要がありません。

このサンプルコードは，IDL 配列定義を示しています。

```
// IDL
typedef long L[10];
```

このサンプルコードは，配列のメモリを割り当ておよび解放するために生成されるメソッドです。

```
// C++
inline L_slice *L_alloc();
// 配列を動的に割り当てます。
```

```
// 失敗すると NULL が返されます
inline void L_free(L_slice *data);
// L_alloc を使って割り当てられた
// L_alloc.
inline void L_copy(L:slice *_to, L_slice *_from)
//_from 配列の内容を _to 配列にコピーします。
inline L_slice *L_dup(const L_slice *_date)
// 新しくコピーされた _date 配列を返します。
```

プリンシパル

Principal は，object インプリメンテーションに対して処理を要求しようとするクライアントアプリケーションに関する情報を表します。Principal の IDL インターフェースで定義されるオペレーションはありません。Principal は，Octet のシーケンスとして実装されます。Principal はクライアントアプリケーションによって設定され，VisiBroker ORB インプリメンテーションによって検査されます。VisiBroker for C++ では，Principal を不透過な型として処理します。その内容が VisiBroker ORB によって検査されることはありません。

valuetype

IDL の valuetype は，それと同じ名前で C++ クラスにマッピングされます。このクラスは，valuetype の各状態メンバーに対応する純粋仮想関数（アクセッサとモディファイア）および valuetype の各オペレーションに対応する純粋仮想関数を持つ抽象基底クラスです。

valuetype の完全スコープ名に「OBV_」を追加して名前を形成する C++ クラスは，抽象基底クラスのアクセッサとモディファイアのデフォルトインプリメンテーションを提供します。

valuetype のインスタンスはアプリケーションで作成する必要があります。インスタンスの作成後，アプリケーションでは，ポインタだけを使用してそれらのインスタンスを処理します。オブジェクトリファレンスが，実際の C++ ポインタまたは類似のオブジェクトとして実装される C++ の _ptr types 型にマッピングされるのとは異なり，C++ valuetype インスタンスのハンドルは実際の C++ ポインタです。これは，オブジェクトリファレンスとの区別に役立ちます。

インターフェースでのマッピングとは異なり，valuetype のリファレンスカウントは，valuetype のインスタンスで実装します。valuetype の _var 型は，リファレンスカウントを自動化します。次のサンプルコードは，これらの機能について示しています。

```
valuetype Example {

 Short op1();
Long  op2( in Example x );
Private short val1;
Public long  val2;
};
```

次のサンプルコードでは，3 つのクラスについて，IDL 定義から C++ へのマッピングを示しています。

```
class  Example  : public virtual CORBA::ValueBase   {
 public:
    virtual CORBA::Short op1() = 0;
    virtual CORBA::Long op2(Example_ptr _x) = 0;
 // すべてのパブリックな状態に対する純粋仮想ゲッター／セッター関数
 // アクセッサはリファレンスによって読み取り／書き込みアクセスが可能なので
 // これらのアクセッサは C++ の共用体メンバーに似ています。
 virtual void val2(const CORBA::Long _val2) = 0;
 virtual const CORBA::Long val2() const = 0;
protected:
```

```
 Example() {}
 virtual ~Example() {}
 virtual void val1(const CORBA::Short _val1) = 0;
 virtual const CORBA::Short val1() const = 0;
    private:
 void operator=(const Example&);
 };
class OBV_Example: public virtual Example{
 public:
 virtual void val2(const CORBA::Long _val2) {
 _obv_val2 = _val2;
 }
 virtual const CORBA::Long val2() const {
 return _obv_val2;
 }
protected:
 virtual void val1(const CORBA::Short _val1) {
 _obv_val1 = _val1;
 }
 virtual const CORBA::Short val1() const {
 return _obv_val1;   }
 OBV_Example() {}
 virtual ~OBV_Example() {}
 OBV_Example(const CORBA::Short _val1,
      const CORBA::Long _val2) {
    _obv_val1 = _val1;
    _obv_val2 = _val2;
 }
 CORBA::Short _obv_val1;
 CORBA::Long _obv_val2;
 };
class Example_init : public CORBA::ValueFactoryBase {
 };
```

_init クラスは，valuetype のファクトリを実装する方法を提供します。valuetype は，有線上を値で渡されるため，通常は，ストリームとして送信された valuetype の受信側がファクトリを実装して，ストリームから valuetype のインスタンスを作成します。ストリームを介して valuetype を受信する可能性がある場合は，サーバーとクライアントの両方でファクトリを実装する必要があります。次のサンプルコードに示すように，_init クラスでは，CORBA::ValueBase * を返す create_for_unmarshal も実装する必要があります。

このサンプルコードは，_init クラスの例を示しています。

```
class Example_init_impl: public Example_init{

public:
   Example_init; _impl();
   virtual ~Example_init();
CORBA::ValueBase * create_for_unmarshal() {
 ...// Example_ptr を返します。
 }
};
```

次のように，valuetype をほかの valuetype から派生させることができます。

このサンプルコードは，ほかの valuetype から派生された valuetype の IDL を示しています。

```
valuetype DerivedExample: Example{

 Short op3();
};
```

DerivedExample クラスの C++ インターフェースは次のとおりです。

```
// IDL valuetype: DerivedExample

class  DerivedExample  : public virtual Example   {
 public:
```

```
    virtual CORBA::Short op3() = 0;
  protected:
    DerivedExample() {}
    virtual ~DerivedExample() {}
  private:
    void operator=(const DerivedExample&);
};
class OBV_DerivedExample: public virtual DerivedExample, public virtual OBV_Example{
  protected:
    OBV_DerivedExample() {}
    virtual ~OBV_DerivedExample() {}
};
class DerivedExample_init : public CORBA::ValueFactoryBase { };
```

次のサンプルコードに示すように，派生された valuetype はベースの valuetype に切り詰めることができます。これは，ストリームの受信側で，派生 valuetype を構築できず，ベースの valuetype しか構築できない場合に必要です。

このサンプルコードは，切り詰められた派生 valuetype を示しています。

```
valuetype DerivedExample : truncatable Example { };
```

DerivedExample クラスの Type 情報に，基底クラス Example が切り詰め可能であることを示す情報が追加される点を除くと，この場合のマッピングは，通常の派生 valuetype と同じです。

valuetype をインターフェースから派生させることはできませんが，インターフェースのすべてのオペレーションを提供することにより，1 つ以上のインターフェースをサポートすることはできます。IDL のキーワード **supports,** は，この目的で使用されます。

このサンプルコードは，派生した valuetype をサポートしている IDL キーワードを示しています。

```
interface myInterface{

  long op5();
};
valuetype IderivedExample supports myInterface {
  Short op6();
};
```

この IDL 定義に対する C++ マッピングは次のとおりです。

このサンプルコードは，派生した valuetype の C++ を示しています。

```
// IDL valuetype: DerivedExample

class  IderivedExample  : public virtual CORBA::ValueBase   {
  public:
    virtual CORBA::Short op6() = 0;
    virtual CORBA::Long op5() = 0;
  protected:
    IderivedExample() {}
    virtual ~IderivedExample() {}
  private:
    void operator=(const IderivedExample&);
};
class OBV_IderivedExample: public virtual IderivedExample{
  protected:
    OBV_IderivedExample() {}
    virtual ~OBV_IderivedExample() {}
};
```

リファレンスカウント用に，C++ マッピングには 2 つの標準クラスがあります。CORBA::DefaultValueRefCountBase クラスは，アプリケーションが提供する具象 valuetype がどの IDL インターフェースからも派生しない場合に，基底クラスとして機能します。このような valuetype に対しても，アプリケーションは，独自のリファレンスカウントメカニズムを自由に実装できます。もう 1 つの PortableServer::ValueRefCountBase クラスは，

アプリケーションが提供する具象 valuetype クラスが 1 つ以上の IDL インターフェースから派生する場合に，基底クラスを提供します。

値ボックス

valuebox は，構造体，共用体，any，文字列，プリミティブ型，オブジェクトリファレンス，列挙体，シーケンス，および配列の各型に適用される valuetype です。これらの型は，メソッド，継承，およびインターフェースをサポートしません。valuebox にはリファレンスカウントがあり，`CORBA::DefaultValueRefCountBase` から派生します。マッピングは，基底の型によって異なります。すべての valuebox の C++ クラスは，基礎になる型へのマッピング用に `_boxed_in()`，`_boxed_out()`，および `_boxed_inout()` を提供します。valuebox のファクトリは，生成されたスタブによって自動的に登録されます。

詳細については，『CORBA 2.3 IDL2CPP specification』の「Chapter 1.17」を参照してください。valuebox のファクトリは，生成されたスタブによって自動的に登録されます。

抽象インターフェース

抽象インターフェースは，オブジェクトがリファレンス（IOR）によって渡されるか，それとも値（valuetype）によって渡されるかを実行時に判定する場合に使用されます。この目的で，プレフィクス「abstract」をインターフェース宣言の前で使用します。

このサンプルコードは，IDL サンプルコードを示しています。

```
abstract interface foo {
   Void func():
}
```

抽象インターフェースをサポートする valuetype は，その抽象インターフェースとして渡すことができます。抽象インターフェースは次のように宣言されます。

```
valuetype vt supports foo {
   ...
};
```

同様に，抽象インターフェースとして渡す必要があるインターフェースは，次のように宣言されます。

```
interface intf : foo {
}
```

前に宣言されている抽象インターフェース foo に対する C++ マッピングにより，次のクラスが生成されます。

```
class foo_var : public CORBA::_var{
   ...
}
class foo_out{
   ...
};
class foo : public virtual CORBA:::AbstractBase{
   private:
       ...
      void operator=(const foo&) {}
   protected:
      foo();
      foo(const foo& ref) {}
      virtual ~foo() {}
   public:
      static CORBA::Object* _factory():
      foo_ptr _this();
      static foo_ptr _nil() { ... }
      static foo_ptr _narrow(CORBA::AbstractBase* _obj);
      static foo_ptr _narrow(CORBA::Object_ptr _obj);
```

```
        static foo_ptr _narrow(CORBA::ValueBase_ptr _obj);
        virtual void func() = 0;
        ...
};
class _vis_foo_stub : public virtual foo, public virtual CORBA_Object {
    public :
        _vis_foo_stub() {}
        virtual ~_vis_foo_stub() {}
        ...
        virtual void func():
}
```

このサンプルには，_var クラス，_out クラス，および前のサンプルコードで説明したメソッドを実装する CORBA::AbstractBase の派生クラスがあります。

第 6 章

VisiBroker のプロパティ

ここでは，Borland VisiBroker のプロパティについて説明します。

スマートエージェントおよびスマートエージェント通信のプロパティ

表 6.1　スマートエージェントのプロパティ

プロパティ	デフォルト値	以前のプロパティ	説明
vbroker.agent.addrFile	null	ORBagentAddrFile	スマートエージェントが稼動するホストの IP アドレスまたはホスト名を格納しているファイルを指定します。
vbroker.agent.localFile	null	N/A	マルチホームマシン上で使用するネットワークインターフェースを指定します。以前は OSAGENT_LOCAL_FILE 環境変数でした。
vbroker.agent.clientHandlerPort	null	N/A	スマートエージェントがクライアント（この場合は，VisiBroker アプリケーション）の存在を確認するために使用するポートを指定します。デフォルト値 null を使用すると，スマートエージェントはランダムなポート番号を使用して接続します。
vbroker.agent.port	14000	ORBagentPort	使用しているネットワーク上のドメインを定義するポート番号を指定します。VisiBroker アプリケーションとスマートエージェントのポート番号が同じ場合は，相互が協調して機能します。このプロパティは，OSAGENT_PORT 環境変数と同じです。

次の表で説明するプロパティは，スマートエージェント通信のために ORB によって使用されます。

プロパティ	デフォルト値	以前のプロパティ	説明
vbroker.agent.keepAliveTimer	120	N/A	ORB がスマートエージェントにキープアライブメッセージを送信する 時間間隔（秒単位）です (クライアントとサーバーの両方に適用可能)。有効な値は 1 〜 120 の範囲の整数です。
vbroker.agent.retryDelay	0	N/A	スマートエージェントとの接続が切断された場合に，スマートエージェントへの再接続を試みる前にプロセスが一時停止する時間（秒単位）です。値が -1 の場合，スマートエージェントとの接続が切断されると，プロセスは終了します。デフォルト値 0 は，一時停止なしで再接続が行われることを意味します。
vbroker.agent.addr	null	ORBagentAddr	スマートエージェントが稼動するホストの IP アドレスまたはホスト名を指定します。デフォルト値の null は，VisiBroker アプリケーションに OSAGENT_ADDR 環境変数の値を使用するように指示します。OSAGENT_ADDR 変数が設定されていない場合は，スマートエージェントがローカルホストで動作しているとみなされます。
vbroker.agent.addrFile	null	ORBagentAddrFile	スマートエージェントが稼動するホストの IP アドレスまたはホスト名を格納しているファイルを指定します。
vbroker.agent.debug	false	ORBdebug	true に設定した場合は，システムが VisiBroker アプリケーションとスマートエージェントとの通信に関するデバッグ情報を表示します。
vbroker.agent.enableLocator	true	ORBdisableLocator	false に設定した場合，VisiBroker アプリケーションはスマートエージェントと通信できません。
vbroker.agent.port	14000	ORBagentPort	使用しているネットワーク上のドメインを定義するポート番号を指定します。VisiBroker アプリケーションとスマートエージェントのポート番号が同じ場合は，相互が協調して機能します。このプロパティは，OSAGENT_PORT 環境変数と同じです。
vbroker.agent.failOver	true	ORBagentNoFailOver	true に設定した場合は，VisiBroker アプリケーションは，別のスマートエージェントにフェイルオーバーできます。

プロパティ	デフォルト値	以前のプロパティ	説明
vborker.agent.clientPort	0	N/A	DSUser ソケットにバインドするポート番号の最初の値を指定します。
vbroker.agent.clientPortRange	0	N/A	DSUser ソケットにバインドするポート番号の範囲を指定します。このプロパティは，vbroker.agent.clientPort プロパティと組み合わせて使用する必要があります。

VisiBroker ORB のプロパティ

次の表に，VisiBroker ORB のプロパティを示します。

表 6.2　VisiBroker ORB のプロパティ

プロパティ	デフォルト値	説明
vbroker.orb.cacheDSQuery	true	true に設定した場合は，VisiBroker アプリケーションが IOR をキャッシュできます。
vbroker.orb.rebindForward	0	この値は，クライアントが転送先ターゲットへの接続を試行する回数を決定します。ネットワーク障害などの理由で，クライアントが転送先ターゲットと通信できない場合に，このプロパティを使用できます。デフォルト値 0 は，クライアントが接続を試行し続けることを意味します。
vbroker.orb.activationIOR	null	起動されたサーバーは，それを起動した OAD とのコンタクトを容易に確立できます。
vbroker.orb.oadUID	0	サーバーを起動した OAD がまだ存在するかどうかを確認するために使用されます。値 1 は，OAD がまだ実行中であることを示します。
vbroker.orb.propStorage	null	プロパティ値を保持するプロパティファイルを指定します。
vbroker.orb.backCompat	FALSE	TRUE に設定した場合，サーバーは下位互換モードで動作します。
vbroker.orb.nullstring	FALSE	TRUE に設定した場合は，ヌル文字列をマーシャリングできます。ただし，このプロパティは使用されなくなり，vbroker.orb.enableNullString プロパティに置き換えられています。
vbroker.orb.admDir	null	さまざまなシステムファイルを置く管理ディレクトリを指定します。このプロパティは，VBROKER_ADM 環境変数を使用して設定できます。
vbroker.orb.isNTService	FALSE	TRUE に設定すると，このプロパティとコンパイルフラグ WIN32 を組み合わせれば，ユーザーがログアウトしても NT サービス／COM+ アプリケーションの実行を続けることができます。
vbroker.orb.enableServerManager	FALSE	TRUE に設定した場合は，サーバーの起動時にサーバーマネージャが有効になり，クライアントからアクセスできるようになります。
vbroker.orb.input.maxBuffers	16	プール内に保持される入力バッファの最大数を指定します。
vbroker.orb.input.buffSize	255	入力バッファのサイズを指定します。
vbroker.orb.output.maxBuffers	16	プール内に保持される出力バッファの最大数を指定します。
vbroker.orb.output.buffSize	255	出力バッファのサイズを指定します。

表 6.2　VisiBroker ORB のプロパティ（続き）

プロパティ	デフォルト値	説明
vbroker.orb.initRef	null	初期リファレンスを指定します。文字列化された IOR に加えて，corbaloc などのオブジェクト URL フォーマットも使用できます。文字列化された IOR がファイル内にある場合は，下で説明されている "file://" URL もサポートされます。
vbroker.orb.defaultInitRef	null	デフォルトの初期リファレンスを指定します。文字列化された IOR に加えて，corbaloc などのオブジェクト URL フォーマットも使用できます。文字列化された IOR がファイル内にある場合は，下で説明されている "file://" URL もサポートされます。
vbroker.orb.boa_map.TSingle	boa_s	シングルスレッドの BOA bid ポリシーを boa_s にマップします。
vbroker.orb.boa_map.TPool	boa_tp	スレッドプールの BOA ビッドポリシーを boa_tp にマップします。
vbroker.orb.boa_map.TSession	boa_ts	スレッドセッションの BOA bid ポリシーを boa_ts にマップします。
vbroker.orb.boa_map.TPool_LIOP	boa_ltp	ローカルスレッドプールの BOA ビッドポリシーを boa_ltp にマップします。
vbroker.orb.alwaysProxy	false	true に設定した場合は，クライアントが必ず GateKeeper を使用してサーバーに接続する必要があることを示します。
vbroker.orb.gatekeeper.ior	null	IOR の提供元の GateKeeper を介してサーバーに接続するようにクライアントアプリケーションに強制します。
vbroker.locator.ior	null	スマートエージェントへのプロキシとして使用される GateKeeper の IOR を指定します。このプロパティを設定しなかった場合，vbroker.orb.gatekeeper.ior プロパティで指定されている GateKeeper がこの目的のために使用されます。詳細は，『VisiBroker GateKeeper ガイド』を参照してください。
vbroker.orb.exportFirewallPath	false	サーバーが公開する任意のサーバントの IOR の一部としてファイアウォール情報を挿入するようにサーバーアプリケーションに強制します。これを POA ごとに選択的に指定するには，コード内で Firewall::FirewallPolicy を使用します。
vbroker.orb.proxyPassthru	false	true に設定された場合は，アプリケーションスコープ内でグローバルに PASSTHROUGH ファイアウォールモードを強制します。これをオブジェクトごと，または ORB ごとに選択的に指定するには，コード内で QoSExt::ProxyModePolicy を使用します。
vbroker.orb.bids.critical	inprocess	クリティカルビッドは，ビッド順の中のどこで指定されていても，最高の優先順位を持ちます。クリティカルビッドの値が複数ある場合，それらの相対的な重要度は，bidOrder プロパティによって決まります。

表 6.2 VisiBroker ORB のプロパティ（続き）

プロパティ	デフォルト値	説明
vbroker.orb.bidOrder	inprocess:liop:ssl:iiop:proxy:hiop:locator	さまざまなトランスポートについて，重要度の相対的な順序を指定できます。トランスポートには次の優先順位が指定されています。 **1** inprocess **2** liop **3** ssl **4** iiop **5** proxy **6** hiop **7** locator 上にあるトランスポートほど優先順位が高くなります。その例を次に示します。たとえば，IOR が LIOP と IIOP の両方のプロファイルを持つ場合，最初に使用されるのは LIOP です。LIOP が失敗した場合にのみ，IIOP を試します。vbroker.orb.bids.critical プロパティで指定されるクリティカルビッドは，ビッド順の中のどこで指定されていても，最高の優先順位を持ちます。
vbroker.orb.dynamicLibs	null	VisiBroker ORB が使用できるサービスのリストを指定します。各サービスは，コンマで区切られます。
vbroker.orb.embedCodeset	true	IOR が作成されると，VisiBroker ORB はその IOR にコードセットのコンポーネントを埋め込みます。これは，一部の非準拠 ORB で問題になる可能性があります。embedCodeset プロパティをオフにすると，IOR にコードセットを埋め込まないように Visibroker ORB に指示できます。false に設定すると，クライアントとサーバーの間で文字とワイド文字の変換がネゴシエートされません。
vbroker.orb.enableVB4backcompat	false	このプロパティは，VisiBroker 4.0 と 4.1 で GIOP 1.2 に準拠しない動作を処理するための回避策を有効にします。特に GateKeeper が関係している場合，VisiBroker 4.1.1 またはそれ以前のリリースで実行されている VisiBroker クライアントは，どれも影響を受けます。Visibroker 4.0 または 4.1 クライアントで作業する場合，このフラグを true に設定する必要があります。これは，サーバー側だけのフラグです。クライアント側に対応するフラグはありません。
vbroker.orb.enableNullString	false	TRUE に設定した場合は，ヌル文字列をマーシャリングできます。
vbroker.orb.procId	0	サーバーのプロセス ID を指定します。
vbroker.orb.usingPoll	true	UNIX プラットフォームでは，ORB は，このプロパティの値に基づいて，I/O 多重化のためにシステム呼び出し select() または poll() を使用します。値が true の場合は，poll() を使用します。そうでない場合は，select() を使用します。True がデフォルト値です。

ファイルの URL は，“file:// *ドメイン名* / *パス* / *ファイル*” の標準フォーマットにしたがいます。ただし，VisiBroker for C++ がサポートするフォーマットにはいくつかの制約があります。

- URL の*プロトコル*部分は「file://」である必要があります。

- URL の*ドメイン名*は空である必要があります。
- すべてのパスは絶対パスで指定されます（相対パスは使用できません）。
- パスには文字「:」を入れることができません。パス区切り文字は「/」にする必要があります。
- Windows の場合は，ドライブ文字のコロン「:」を「|」に置き換える必要があります。

次は有効なパスの例です。

- file:///home/user/appl.ior
- file:///C|/My Documents/User/root.txt

サーバーマネージャのプロパティ

次の表は，サーバーマネージャのプロパティの一覧です。

表 6.3　サーバーマネージャのプロパティ

プロパティ	デフォルト値	説明
vbroker.serverManager.name	null	サーバーマネージャの名前を指定します。
vbroker.serverManager.enableOperations	true	true に設定した場合は，サーバーマネージャがオペレーションをエクスポーズして，それを呼び出すことができるようにします。
vbroker.serverManager.enableSetProperty	true	true に設定した場合は，サーバーマネージャがプロパティをエクスポーズして，それを変更できるようにします。

追加プロパティ

ここでは，サーバーマネージャでサポートされている新しいプロパティについて説明します。これらのプロパティは，コンテナを介して照会できます。

サーバー側のリソース使用量関連のプロパティ

プロパティ	説明
vbroker.se.<SE_name>.scm.<SCM_name>.manager.allocatedFileDescriptors	サーバー接続マネージャ（SCM）が使用している現在のファイルデスクリプタ数。この値は，通常，現在の着信接続の数に，リスナーによって使用される 2 つを足した数になります。
vbroker.se.<SE_name>.scm.<SCM_name>.manager.maxFileDescriptor	SCM でのファイルデスクリプタの最大値。
vbroker.se.<SE_name>.scm.<SCM_name>.manager.inUseConnections	ORB で実行中の要求がある着信接続の数。
vbroker.se.<SE_name>.scm.<SCM_name>.manager.idleConnections	ORB で現在実行中の要求がない着信接続の数。
vbroker.se.<SE_name>.scm.<SCM_name>.manager.idledTimeoutConnections	アイドルタイムアウト設定を過ぎたが，（ガベージコレクションの制限などの理由により）アイドル状態のまま閉じられていないアイドル接続の数。
vbroker.se.<SE_name>.scm.<SCM_name>.dispatcher.inUseThreads	現在，ディスパッチャ内で要求を実行中のスレッドの数。
vbroker.se.<SE_name>.scm.<SCM_name>.dispatcher.idleThreads	作業の割り当てを待機しているアイドル状態のスレッドの数。

クライアント側のリソース使用量関連のプロパティ

プロパティ	説明
vbroker.ce.<CE_name>.ccm.maxFileDescriptor	クライアント接続マネージャ（CCM）内のファイルデスクリプタの最大数。
vbroker.ce.<CE_name>.ccm.activeConnections	アクティブプール内の接続の数。つまり，オブジェクトリファレンスがこれらの接続を使用しています。
vbroker.ce.<CE_name>.ccm.cachedConnections	キャッシュプール内の接続の数。つまり，これらの接続を使用しているオブジェクトリファレンスはありません。
vbroker.ce.<CE_name>.ccm.inUseConnections	保留中の要求がある発信接続の数。
vbroker.ce.<CE_name>.ccm.idleConnections	保留中の要求がない発信接続の数。
vbroker.ce.<CE_name>.ccm.idledTimeoutConnections	アイドルタイムアウト設定を過ぎたが，アイドル状態のまま閉じられていないアイドル接続の数。

スマートエージェント関連のプロパティ

プロパティ	説明
vbroker.agent.currentAgentIP	現在の ORB のスマートエージェントの IP アドレス。
vbroker.agent.currentAgentClientPort	ORB が要求を送信している先のスマートエージェントのポート。

その他のプロパティ

プロパティ	説明
vbroker.env.path	ORB が実行されている PATH 環境変数の値。
vbroker.env.shlibPath	共有ライブラリパス環境変数の値。HP-UX では，これは SHLIB_PATH 環境変数に対応します。
vbroker.env.orbVersion	現在ロードされている ORB の ORB バージョン。HP-UX では，vbver liborb_r.sl を実行して取得することもできます。
vbroker.process.fileDescriptorLimit	現在のプロセスのファイルデスクリプタの最大数。
vbroker.orb.uid	VisiBroker アプリケーションを開始したユーザーのユーザー ID。
vbroker.orb.commandLine	CORBA::ORB_init メソッドに渡されたコマンドライン引数。

ロケーションサービスのプロパティ

次の表は，ロケーションサービスのプロパティの一覧です。

プロパティ	デフォルト値	説明
vbroker.locationservice.debug	false	true に設定した場合は，ロケーションサービスがデバッグ情報を表示できます。
vbroker.locationservice.verify	false	true に設定した場合，ロケーションサービスは，スマートエージェントから送信されたオブジェクトリファレンスの参照先のオブジェクトが存在するかどうかを調べることができます。存在を確認する対象は，BY_INSTANCE に登録されているオブジェクトだけです。OAD または BY_POA ポリシーに登録されているオブジェクトの存在は確認の対象外です。
vbroker.locationservice.timeout	1	ロケーションサービスと対話する場合の接続，受信，および送信のタイムアウト（秒）を指定します。

イベントサービスのプロパティ

次の表は，イベントサービスのプロパティの一覧です。

プロパティ	デフォルト値	説明
vbroker.events.maxQueueLength	100	動作が遅いコンシューマのキューに入るメッセージの数を指定します。
vbroker.events.factory	false	true に設定した場合は，イベントチャネルのかわりに，イベントチャネルファクトリをインスタンス化できます。
vbroker.events.debug	false	true に設定した場合は，デバッグ情報を出力できます。
vbroker.events.interactive	false	true に設定した場合は，コンソール駆動の対話モードでイベントチャネルを実行できます。

ネーミングサービス（VisiNaming）のプロパティ

次の表は，VisiNaming サービスのプロパティを一覧です。

表 6.4　VisiNaming サービスのコアプロパティ

プロパティ	デフォルト値	説明
vbroker.naming.adminPwd	inprise	Visibroker ネーミングサービスの管理操作に必要なパスワード。
vbroker.naming.enableSlave	0	1 に設定した場合は，マスター／スレーブのネーミングサービス設定が有効になります。マスター／スレーブのネーミングサービスの設定方法については，203 ページの「VisiNaming サービスクラスタによるフェイルオーバーと負荷分散」を参照してください。
vbroker.naming.iorFile	ns.ior	ネーミングサービス IOR を格納するためのフルパス名を指定します。このプロパティを設定しないと，ネーミングサービスは IOR を現在のディレクトリにある ns.ior という名前のファイルに出力します。ネーミングサービスは，IOR の出力時にファイルアクセス許可の例外を暗黙的に無視します。

表 6.4　VisiNaming サービスのコアプロパティ（続き）

プロパティ	デフォルト値	説明
`vbroker.naming.logLevel`	`emerg`	ネーミングサービスから出力されるログメッセージのレベルを指定します。次の値を指定できます。 • emerg (0)：なんらかの異常な状態を示します。 • alert (1)：ユーザーが注意する必要がある状態です。たとえば，セキュリティが無効になっている場合です。 • crit　(2)：重大な状態です。たとえば，デバイスにエラーが発生した場合です。 • err (3)：エラー状態です。 • warning　(4)：警告状態です。トラブルシューティングの助言も表示される場合があります。 • notice (5)：接続を開く場合など，エラーではないが注意する必要がある状態です。 • info (6)：実行中のバインディングなどの情報を提供します。 • debug (7)：開発者向けのデバッグメッセージです。

`vbroker.naming.logUpdate`	`false`	このプロパティにより，`CosNaming::NamingContext`，`CosNamingExt::Cluster`，および `CosNamingExt::ClusterManager` インターフェースのすべての更新操作をログに記録できます。
		このプロパティが有効な `CosNaming::NamingContext` インターフェースのオペレーションは，次のとおりです。
		`bind`，`bind_context`，`bind_new_context`，`destroy`，`rebind`，`rebind_context`，`unbind`。
		このプロパティが有効な `CosNamingExt::Cluster` インターフェースのオペレーションは，次のとおりです。
		`bind`，`rebind`，`unbind`，`destroy`。
		このプロパティが有効な `CosNamingExt::ClusterManager` インターフェースのオペレーションは，次のとおりです。
		`create_cluster`
		このプロパティの値が `true` に設定されている場合に上のいずれかのメソッドが呼び出されると，次のようなログメッセージが出力されます（この出力は実行中のバインド操作を示します）。
		`00000007,5/26/04 10:11 AM,127.0.0.1,00000000,`
		`VBJ-Application,VBJ ThreadPool Worker,INFO,`
		
		`OPERATION NAME : bind`
		`CLIENT END POINT : Connection[socket=Socket`
		`[addr=/127.0.0.1, port=2026, localport=1993]]`
		`PARAMETER 0 : [(Tom.LoanAccount)]`
		`PARAMETER 1 : Stub[repository_id=IDL:Bank/`
		`LoanAccount:1.0, key=TransientId[poaName=/,`
		`id={4 bytes:`
		`(0)(0)(0)(0)},sec=505,usec=990917734,`
		`key_string=%00VB%01%00%00%00%02/`
		`%00%20%20%00%00%00`
		`04%00%00%00%00%00%00%01%f9;%104f],codebase=null]`

詳細は，「クラスタ」を参照してください。

表 6.5　オブジェクトクラスタ関連のプロパティ

プロパティ	デフォルト値	説明
vbroker.naming.enableClusterFailover	true	true に設定した場合は，VisiNaming サービスから取得されたオブジェクトのフェイルオーバーを処理するインターセプタがインストールされます。オブジェクトに障害が発生した場合は，元のオブジェクトと同じクラスタにある別のオブジェクトに対して透過的な再接続を試みます。
vbroker.naming.propBindOn	0	1 の場合，暗黙的クラスタリング機能が有効になります。
vbroker.naming.smrr.pruneStaleRef	1	このプロパティが関係するのは，ネーミングサービスクラスタが SmartRoundRobin 基準を使用する場合です。このプロパティを 1 に設定すると，ネーミングサービスが SmartRoundRobin 基準を使用してクラスタへ以前にバインドした無効なオブジェクトリファレンスを発見すると，バインディングから除去します。このプロパティを 0 に設定すると，クラスタにおける無効なオブジェクトリファレンスバインディングは削除されません。ただし，SmartRoundRobin 基準を持つクラスタは，resolve() 呼び出しまたは select() 呼び出しでアクティブオブジェクトリファレンスを常に返します。これは，そのようなオブジェクトバインディングが存在するのであれば，vbroker.naming.smrr.pruneStaleRef プロパティの値に関係ありません。デフォルトでは，4.5 ネーミングサービスでの暗黙的なクラスタリングは，このプロパティ値が 1 に設定された SmartRoundRobin 基準を使用します。このプロパティを 2 に設定すると，無効なリファレンスが削除されなくなり，バインディングのクリーンアップは VisiNaming ではなくアプリケーションが行うことになります。

詳細は，203 ページの「VisiNaming サービスクラスタによるフェイルオーバーと負荷分散」を参照してください。

表 6.6　VisiNaming サービスクラスタ関連のプロパティ

プロパティ	デフォルト値	説明
vbroker.naming.enableSlave	0	「VisiNaming サービスのコアプロパティ」を参照してください。
vbroker.naming.slaveMode	デフォルトなし。cluster または slave に設定できます。	このプロパティを使用して，クラスタモードまたはマスター／スレーブモードの VisiNaming サービスインスタンスを設定します。このプロパティを有効にするには，vbroker.naming.enableSlave プロパティを 1 に設定する必要があります。 クラスタモードの VisiNaming サービスインスタンスを設定するには，このプロパティを cluster に設定します。これで，クラスタを構成する VisiNaming サービスインスタンス間で VisiNaming サービスクライアントが負荷分散されます。これらのインスタンス間でのクライアントのフェイルオーバーが有効になります。 マスター／スレーブモードの VisiNaming サービスインスタンスを設定するには，このプロパティを slave に設定します。VisiNaming サービスクライアントは，マスターの実行中は常にマスターサーバーにバインドされますが，マスターサーバーがダウンした場合はスレーブサーバーにフェイルオーバーします。

表 6.6　VisiNaming サービスクラスタ関連のプロパティ（続き）

プロパティ	デフォルト値	説明
vbroker.naming.serverClusterName	null	このプロパティは，VisiNaming サービスクラスタの名前を指定します。このプロパティを使用して，複数の VisiNaming サービスインスタンスにクラスタ名（たとえば，clusterXYZ）を設定した場合，それらのインスタンスは特定のクラスタに属します。
vbroker.naming.serverNames	null	このプロパティは，クラスタに属している VisiNaming サービスインスタンスのファクトリ名を指定します。クラスタ内の各 VisiNaming サービスインスタンスは，このプロパティを使用して，クラスタを構成するすべてのインスタンスを認識するように設定されます。リスト内の名前は一意である必要があります。このプロパティは次の形式をサポートします。 vbroker.naming.serverNames= Server1:Server2:Server3 関連のプロパティ vbroker.naming.serverAddresses を参照してください。
vbroker.naming.serverAddresses	null	このプロパティは，ホストおよび VisiNaming サービスクラスタを構成する VisiNaming サービスインスタンスの監視ポートを指定します。このリスト内の VisiNaming サービスインスタンスの順番は，関連プロパティ vbroker.naming.serverNames（VisiNaming サービスクラスタを構成する VisiNaming サービスインスタンスの名前を指定する）の順番と同じである必要があります。このプロパティは次の形式をサポートします。 vbroker.naming.serverAddresses=host1: port1;host2:port2;host3:port3
vbroker.naming.anyServiceOrder (To be set on VisiNaming Service clients)	false	VisiNaming サービスインスタンスが VisiNaming サービスクラスタモードで設定されている場合に負荷分散機能とフェイルオーバー機能を使用するには，VisiNaming サービスクライアントでこのプロパティを true に設定する必要があります。このプロパティの使い方の例を次に示します。 client - DVbroker.naming.anyServiceOrder=true

取り替え可能なバックストアプロパティ

次の表は，VisiNaming サービスの取り替え可能なバックストアのタイプに対するプロパティ情報を示します。

すべてのアダプタに共通するデフォルトのプロパティ

プロパティ	デフォルト値	説明
vbroker.naming.backingStoreType	InMemory	使用するネーミングサービスアダプタのタイプを指定します。このプロパティは，VisiNaming サービスで使用するバックストアのタイプを指定します。有効なオプションは，InMemory，JDBC，Dx，JNDI です。デフォルトは InMemory です。
vbroker.naming.cacheOn	0	ネーミングサービスキャッシュを使用するかどうかを指定します。値を 1 にすると，キャッシュが有効になります。
vbroker.naming.cache.connectString	N/A	ネーミングサービスキャッシュが有効で（vbroker.naming.cacheOn=1），ネーミングサービスインスタンスがクラスタモードまたはマスター／スレーブモードで設定されている場合は，このプロパティが必要です。これにより，イベントサービスのインスタンスを <hostname>:<port> の形式で検索できます。たとえば，次のようになります。 vbroker.naming.cache.connectString=127.0.0.1:14500 キャッシング機能を有効にし，適切なプロパティを設定する方法の詳細は，197 ページの「キャッシング機能」を参照してください。
vbroker.naming.cache.size	2000	このプロパティは，ネーミングサービスキャッシュのサイズを指定します。値を大きくすると，より多くのデータをキャッシュできますが，メモリの消費量が増大します。
vbroker.naming.cache.timeout	0（制限なし）	このプロパティには，前回アクセスされてからデータを保持する時間（秒）を指定します。この時間が経過すると，キャッシュのデータはメモリを解放するために破棄されます。キャッシュに保持されるエントリは，LRU（使用頻度が低い）順に削除されます。

JDBC アダプタのプロパティ

プロパティ	デフォルト値	説明
vbroker.naming.jdbcDriver	com.borland.datastore.jdbc.DataStoreDriver	このプロパティでは，バックストアとして使用するデータベースへアクセスするために必要な JDBC ドライバを指定します。VisiNaming サービスは，指定された適切な JDBC ドライバをロードします。有効な値は次のとおりです。 • com.borland.datastore.jdbc.DataStoreDriver—JDataStore ドライバ。 • com.sybase.jdbc.SybDriver—Sybase ドライバ。 • oracle.jdbc.driver.OracleDriver—Oracle ドライバ。 • interbase.interclient.Driver—Interbase ドライバ。 • weblogic.jdbc.mssqlserver4.Driver—WebLogic MS SQLServer ドライバ。 • COM.ibm.db2.jdbc.app.DB2Driver—IBM DB2 ドライバ。
vbroker.naming.loginName	VisiNaming	データベースに関連付けられているログイン名を指定します。

プロパティ	デフォルト値	説明
vbroker.naming.loginPwd	VisiNaming	データベースに関連付けられているログインパスワードを指定します。
vbroker.naming.poolSize	5	バックストアとして JDBC アダプタを使用する場合は，このプロパティで接続プールのデータベース接続数を指定します。
vbroker.naming.url	jdbc:borland:dslocal:rootDB.jds	ネーミングサービスがアクセスするデータベースの場所を指定します。設定内容は，使用するデータベースによって異なります。次の値を指定できます。 • jdbc:borland:dslocal:<db-name>—JDataStore UTL • jdbc:sybase:Tds:<host-name>:<port-number>/<db-name>—Sybase URL • jdbc:oracle:thin@<host-name>:<port-number>:<sid>—Oracle URL • jdbc:interbase://<server-name>/<full-db-path>—Interbase URL • jdbc:weblogic:mssqlserver4:<db-name>@<host-name>:<port-number>—WebLogic MS SQLSever URL • jdbc:db2:<db-name>—IBM DB2 URL • <full-path-JDataStore-db>—DataExpress3 URL（ネイティブドライバの場合）
vbroker.naming.minReconInterval	30	このプロパティは，ネーミングサービスのデータベース再接続間隔を秒単位で設定します。デフォルト値は，30 です。この要求と最後の接続時刻の間の時間が vset 値未満の場合，ネーミングサービスは再接続要求を無視し，CannotProceed 例外を生成します。このプロパティの有効値は 0 以上の整数です。0 に設定された場合，ネーミングサービスは，要求があるたびにデータベースに再接続しようとします。

DataExpress アダプタのプロパティ

次に，DataExpress アダプタのプロパティについて説明します。

プロパティ	説明
vbroker.naming.backingStoreType	このプロパティは，Dx に設定します。
vbroker.naming.loginName	このプロパティは，データベースに関連付けられているログイン名です。デフォルトは VisiNaming です。
vbroker.naming.loginPwd	このプロパティは，データベースに関連付けられていログイン用のパスワードです。デフォルト値は，VisiNaming です。
vbroker.naming.url	このプロパティは，データベースの場所を指定します。

JNDI アダプタのプロパティ

次に示すのは，JNDI アダプタの設定ファイルで指定できる設定のサンプルです。

設定値	説明
vbroker.naming.backingStoreType=JNDI	バックストアのタイプを指定します。JNDI アダプタの場合は JNDI です。
vbroker.naming.loginName=<user_name>	JNDI バックサーバーでのユーザーログイン名です。
vbroker.naming.loginPwd=<password>	JNDI バックサーバーユーザーのパスワードです。
vbroker.naming.jndiInitialFactory=com.sun.jndi.ldap.LdapCtxFactory	JNDI 初期ファクトリを指定します。
vbroker.naming.jndiProviderURL=ldap://<hostname>:389/<initial root context>	JNDI プロバイダの URL を指定します。
vbroker.naming.jndiAuthentication=simple	JNDI バックサーバーがサポートしている JNDI 認証のタイプを指定します。

VisiNaming サービスのセキュリティ関連プロパティ

プロパティ	値	デフォルト値	説明
vbroker.naming.security.disable	boolean	true	セキュリティサービスが無効かどうかを指定します。
vbroker.naming.security.authDomain	string	""	ネーミングサービスメソッドのアクセス承認に使用される承認ドメイン名を指定します。
vbroker.naming.security.transport	int	3	このプロパティは，ネーミングサービスが使用するトランスポートを示します。次の値がサポートされています。 ServerQoPPolicy.SECURE_ONLY=1 ServerQoPPolicy.CLEAR_ONLY=0 ServerQoPPolicy.ALL=3
vbroker.naming.security.requireAuthentication	boolean	false	ネーミングクライアント認証が必要かどうかを指定します。ただし，vbroker.naming.security.disable プロパティが true に設定されている場合は，この requireAuthentication プロパティの値に関係なく，クライアント認証は実行されません。
vbroker.naming.security.enableAuthorization	boolean	false	メソッドアクセス承認が有効かどうかを指定します。
vbroker.naming.security.requiredRolesFile	string	*null*	プロテクトオブジェクト型の各メソッドの起動に必要な役割を含むファイルをポイントします。詳細は，209 ページの「メソッドレベル承認」を参照してください。

OAD のプロパティ

次の表は，設定可能 OAD のプロパティの一覧です。

プロパティ	デフォルト値	説明
vbroker.oad.spawnTimeOut	20	OAD が実行可能プログラムを生成した後，システムが目的のオブジェクトからのコールバックを待機して，NO_RESPONSE 例外を生成するまで時間を秒単位で指定します。
vbroker.oad.verbose	false	OAD がオペレーションに関する詳細情報を出力します。
vbroker.oad.readOnly	false	true に設定した場合は，OAD インプリメンテーションを登録，登録解除，または変更できません。
vbroker.oad.iorFile	Oadj.ior	OAD の文字列化された IOR を記したファイルの名前を指定します。
vbroker.oad.quoteSpaces	false	コマンドを引用するかどうかを指定します。
vbroker.oad.killOnUnregister	false	登録解除された子のサーバープロセスを終了するかどうかを指定します。
vbroker.oad.verifyRegistration	false	オブジェクトの登録を確認するかどうかを指定します。

次の表は，プロパティファイルで上書きできない OAD のプロパティの一覧です。ただし，環境変数またはコマンドラインを使用すると，これらのプロパティを上書きできます。

プロパティ	デフォルト値	説明
vbroker.oad.implName	impl_rep	インプリメンテーションリポジトリのファイル名を指定します。
vbroker.oad.implPath	null	インプリメンテーションリポジトリが保存されているディレクトリを指定します。
vbroker.oad.path	null	OAD のディレクトリを指定します。
vbroker.oad.systemRoot	null	ルートディレクトリを指定します。
vbroker.oad.windir	null	Windows ディレクトリを指定します。

インターフェースリポジトリのプロパティ

次の表は，インターフェースリポジトリ（IR）のプロパティの一覧です。

プロパティ	デフォルト値	説明
vbroker.ir.debug	false	true に設定した場合は, IR リゾルバがデバッグ情報を表示できます。
vbroker.ir.ior	null	vbroker.ir.name プロパティがデフォルト値 null に設定された場合，VisiBroker ORB はこのプロパティを使用して IR を検索します。
vbroker.ir.name	null	VisiBroker ORB が IR の検索に使用する名前を指定します。

TypeCode のプロパティ

次の表は，VisiBroker for C++ の TypeCode のプロパティの一覧です。

表 6.7　TypeCode のプロパティ

プロパティ	デフォルト値	説明
vbroker.typecode.debug	FALSE	TRUE に設定した場合は，タイプコードがデバッグを表示します。
vbroker.typecode.noIndirection	FALSE	TRUE に設定した場合，このプロパティは，再帰的なタイプコードを作成する場合にインダイレクションの使用を許容しません。

クライアント側 LIOP 接続のプロパティ

次の表は，VisiBroker for C++ のクライアント側 LIOP 接続のプロパティの一覧です。

表 6.8　クライアント側 LIOP 接続のプロパティ

プロパティ	デフォルト値	説明
vbroker.ce.liop.ccm.connectionCacheMax	5	クライアントごとのキャッシュ接続の最大数を指定します。クライアントが接続を解放すると，その接続がキャッシュされます。そのために，その後新しい接続が必要になると，クライアントは，実際に新しい接続を作成するかわりに，キャッシュから使用可能な接続を取得できます。
vbroker.ce.liop.ccm.disableConnectionCache	false	true に設定した場合，このプロパティは，クライアント側の接続キャッシュを無効にします。
vbroker.ce.liop.ccm.connectionMax	0	クライアントごとの接続総数の最大数を指定します。これには，アクティブな接続に加えて，キャッシュされている接続が含まれます。デフォルト値の 0 の場合，クライアントは，既存のアクティブな接続とキャッシュされている接続のどちらも閉じようとしません。
vbroker.ce.liop.ccm.connectionMaxIdle	360	キャッシュされている接続を閉じるかどうかをクライアントが判定するための基準の時間（秒単位）を指定します。つまり，接続がキャッシュされてからこの時間以上アイドルになると，クライアントはその接続を閉じます。
vbroker.ce.liop.ccm.type	Pool	クライアントが使用する接続管理のタイプを指定します。デフォルト値は，Pool（接続プール）です。現在，このプロパティの有効な値はこれだけです。
vbroker.ce.liop.connection.rcvBufSize	0	受信ソケットバッファのサイズを指定します。デフォルト値 0 の場合は，システム依存の値が使用されます。

表 6.8　クライアント側 LIOP 接続のプロパティ（続き）

プロパティ	デフォルト値	説明
vbroker.ce.liop.connection.sendBufSize	0	送信ソケットバッファのサイズを指定します。デフォルト値 0 の場合は，システム依存の値が使用されます。
vbroker.ce.liop.connection.shmSize	4096	共有メモリのサイズ（バイト単位）を指定します。クライアントプログラムとオブジェクトインプリメンテーションが共有メモリを介して通信する場合は，このオプションを調整するとパフォーマンスが向上することがあります。
vbroker.se.default.local.listener.doorMaxMsgSize	1,000,000	Solaris の高速 IPC（door）メカニズムを使用して送信されるメッセージの最大サイズを指定します（クライアントとサーバーが同じマシンで動作している場合）。メッセージサイズがデフォルト値（1,000,000）より大きい場合は，IPC を使用して送信されず，次に使用できるメカニズム（UNIX ドメインソケットまたは TCP/IP ソケット）がデフォルトとして使用されます。

クライアント側 IIOP 接続のプロパティ

次の表は，VisiBroker for C++ クライアント側 IIOP 接続のプロパティの一覧です。

表 6.9　クライアント側 IIOP 接続のプロパティ

プロパティ	デフォルト値	説明
vbroker.ce.iiop.ccm.connectionCacheMax	5	クライアントごとのキャッシュ接続の最大数を指定します。クライアントが接続を解放すると，その接続がキャッシュされます。そのために，その後新しい接続が必要になると，クライアントは，単に実際に新しい接続を作成するかわりに，まずキャッシュから使用可能な接続を取得しようとします。
vbroker.ce.iiop.ccm.disableConnectionCache	false	このプロパティを true に設定すると，クライアント側の接続キャッシュが無効になります。
vbroker.ce.iiop.ccm.connectionMax	0	クライアントごとの接続総数の最大数を指定します。これは，アクティブな接続の数とキャッシュされた接続の数に等しくなります。デフォルト値の 0 の場合，クライアントは，既存のアクティブな接続とキャッシュされている接続のどちらも閉じようとしません。
vbroker.ce.iiop.ccm.connectionMaxIdle	0	キャッシュされている接続を閉じるかどうかをクライアントが判定するための基準の時間（秒単位）を指定します。つまり，接続がキャッシュされてからこの時間以上アイドルになると，クライアントはその接続を閉じます。

表 6.9　クライアント側 IIOP 接続のプロパティ（続き）

プロパティ	デフォルト値	説明
vbroker.ce.iiop.ccm.type	Pool	クライアントが使用する接続管理のタイプを指定します。Pool 値は，接続プールを意味します。現在，このプロパティの有効な値はこれだけです。
vbroker.ce.iiop.connection.rcvBufSize	0	受信ソケットバッファのサイズを指定します。デフォルト値 0 の場合は，システム依存の値が使用されます。
vbroker.ce.iiop.connection.sendBufSize	0	送信ソケットバッファのサイズを指定します。デフォルト値 0 の場合は，システム依存の値が使用されます。
vbroker.ce.iiop.connection.tcpNoDelay	FALSE	TRUE に設定した場合，サーバーのソケットは，バッファがいっぱいになるたびにデータを一括して送信するのではなく，書き込まれたデータをただちに送信するように設定されます。
vbroker.ce.iiop.host	なし	クライアント側のソケットを目的のインターフェースにバインドします。値が null の場合は，ワイルドカードインターフェースが使用されます。
vbroker.ce.iiop.connection.noCallback	FALSE	TRUE に設定した場合，サーバーがクライアントにコールバックできます。
vbroker.ce.iiop.connection.socketLinger	0	TCP/IP 設定。
vbroker.ce.iiop.connection.keepAlive	TRUE	TCP/IP 設定。

QoS 関連のプロパティ

プロパティ	デフォルト値	説明
vbroker.qos.cache	True	クライアントが要求を行うたびに QoS ポリシーをチェックするのではなく，デリゲートごとに QoS ポリシーをキャッシュするかどうかを指定します。

クライアント側インプロセス接続のプロパティ

次の表は，クライアント側インプロセス接続のプロパティの一覧です。

プロパティ	デフォルト値	説明
vbroker.ce.inprocess.ccm.bid	9488	POA bidder の bid 値を指定します。この値は，クライアント接続を処理するプロトコルを選択するとき，VisiBroker ORB で使用される自動処理に影響します。
vbroker.ce.iiop.ccm.bid	10000	iiop bidder の bid 値を指定します。この値は，クライアント接続を処理するプロトコルを選択するとき，VisiBroker ORB で使用される自動処理に影響します。

サーバー側サーバーエンジンのプロパティ

次の表は，サーバー側サーバーエンジンのプロパティの一覧です。

プロパティ	デフォルト値	説明
vbroker.se.default	iiop_tp	デフォルトのサーバーエンジンを指定します。

サーバー側スレッドセッション IIOP_TS/IIOP_TS 接続のプロパティ

次の表は，サーバー側スレッドセッション IIOP_TS/IIOP_TS 接続のプロパティの一覧です。

プロパティ	デフォルト値	説明
vbroker.se.iiop_ts.host	null	このサーバーエンジンによって使用されるホスト名を指定します。デフォルト値の null の場合は，システムからホスト名が取得されます。
vbroker.se.iiop_ts.proxyHost	null	このサーバーエンジンによって使用されるプロキシホスト名を指定します。デフォルト値の null の場合は，システムからホスト名が取得されます。
vbroker.se.iiop_ts.scms	iiop_ts	サーバー接続マネージャの名前のリストを指定します。
vbroker.se.iiop_ts.scm.iiop_ts.manager.type	Socket	サーバー接続マネージャのタイプを指定します。
vbroker.se.iiop_ts.scm.iiop_ts.manager.connectionMax	0	サーバーが受け付ける接続の最大数を指定します。デフォルト値の 0 は，無制限を指定します。
vbroker.se.iiop_ts.scm.iiop_ts.manager.connectionMaxIdle	0	アクティブでない接続を閉じるかどうかをサーバーが判定するための基準の時間を秒単位で指定します。
vbroker.se.iiop_ts.scm.iiop_ts.manager.garbageCollectTimer	30	接続オブジェクトのガベージコレクションの間隔の秒数。
vbroker.se.iiop_ts.scm.iiop_ts.listener.type	IIOP	リスナーが使用するプロトコルを指定します。
vbroker.se.iiop_ts.scm.iiop_ts.listener.port	0	ホスト名のプロパティとともに使用されるポート番号を定義します。デフォルト値は 0 で，その場合は，システムがポート番号をランダムに選択します。
vbroker.se.iiop_ts.scm.iiop_ts.listener.proxyPort	0	プロキシホスト名のプロパティとともに使用されるプロキシポート番号を定義します。デフォルト値は 0 で，その場合は，システムがポート番号をランダムに選択します。
vbroker.se.iiop_ts.scm.iiop_ts.listener.rcvBufSize	0	受信ソケットバッファのサイズを指定します。デフォルト値 0 の場合は，システム依存の値が使用されます。
vbroker.se.iiop_ts.scm.iiop_ts.listener.sendBufSize	0	送信バッファのサイズを指定します。デフォルト値 0 の場合は，システム依存の値が使用されます。
vbroker.se.iiop_ts.scm.iiop_ts.listener.socketLinger	0	TCP/IP 設定。
vbroker.se.iiop_ts.scm.iiop_ts.listener.keepAlive	true	TCP/IP 設定。

プロパティ	デフォルト値	説明
vbroker.se.iiop_ts.scm.iiop_ts.listener.giopVersion	1.2	このプロパティは，未知の GIOP のマイナーバージョンを正しく処理できない古い VisiBroker ORB で発生する相互運用性の問題を解決するために使用されます。このプロパティの正しい値は，1.0，1.1，および 1.2 です。たとえば，GIOP 1.1 ior を生成するには，次のようにネーミングサービスを起動します。 `nameserv -VBJprop vbroker.se.iiop_tp.scm.iiop_tp.listener.giopVersion=1.1`
vbroker.se.iiop_ts.scm.iiop_ts.dispatcher.type	"ThreadSession"	サーバー接続マネージャで使用されるスレッドディスパッチャのタイプを指定します。
vbroker.se.iiop_ts.scm.iiop_ts.dispatcher.threadStackSize	0	スレッドスタックのサイズ デフォルト値は 0 で，システムのデフォルトになります。ただし，HP-UX プラットフォームでは，デフォルト値は 128 KB です。
vbroker.se.iiop_ts.scm.iiop_ts.dispatcher.coolingTime	3	接続がホットである（さらに多くの要求が予測される）とみなされる時間（秒単位）。この時間が経過すると，接続はディスパッチャから戻されます。
vbroker.se.iiop_ts.scm.iiop_ts.connection.rcvBufSize	0	受信ソケットバッファのサイズを指定します。デフォルト値 0 の場合は，システム依存の値が使用されます。
vbroker.se.iiop_ts.scm.iiop_ts.connection.sendBufSize	0	送信バッファのサイズを指定します。デフォルト値 0 の場合は，システム依存の値が使用されます。
vbroker.se.iiop_ts.scm.iiop_ts.connection.socketLinger	0	TCP/IP 設定。
vbroker.se.iiop_ts.scm.iiop_ts.connection.keepAlive	true	TCP/IP 設定。
vbroker.se.iiop_ts.scm.iiop_ts.connection.tcpNoDelay	true	このプロパティを false に設定した場合は，ソケットのバッファリングがオンになります。true に設定した場合は，ソケットのバッファリングをオフにして準備ができしだい，すべてのパケットを送信します。

サーバー側スレッドセッション BOA_TS/BOA_TS 接続のプロパティ

このプロトコルは，スレッドセッション iiop_ts/iiop_ts 接続プロパティと同じプロパティを持ちます。ただし，各プロパティの iiop_ts が boa_ts にかわります。たとえば，vbroker.se.iiop_ts.scm.iiop_ts.manager.connectionMax は vbroker.se.boa_ts.scm.boa_ts.manager.connectionMax になります。また，vbroker.se.boa_ts.scms のデフォルト値は boa_ts です。

サーバー側スレッドプール IIOP_TP/IIOP_TP 接続のプロパティ

次の表は，サーバー側スレッドプール IIOP_TP/IIOP_TP 接続のプロパティの一覧です。

プロパティ	デフォルト値	説明
vbroker.se.iiop_tp.host	null	このサーバーエンジンによって使用されるホスト名を指定します。デフォルト値の null の場合は，システムからホスト名が取得されます。ホスト名または IP アドレスが受け入れ可能な値です。
vbroker.se.iiop_tp.proxyHost	null	このサーバーエンジンによって使用されるプロキシホスト名を指定します。デフォルト値の null の場合は，システムからホスト名が取得されます。ホスト名または IP アドレスが受け入れ可能な値です。

プロパティ	デフォルト値	説明
vbroker.se.iiop_tp.scms	iiop_tp	サーバー接続マネージャの名前のリストを指定します。
vbroker.se.iiop_tp.scm.iiop_tp.manager.type	Socket	サーバー接続マネージャのタイプを指定します。
vbroker.se.iiop_tp.scm.iiop_tp.manager.connectionMax	0	サーバー上のキャッシュ接続の最大数を指定します。デフォルト値の 0 は，無制限を指定します。
vbroker.se.iiop_tp.scm.iiop_tp.manager.connectionMaxIdle	0	アクティブでない接続を閉じるかどうかをサーバーが判定するための基準の時間を秒単位で指定します。
vbroker.se.iiop_tp.scm.iiop_tp.manager.garbageCollectTimer	30	接続に対するガベージコレクションタイマーを秒単位で指定します。
vbroker.se.iiop_tp.scm.iiop_tp.listener.type	IIOP	リスナーが使用するプロトコルを指定します。
vbroker.se.iiop_tp.scm.iiop_tp.listener.port	0	ホスト名のプロパティで使用するポート番号を定義します。デフォルト値は 0 で，その場合は，システムがポート番号をランダムに選択します。
vbroker.se.iiop_tp.scm.iiop_tp.listener.proxyPort	0	プロキシホスト名のプロパティとともに使用されるプロキシポート番号を定義します。デフォルト値は 0 で，その場合は，システムがポート番号をランダムに選択します。
vbroker.se.iiop_tp.scm.iiop_tp.listener.rcvBufSize	0	受信ソケットバッファのサイズを指定します。デフォルト値 0 の場合は，システム依存の値が使用されます。
vbroker.se.iiop_tp.scm.iiop_tp.listener.sendBufSize	0	送信バッファのサイズを指定します。デフォルト値 0 の場合は，システム依存の値が使用されます。
vbroker.se.iiop_tp.scm.iiop_tp.listener.socketLinger	0	TCP/IP 設定。
vbroker.se.iiop_tp.scm.iiop_tp.listener.keepAlive	true	TCP/IP 設定。
vbroker.se.iiop_tp.scm.iiop_tp.listener.giopVersion	1.2	このプロパティは，未知の GIOP のマイナーバージョンを正しく処理できない古い VisiBroker ORB で発生する相互運用性の問題を解決するために使用されます。このプロパティに設定できる値は，1.0，1.1，および 1.2 です。
vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.type	ThreadPool	サーバー接続マネージャで使用されるスレッドディスパッチャのタイプを指定します。
vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.threadMin	0	サーバー接続マネージャが作成できるスレッドの最小数を指定します。
vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.threadMax	0	サーバー接続マネージャが作成できるスレッドの最大数を指定します。デフォルト値の 0 の場合は，ORB がヒューリスティックに基づく内部アルゴリズムを使用してスレッド生成を制御します。
vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.threadMaxIdle	300	アイドルなスレッドを破棄するまでの時間を秒単位で指定します。
vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.threadStackSize	0	スレッドスタックのサイズ デフォルト値は 0 で，システムのデフォルトになります。ただし，HP-UX プラットフォームでは，デフォルト値は 128 KB です。
vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.coolingTime	3	接続がホットである（さらに多くの要求が予測される）とみなされる時間（秒単位）。この時間が経過すると，接続はディスパッチャから戻されます。
vbroker.se.iiop_tp.scm.iiop_tp.connection.rcvBufSize	0	受信ソケットバッファのサイズを指定します。デフォルト値 0 の場合は，システム依存の値が使用されます。

プロパティ	デフォルト値	説明
vbroker.se.iiop_tp.scm.iiop_tp.connection.sendBufSize	0	送信バッファのサイズを指定します。デフォルト値 0 の場合は，システム依存の値が使用されます。
vbroker.se.iiop_tp.scm.iiop_tp.connection.socketLinger	0	TCP/IP 設定。
vbroker.se.iiop_tp.scm.iiop_tp.connection.keepAlive	true	TCP/IP 設定。
vbroker.se.iiop_tp.scm.iiop_tp.connection.tcpNoDelay	true	このプロパティを false に設定した場合は，ソケットのバッファリングがオンになります。デフォルト値の true に設定した場合は，ソケットのバッファリングをオフにして準備ができしだい，すべてのパケットを送信します。

サーバー側スレッドプール BOA_TP/BOA_TP 接続のプロパティ

このプロトコルは，スレッドプール iiop_tp/iiop_tp 接続プロパティと同じプロパティを持ちます。ただし，各プロパティの iiop_tp が boa_tp にかわります。たとえば，vbroker.se.iiop_tp.scm.iiop_tp.manager.connectionMax は vbroker.se.boa_tp.scm.boa_tp.manager.connectionMax になります。また，vbroker.se.boa_tp.scms のデフォルト値は boa_tp です。

サーバー側スレッドプール LIOP_TP/LIOP_TP 接続のプロパティ

次の表は，サーバー側スレッドプール LIOP_TP/LIOP_TP 接続のプロパティの一覧です。

プロパティ	デフォルト値	説明
vbroker.se.liop_tp.host	null	このサーバーエンジンによって使用されるホスト名を指定します。デフォルト値の null の場合は，システムからホスト名が取得されます。ホスト名または IP アドレスが受け入れ可能な値です。
vbroker.se.liop_tp.proxyHost	null	このサーバーエンジンによって使用されるプロキシホスト名を指定します。デフォルト値の null の場合は，システムからホスト名が取得されます。ホスト名または IP アドレスが受け入れ可能な値です。
vbroker.se.liop_tp.scms	liop_tp	サーバー接続マネージャの名前のリストを指定します。
vbroker.se.liop_tp.scm.liop_tp.manager.type	Local	サーバー接続マネージャのタイプを指定します。
vbroker.se.liop_tp.scm.liop_tp.manager.connectionMax	0	サーバー上のキャッシュ接続の最大数を指定します。デフォルト値の 0 は無制限を指定します。
vbroker.se.liop_tp.scm.liop_tp.manager.connectionMaxIdle	0	アクティブでない接続を閉じるかどうかをサーバーが判定するための基準の時間を秒単位で指定します。
vbroker.se.liop_tp.scm.liop_tp.manager.garbageCollectTimer	30	接続に対するガベージコレクションタイマーを秒単位で指定します。
vbroker.se.liop_tp.scm.liop_tp.listener.type	LIOP	リスナーが使用するプロトコルを指定します。
vbroker.se.liop_tp.scm.liop_tp.listener.port	0	ホスト名のプロパティで使用するポート番号を定義します。デフォルト値は 0 で，その場合は，システムがポート番号をランダムに選択します。
vbroker.se.liop_tp.scm.liop_tp.listener.proxyPort	0	プロキシホスト名のプロパティとともに使用されるプロキシポート番号を定義します。デフォルト値は 0 で，その場合は，システムがポート番号をランダムに選択します。

プロパティ	デフォルト値	説明
vbroker.se.default.local.listener.door	true	同じマシンで動作しているクライアントとサーバーの通信に，Door API を使用するかどうかを指定します。true に設定すると，LIOP に Door API が使用されます。false に設定すると，LIOP は，IPC の UNIX ドメインソケットを使用します。このプロパティは，Solaris オペレーティングシステム専用です。
vbroker.se.xxx.scm.yyy.listener.shmSize	4096	共有メモリ割り当てのサイズ。単位はバイト数。クライアントプログラムとオブジェクトインプリメンテーションが共有メモリを介して通信する場合は，このオプションを調整するとパフォーマンスが向上することがあります。
vbroker.se.xxx.scm.yyy.listener.userConstrained	0	true に設定した場合，ファイルは，そのディレクトリの所有者しかアクセスできないディレクトリに隠されます。
vbroker.se.liop_tp.scm.liop_tp.listener.giopVersion	1.2	このプロパティは，未知の GIOP のマイナーバージョンを正しく処理できない古い VisiBroker ORB で発生する相互運用性の問題を解決するために使用されます。このプロパティに設定できる値は，1.0，1.1，および 1.2 です。
vbroker.se.liop_tp.scm.liop_tp.listener.allowedGroups	null	サーバーアプリケーションが，Windows 上のローカル IPC 通信で使用される有効な同期オブジェクトの管理元を制御できるようにします。セミコロンで区切られた Windows ユーザーグループが LIOP を使用してサーバーにアクセスできるようにします。指定されたグループに属していないユーザーは，接続が許可されず，IIOP にフェイルオーバーします。
vbroker.se.liop_tp.scm.liop_tp.dispatcher.type	ThreadPool	サーバー接続マネージャで使用されるスレッドディスパッチャのタイプを指定します。
vbroker.se.liop_tp.scm.liop_tp.dispatcher.threadMin	0	サーバー接続マネージャが作成できるスレッドの最小数を指定します。
vbroker.se.liop_tp.scm.liop_tp.dispatcher.threadMax	0	サーバー接続マネージャが作成できるスレッドの最大数を指定します。デフォルト値の 0 の場合は，ORB がヒューリスティックに基づく内部アルゴリズムを使用してスレッド生成を制御します。
vbroker.se.liop_tp.scm.liop_tp.dispatcher.threadMaxIdle	300	アイドルなスレッドを破棄するまでの時間を秒単位で指定します。
vbroker.se.liop_ts.scm.liop_ts.dispatcher.threadStackSize	0	スレッドスタックのサイズ デフォルト値は 0 で，システムのデフォルトになります。ただし，HP-UX プラットフォームでは，デフォルト値は 128 KB です。
vbroker.se.liop_tp.scm.liop_tp.dispatcher.coolingTime	3	接続がホットである（さらに多くの要求が予測される）とみなされる時間（秒単位）。この時間が経過すると，接続はディスパッチャから戻されます。

サーバー側スレッドプール BOA_LTP/BOA_LTP 接続のプロパティ

このプロトコルは，スレッドプール liop_tp/liop_tp 接続プロパティと同じプロパティを持ちます。ただし，各プロパティの liop_tp が boa_ltp にかわります。たとえば，vbroker.se.liop_tp.scm.liop_tp.manager.connectionMax は vbroker.se.boa_ltp.scm.boa_ltp.manager.connectionMax になります。また，vbroker.se.boa_ltp.scms のデフォルト値は boa_ltp です。

双方向通信をサポートするプロパティ

次の表は，双方向通信をサポートするプロパティの一覧です。これらのプロパティは，SCM の作成時に一度だけ評価されます。どの場合も，SCM の exportBiDir プロパティと importBiDir プロパティは，enableBiDir プロパティより優先します。両方のプロパティに相反する値を設定すると，SCM 固有のプロパティが適用されます。このため，enableBiDir プロパティをグローバルに設定して，各 SCM では選択的に双方向通信をオフにすることができます。

プロパティ	デフォルト値	説明
vbroker.orb.enableBiDir	なし	双方向接続を選択的に構築できます。クライアントが vbroker.orb.enableBiDir=client と定義し，サーバーが vbroker.orb.enableBiDir=server と定義している場合は，GateKeeper の vbroker.orb.enableBiDir の値によって接続状態が決定されます。このプロパティの有効値は server，client，both，または none です。
vbroker.se.<se>.scm.<scm>.manager.exportBiDir	デフォルトでは，ORB はこのプロパティに値を設定しません。	これはクライアント側のプロパティです。このプロパティを true に設定すると，指定したサーバーエンジンで双方向コールバック POA を作成できるようになります。このプロパティを false に設定すると，指定したサーバーエンジンで双方向 POA を作成できなくなります。
vbroker.se.<se>.scm.<scm>.manager.importBiDir	デフォルトでは，ORB は値を設定しません。	これはサーバー側のプロパティです。このプロパティを true に設定すると，サーバー側はすでにクライアントによって確立されている接続を再利用して，クライアントに要求を送信できます。このプロパティを false に設定すると，接続の再利用は無効になります。

デバッグログのプロパティ

ここでは，デバッグログステートメントの出力を制御および設定するために使用できるプロパティの詳細について説明します。

デバッグログステートメントは，記録された ORB の領域にしたがってカテゴリに分類されます。このカテゴリをソース名と言います。現在，次のソース名が記録されます。

- connection：クライアント側接続，サーバー側接続，接続プールなどの接続関連ソース領域のログ
- client - クライアント側呼び出しパスのログ
- agent - Osagent 通信のログ
- cdr - GIOP 領域のログ
- se - ディスパッチャ，リスナーなどのサーバーエンジンのログ

- server - サーバー側呼び出しパスのログ
- orb - ORB のログ

次の表では，ログとフィルタリングを有効にするために使用されるプロパティについて説明します。

プロパティ	デフォルト値	説明
vbroker.log.enable	false	true に設定すると，フィルタリングされない限り，すべてのログステートメントが作成されます。 指定できる値は，true または false です。
vbroker.log.logLevel	debug	ログメッセージのログレベルを指定します。レベルを設定すると，指定したレベル以上のログレベルを持つログが転送されます。このプロパティはグローバルに適用されます。 指定できる値は，レベルが高い順に，emerg，alert，crit，err，warning，notice，info，debug です。 次に，各ログレベルの意味を示します。 • emerg - なんらかの異常な状態を示します。 • alert - ユーザーが注意する必要がある状態です。たとえば，セキュリティが無効になっている場合です。 • crit - 重大な状態です。たとえば，デバイスにエラーが発生した場合です。 • err - エラー状態です。 • warning - 警告状態です。トラブルシューティングの助言も表示される場合があります。接続が開かれるときなどに表示されます。 • info - 実行中のバインディングなどの情報を提供します。 • debug - 開発者によって使用されるデバッグ状態です。
vbroker.log.default.filter.register	null	ログを制御（フィルタリング）する記録元のソースのソース名を登録します。 指定できる値は，client，server，connection，cdr，se，agent，orb です。コンマ区切りの文字列で，複数の値を指定できます。 vbroker.log.filter.default.<source-name>.enable プロパティと vbroker.log.filter.default.<source-name>.logLevel プロパティを使用してソース名を明示的に制御する前に，このプロパティを使用してソース名を登録しておく必要があります。

プロパティ	デフォルト値	説明
vbroker.log.default.filter.<source-name>.enable	true	ソース名が登録されている場合，このプロパティを使用して，ソースからのログ出力を明示的に制御できます。 指定できる値は，true または false です。
vbroker.log.default.filter.<source-name>.logLevel	debug	このプロパティを使用すると，グローバルなログレベルのプロパティを詳細に制御できます。このプロパティを使用して明示的に指定されたログレベルは，特定のソース名に適用されます。 指定できる値は，グローバルな logLevel プロパティの値と同じです。
vbroker.log.default.filter.all.enable	true	これは，前のプロパティの特殊なケースで，組み込みのソース名 "all" を使用します。"all" は，登録されていないすべてのソース名を示します。
vbroker.log.enableSigHandler	false	true に設定すると，SIGUSR2 に基づくシグナルハンドラをインストールして，実行時のログの切り替えを可能にします。 指定できる値は，true または false です。 メモ：UNIX プラットフォームにのみ適用されます。

ログの出力は，簡単なレイアウトまたは複雑な Log4J XML イベントレイアウト（形式）で，コンソールまたはローリンググローカルシステムファイルの一方または両方に追加（転送）できます。デフォルトでは，ログは簡単なレイアウトでコンソールに追加されます。サポートされているさまざまなアペンダとレイアウトは，次のとおりです。

- `stdout` - コンソールアペンダの名前。
- `rolling` - ローリングファイルアペンダの名前。
- `simple` - 簡単な定義済み出力レイアウトの名前。
- `xml` - Log4J XML イベントレイアウトの名前。
- `full` - 全レコードフィールド出力レイアウトの名前。

次の表に，ログの出力先と形式を設定するためのプロパティおよびその説明を示します。

プロパティ	デフォルト値	説明
vbroker.log.default.appenders		ログの出力先を指定するためのコンマで区切られたアペンダインスタンス名のリスト。
vbroker.log.default.appender.<appender-inst-name>.appenderType	stdout	ロガーに設定する必要があるアペンダインスタンスのタイプ。値は，stdout，rolling，またはカスタムアペンダタイプになります。
vbroker.log.default.appender.<appender-inst-name>.layoutType	simple	登録されているアペンダの宛先に関連付けるレイアウト（形式）の種類。 値は，simple，xml，またはカスタムレイアウトタイプになります。

例

組み込みローリングアペンダタイプの場合は，次の設定を作成できます。プロパティは下で説明されています。各アペンダインスタンスについて，アペンダタイプが "rolling" に指定されているとします。

プロパティ	デフォルト値	説明
vbroker.log.default.appender.<appender-inst-name>.logDir	<current_directory>	ローリングログファイルを保存するディレクトリ。
vbroker.log.default.appender.<appender-inst-name>.fileNam	vbrolling.log	ローリングログファイルの名前。
vbroker.log.default.appender.<appender-inst-name>.maxFileSize	10	ロールオーバーする前の各バックアップのサイズ（MB 単位）。 1 以上の値を指定できます。
vbroker.log.default.appender.<appender-inst-name>.maxBackupIndex	1	必要なバックアップ数。0 に設定すると，バックアップは作成されず，ログがファイルに継続的に追加されます。 0 以上の値を指定できます。

カスタムアペンダ／レイアウトタイプを定義する場合は，次のプロパティを使用できます。

プロパティ	デフォルト値	説明
vbroker.log.appender.register		ロガーフレームワークに導入されるコンマ区切りの新しいアペンダタイプ名
vbroker.log.appender.<appender-type-name>.sharedLib		カスタムアペンダを含む共有ライブラリまたは DLL のファイル名を含む完全パス
vbroker.log.layout.register		ロガーフレームワークに導入されるコンマ区切りの新しいレイアウトタイプ名
vbroker.log.appender.<layout-type-name>.sharedLib		カスタムレイアウトを含む共有ライブラリまたは DLL のファイル名を含む完全パス

例

次に，デバッグログのプロパティの使用例を示します。サンプルコマンドで，*vbapp* は VisiBroker for C++ アプリケーションです。

1 デフォルトのログレベルのログをオンにする

```
prompt> vbapp -Dvbroker.log.enable=true
```

2 info レベル以上のログだけを追跡する

```
prompt> vbapp -Dvbroker.log.enable=true -Dvbroker.log.logLevel=info
```

3 エージェント関連のコンポーネントステートメントをオフにする

```
prompt> vbapp -Dvbroker.log.enable=true \
-Dvbroker.log.default.filter.register=agent \
-Dvbroker.log.default.filter.agent.enable=false
```

4 クライアントおよび接続関連領域だけを追跡する

```
prompt> vbapp -Dvbroker.log.enable=true \
-Dvbroker.log.default.filter.all.enable=false \
-Dvbroker.log.default.register=client,connection
```

5 se 領域の emerg レベルのログ，cdr 領域の err レベルのログ，およびその他の info レベルのログを追跡する

```
prompt> vbapp -Dvbroker.log.enable=true \
-Dvbroker.log.logLevel=info \
```

```
-Dvbroker.log.default.filter.register=se,cdr \
-Dvbroker.log.default.filter.se.logLevel=emerg \
-Dvbroker.log.default.filter.cdr.logLevel=err
```

6 バックアップを 3 つ作成するようにローカルファイルシステムへの出力を設定する

```
prompt> vbapp -Dvbroker.log.enable=true -Dvbroker.log.default.appenders=myappinst1 \

-Dvbroker.log.default.appender.myappinst1.appenderType=rolling \
-Dvbroker.log.default.appender.myappinst1.logDir=/opt/vbc \
-Dvbroker.log.default.appender.myappinst1.fileName=vbc.log \
-Dvbroker.log.default.appender.myappinst1.maxBackupIndex=3
```

7 コンソールとローカルファイルシステムの両方への XML 形式での出力を設定する

```
prompt> vbapp -Dvbroker.log.enable=true -
Dvbroker.log.default.appenders=myappinst1,myappinst2\

-Dvbroker.log.default.appender.myappinst1.appenderType=rolling \
-Dvbroker.log.default.appender.myappinst2.appenderType=stdout \
-Dvbroker.log.default.appender.myappinst1.logDir=/opt/vbc \
-Dvbroker.log.default.appender.myappinst1.fileName=vbc.log \
-Dvbroker.log.default.appender.myappinst1.layoutType=xml \
-Dvbroker.log.default.appender.myappinst2.layoutType=xml
```

8 2 つのアペンダインスタンス（stdout とカスタムアペンダ）への出力を設定する

```
prompt> vbapp -Dvbroker.log.enable=true \
-Dvbroker.log.appender.register=mycustomapp \
-Dvbroker.log.appender.mycustomapp.sharedLib=libCustomApp.so \
-Dvbroker.log.layout.register=mycustomlyt \
-Dvbroker.log.layout.mycustomlyt.sharedLib=libCustomLyt.so \
-Dvbroker.log.default.appenders=myappinst1,myappinst2 \
-Dvbroker.log.default.appender.myappinst1.appenderType=mycustomapp \
-Dvbroker.log.default.appender.myappinst1.layoutType=simple \
-Dvbroker.log.default.appender.myappinst2.appenderType=stdout \
-Dvbroker.log.default.appender.myappinst2.layoutType=mycustomlyt
```

Web サービスランタイムのプロパティ

ランタイムを有効にする

プロパティ	デフォルト値	説明
vbroker.ws.enable	false	ブール値パラメータ true または false を受け取ります。この値を true に設定すると，VisiBroker Web サービスランタイムが有効になります。

Web サービス HTTP リスナープロパティ

プロパティ	デフォルト値	説明
vbroker.ws.listener.host	Null	リスナーによって使用されるホスト名を指定します。デフォルトの null の場合は，システムからホスト名が取得されます。
vbroker.ws.listener.port	<8080>	リスナーソケットによって使用されるポート番号を指定します。

Web サービス接続マネージャのプロパティ

プロパティ	デフォルト値	説明
vbroke.ws.keepAliveConnection	False	HTTP サーバーは使用後に接続を閉じます。true に設定すると，接続を維持しようとします。
vbroker.ws.connectionMax	0	keepAliveConnection が true の場合，このプロパティは，サーバーが受け入れる最大接続数を指定します。デフォルト値 0 の場合は，無制限になります。
vbroker.ws.connectionMaxIdle	0	keepAliveConnection が true の場合，このプロパティは，未使用の接続が維持される最大時間を指定します。
vbroker.ws.garbageCollectTimer	30	keepAliveConnection が true の場合，このプロパティは，未使用の接続を削除するガベージコレクションの間隔を指定します。デフォルトは 30 秒です。
vbroker.ws.connection.rcvBufSize	0	クライアント接続ソケットの受信バッファソケットオプション。デフォルト値 0 の場合は，システム依存の値が使用されます。
vbroker.ws.connection. sendBufSize	0	クライアント接続ソケットの送信バッファソケットオプション。デフォルト値 0 の場合は，システム依存の値が使用されます。
vbroker.ws.connection.socketLinger	0	クライアント接続ソケットの TCP ソケットオプション。
vbroker.ws.connection.keepAlive	true	クライアント接続ソケットの TCP ソケットオプション。

SOAP 要求ディスパッチャのプロパティ

プロパティ	デフォルト値	説明
vbroker.ws.dispatcher.threadMax	0	スレッドプールディスパッチャに存在するスレッドの最大数。デフォルト値 0 の場合，スレッド数は無制限になります。
vbroker.ws.dispatcher.threadMin	0	スレッドプールディスパッチャに存在するスレッドの最小数。
vbroker.ws.dispatcher.threadMaxIdle	300	スレッドプールにあるアイドル状態のスレッドを破棄するまでの時間（秒単位）。
vbroker.ws.dispatcher.threadStackSize	0	スレッドプールディスパッチャスレッドのスタックサイズ。デフォルト値 0 の場合は，システム依存の値が使用されます。

第 7 章

例外処理

CORBA モデルにおける例外

CORBA モデルにおける例外には，*システム例外*と*ユーザー例外*があります。CORBA 仕様では，システム例外のセットが定義されています。システム例外は，クライアント要求の処理中にエラーが発生すると生成されます。また，通信エラーがあった場合にも生成されます。システム例外は特に決まった場合でなくても生成されることがあり，インターフェース内で宣言する必要はありません。

ユーザー例外を作成したオブジェクトの IDL 内で定義しておき，生成される条件を指定できます。定義はメソッドのシグニチャに入れます。クライアント要求の処理中にオブジェクトが例外を生成した場合は，VisiBroker ORB がその情報をクライアントに通知する役割を果たします。

システム例外

インターセプタを介してオブジェクトインプリメンテーションから生成することもありますが，システム例外は通常 VisiBroker ORB によって生成されます。インターセプタの詳細については，第 25 章「VisiBroker インターセプタの使い方」を参照してください。`SystemException` は，下に表示するように CORBA 定義のエラー条件の 1 つです。

これらの例外の説明および考えられる原因の一覧については，第 34 章「CORBA 例外」を参照してください。

表 7.1　CORBA 定義のシステム例外

例外の名前	説明
BAD_CONTEXT	コンテキストオブジェクトの処理エラー。
BAD_INV_ORDER	ルーチン呼び出し順序の不正。
BAD_OPERATION	無効なオペレーション。
BAD_PARAM	無効なパラメータが渡された。
BAD_QOS	QoS（Quality of service）をサポートできません。
BAD_TYPECODE	無効なタイプコード。

表 7.1　CORBA 定義のシステム例外

例外の名前	説明
COMM_FAILURE	通信の障害。
DATA_CONVERSION	データ変換のエラー。
FREE_MEM	メモリを解放できない。
IMP_LIMIT	インプリメンテーションの制限の違反。
INITIALIZE	VisiBroker ORB の初期化エラー。
INTERNAL	VisiBroker ORB の内部エラー。
INTF_REPOS	インターフェースリポジトリへのアクセスエラー。
INV_FLAG	無効なフラグが指定された。
INV_INDENT	識別子の構文が不正。
INV_OBJREF	無効なオブジェクトリファレンスが指定された。
INVALID_TRANSACTION	指定されたトランザクションが無効な場合（VisiTransact との組み合わせ）。
MARSHAL	パラメータまたは結果のマーシャリングエラー。
NO_IMPLEMENT	オペレーションのインプリメンテーションを利用できない。
NO_MEMORY	動的なメモリ割り当てのエラー。
NO_PERMISSION	要求されたオペレーションを実行する権限がない。
NO_RESOURCES	リソース不足のために要求を処理できない。
NO_RESPONSE	要求に対する有効な応答がまだない。
OBJ_ADAPTOR	オブジェクトアダプタによってエラーが検出された。
OBJECT_NOT_EXIST	オブジェクトが利用できない。
PERSIST_STORE	永続的ストレージの障害。
TRANSIENT	一時的な障害。
TRANSACTION_MODE	IOR の TransactionPolicy と現在のトランザクションモードの間に不一致が検出された場合（VisiTransact との組み合わせ）。
TRANSACTION_REQUIRED	トランザクションが要求された場合（VisiTransact との組み合わせ）。
TRANSACTION_ROLLEDBACK	トランザクションがロールバックされた場合（VisiTransact との組み合わせ）。
TRANSACTION_UNAVAILABLE	VisiTransact Transaction Service への接続が異常終了した場合。
TIMEOUT	要求タイムアウト。
UNKNOWN	未知の例外。

上記の例外の説明および考えられる原因の一覧については，第 34 章「CORBA 例外」を参照してください。

SystemException クラス

```
class SystemException : public CORBA::Exception {
   public:
      static const char      *_id;
      virtual       ~SystemException();
      CORBA::ULong minor() const;
      void       minor(CORBA::ULong val);
      CORBA::CompletionStatus       completed() const;
      void completed(CORBA::CompletionStatus status);
      ...
      static SystemException *_downcast(Exception *);
      ...
};
```

完了状態の取得

システム例外は，例外を生成した処理が完了したかどうかを通知するための完了状態を持ちます。下のサンプルは，CompletionStatus が CompletionStatus 用に列挙した値です。処理の完了状態が判定できない場合は，COMPLETED_MAYBE が返されます。

```
enum CompletionStatus {
   COMPLETED_YES = 0;
   COMPLETED_NO = 1;
   COMPLETED_MAYBE = 2;
};
```

これらの SystemException メソッドを使って完了状態を取得できます。

```
CompletionStatus completed();
```

マイナーコードの取得と設定

これらの SystemException メソッドを使って minor の取得と設定ができます。マイナーコードは，エラータイプに関する詳しい情報を取得するために使用します。

```
ULong minor() const;
void minor(ULong val);
```

システム例外の種類の判定

VisiBroker の例外クラスでは，プログラムで任意の例外をキャッチして _downcast() メソッドを使ってタイプを判定できます。静的メソッド _downcast() は，任意の Exception オブジェクトへのポインタを受け取ります。CORBA::Object で定義された _downcast() メソッドと同様に，ポインタの種類が SystemException の場合，_downcast() はユーザーにポインタを返します。ポインタが SystemException 型でない場合，_downcast() は NULL ポインタを返します。

システム例外のキャッチ

アプリケーションでは，VisiBroker ORB とリモート呼び出しを try ブロックと catch ブロックで囲む必要があります。下のサンプルコードは，第 3 章「VisiBroker を使ったサンプルアプリケーションの開発」で説明したアカウントクライアントプログラムが例外を出力する方法を示したものです。

```
#include "Bank_c.hh"
int main(int argc, char* const* argv) {
   try {
      CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
      PortableServer::ObjectId_var managerId =
            PortableServer::string_to_ObjectId("BankManager");
      Bank::AccountManager_var manager =
            Bank::AccountManager::_bind("/bank_agent_poa", managerId);
      const char* name = argc > 1 ? argv[1] : "Jack B. Quick";
      Bank::Account_var account = manager->open(name);
      CORBA::Float balance = account->balance();
      cout << "The balance in " << name << "'s account is $" << balance << endl;
   } catch(const CORBA::Exception& e) {
      cerr << e << endl;
      return 1;
   }
   return 0;
}
```

このように変更したクライアントプログラムをサーバーが存在しない状態で実行すると，次のように処理が終了しなかったこと，およびその例外が生成された原因が出力されます。

```
prompt>Client
Exception: CORBA::OBJECT_NOT_EXIST
        Minor: 0
        Completion Status: NO
```

例外をシステム例外にダウンキャスト

キャッチした例外に対して SystemException にダウンキャストを試みるように，Account クライアントプログラムを変更できます。下のサンプルコードは，クライアントプログラムの変更方法を示します。

```
int main(int argc, char* const* argv) {
   try {
      // ORB を初期化します。
      CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
      // 口座にバインドします。
      Account_var account = Account::_bind();
      // 口座の残高を取得します。
      CORBA::Float acct_balance = account->balance();
      // 残高を印刷します。
      cout << "The balance in the account is $"
            << acct_balance << endl;
      } catch(const CORBA::Exception& e) {
      CORBA::SystemException* sys_excep;
         sys_excep = CORBA::SystemException::_downcast((CORBA::Exception*)&e);
         if(sys_excep != NULL) {
            cerr << "System Exception occurred:" << endl;
            cerr << "exception name: " <<
            sys_excep->_name() << endl;
         cerr << "minor code: " << sys_excep->minor() << endl;
         cerr << "completion code: " << sys_excep->completed() << endl;
      } else {
         cerr << "Not a system exception" << endl;
         cerr << e << endl;
         }
      }
}
```

次のサンプルコードは，システム例外が発生した場合の出力結果を示します。

```
System Exception occurred:
      exception name: CORBA::NO_IMPLEMENT
      minor code: 0
      completion code: 1
```

特定のシステム例外のキャッチ

すべての種類の例外をキャッチするのではなく，特定の例外だけをキャッチできます。次のサンプルコードは，この方法を示します。

```
...
int main(int argc, char* const* argv) {
   try {
      // ORB を初期化します。
      CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
      // 口座にバインドします。
      Account_var account = Account::_bind();
      // 残高を取得します。
      CORBA::Float acct_balance = account->balance();
      // 残高を印刷します。
      cout << "The balance in the account is $"                << acct_balance << endl;
   }
   // システムエラーをチェックします。
   catch(const CORBA::SystemException& sys_excep) {
```

```
        cout << "System Exception occurred:" << endl;
        cout << "      exception name: " <<                     sys_excep->_name() <<
endl;
      cout << "      minor code: " <<                sys_excep->minor() << endl;
      cout << "      completion code: " <<                sys_excep->completed() << endl;
    }
}
...
```

ユーザー例外

オブジェクトのインターフェースを IDL で定義する際に，そのオブジェクトが生成するユーザー例外を指定できます。次のサンプルコードは UserException のコードです。idl2cpp コンパイラは，オブジェクトに指定されたユーザー例外をこのコードから派生します。

```
class UserException: public Exception {
   public:
      ...
      static const char         *_id;
      virtual ~UserException();
      static UserException *_downcast(Exception *);
};
```

ユーザー例外の定義

第 3 章「VisiBroker を使ったサンプルアプリケーションの開発」で紹介した Bank Account アプリケーションをさらに拡張して，account オブジェクトが例外を生成するようにします。account オブジェクトに十分な資金がない場合は，AccountFrozen という名前のユーザー例外を生成します。ユーザー例外を追加するために Account インターフェースの IDL 仕様に必要になる部分を太字で示します。

```
// Bank.idl
module Bank {
   interface Account {
      exception AccountFrozen {
      };
      float balance() raises(AccountFrozen);
   };
};
```

idl2cpp コンパイラは，AccountFrozen 例外クラスに次のコードを生成します。

```
class Account : public virtual CORBA::Object {
   ...
   class AccountFrozen: public CORBA_UserException {
      public:
         static const CORBA_Exception::Description description;
         AccountFrozen() {}
         static CORBA::Exception *_factory() {
            return new AccountFrozen();
         }
         ~AccountFrozen() {}
         virtual const CORBA_Exception::Description& _desc() const;
         static AccountFrozen *_downcast(CORBA::Exception *exc);
         CORBA::Exception *_deep_copy() const {
            return new AccountFrozen(*this);
         }
         void _raise() const {
            raise *this;
         }
   ...
}
```

例外を生成するためのオブジェクトの変更

適切なエラー条件下で例外が生成されるように，AccountImpl オブジェクトを変更する必要があります。

```
CORBA::Float AccountImpl::balance()
{
      if( _balance < 50 ) {
            raise Account::AccountFrozen();
      } else {
            return _balance;
}
```

ユーザー例外のキャッチ

オブジェクトインプリメンテーションが例外を生成した場合は，ORB がその例外をクライアントプログラムに通知する役割を果たします。UserException のチェック方法は，SystemException チェックと似ています。AccountFrozen 例外をキャッチするようにアカウントクライアントプログラムを変更するには，下に示すようにコードを変更します。

```
...
   try {
      // ORB を初期化します。
      CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
      // 口座にバインドします。
      Account_var account = Account::_bind();
      // 口座の残高を取得します。
      CORBA::Float acct_balance = account->balance();
   }
   catch(const Account::AccountFrozen& e) {
      cerr << "AccountFrozen returned:" << endl;
      cerr << e << endl;
      return(0);
   }
   // システムエラーをチェックします。
   catch(const CORBA::SystemException& sys_excep) {
   }
...
```

ユーザー例外へのフィールドの追加

ユーザー例外に値を関連付けることができます。下のサンプルコードは，原因コードを AccountFrozen ユーザー例外に追加するために，IDL インターフェース仕様を変更する方法です。例外を生成させるオブジェクトインプリメンテーションで原因コードを設定します。原因コードは，例外が出力ストリームに書き込まれるときに自動的に出力されます。

```
// Bank.idl
module Bank {
   interface Account {
      exception AccountFrozen {
         int reason;
      };
      float balance() raises(AccountFrozen);
   };
};
```

第 8 章

サーバーの基礎

この章では，クライアントの要求を受け取るサーバーを設定するために必要な作業を簡単に説明します。

概要

サーバーを準備する作業には，基本的に次のような操作を行います。

- VisiBroker ORB の初期化
- POA の作成と設定
- POA マネージャをアクティブ化します。
- オブジェクトのアクティブ化
- クライアント要求の待機

この章では，特に注意が必要な概念を紹介するために，各作業の概要について説明します。各手順の詳細は，個別の必要条件によって異なります。

VisiBroker ORB の初期化

前の章で説明したように，VisiBroker ORB は，クライアント要求とオブジェクトインプリメンテーションの間の通信リンクを提供します。VisiBroker ORB で通信するには，次のように，事前に各アプリケーションで VisiBroker ORB を初期化する必要があります。

```
// VisiBroker ORB の初期化
CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
```

POA の作成

以前のバージョンの CORBA オブジェクトアダプタ（*基本オブジェクトアダプタ*，*BOA*）では，ポータブルオブジェクトサーバーのコードを使用できませんでした。この問題に対処するため，OMG は新しい仕様を開発し，*ポータブルオブジェクトアダプタ*（POA）を作成しました。

メモ　POA について説明し始めるとページがいくらあっても足りなくなるので，ここでは POA の基本機能の一部だけを紹介します。詳細については，第 9 章「POA の使い方」と OMG 仕様を参照してください。

POA とそのコンポーネントを一言でいうと，クライアント要求を受け取ったときに呼び出される**サーバント**を決定するものです。サーバントは**抽象オブジェクト**のインプリメンテーションを提供するプログラミングオブジェクトであり，サーバントは CORBA オブジェクトではありません。

各 VisiBroker ORB は POA（*rootPOA*）を 1 つずつ提供します。新しい POA を追加作成して別の動作を設定することができます。また，POA が制御するオブジェクトの特性を定義することもできます。

POA にサーバントを設定するには，次のような作業を実行します。

- ルート POA へのリファレンスの取得
- POA ポリシーの定義
- ルート POA の子 POA の作成
- サーバントの作成とアクティブ化
- POA マネージャを介した POA のアクティブ化

手順はアプリケーションによって異なる場合もあります。

ルート POA へのリファレンスの取得

すべてのサーバーアプリケーションは，オブジェクトを管理したり，新しい POA を作成するために，ルート POA へのリファレンスを取得する必要があります。

```
// ルート POA へのリファレンスを取得します。
CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
// オブジェクトリファレンスを POA へのリファレンスにナローイングします。
PortableServer::POA_var rootPOA = PortableServer::POA::_narrow(obj);
```

ルート POA へのリファレンスは，`resolve_initial_references` で取得します。`resolve_initial_references` は，`CORBA::Object` 型の値を返します。返されるオブジェクトリファレンスは，目的の型にナローイングする必要があります。この例では，`PortableServer::POA` にナローイングしています。

さらに，必要に応じてこのリファレンスを使って別の POA を作成することも可能です。

子 POA の作成

ルート POA には定義済みの**ポリシー**のセットがあり，これらのポリシーは変更できません。ポリシーは，POA の動作や POA が管理するオブジェクトを制御するオブジェクトです。異なる存続期間のポリシーなど，別の動作が必要な場合は新しい POA を作成する必要があります。

新しい POA は `create_POA` で作成し，既存の POA の子 POA になります。POA は必要な数だけ作成できます。

メモ　子 POA は，親 POA のポリシーを継承しません。

次の例では，ルート POA から永続的な存続期間ポリシーを持つ子 POA を作成しています。子 POA の状態を制御するには，ルート POA の POA マネージャを使用します。

```
CORBA::PolicyList policies;
policies.length(1);
policies[(CORBA::ULong)0] = rootPOA->create_lifespan_policy(
      PortableServer::PERSISTENT);
// 適切なポリシーで myPOA を作成します。
PortableServer::POAManager_var rootManager = rootPOA->the_POAManager();
```

```
PortableServer::POA_var myPOA = rootPOA->create_POA( "bank_agent_poa",
      rootManager, policies );
```

サーバントメソッドの実装

IDL は C++ と似た構文を持っており，モジュール，インターフェース，データ構造などの定義に使用します。インターフェースを含む IDL をコンパイルすると，サーバントの基底クラスとして機能するクラスが生成されます。たとえば，Bank.IDL ファイルには AccountManager インターフェースを記述します。

```
module Bank{
   interface Account {
         float balance();
   };
   interface AccountManager {
         Account open (in string name);
   };
};
```

次に示すのは，サーバー側の AccountManager インプリメンテーションです。

```
class AccountManagerImpl : public POA_Bank::AccountManager {
   private:
      Dictionary _accounts;
   public:
      virtual Bank::Account_ptr open(const char* name) {
         // account ディレクトリ内で口座を検索します。
         Bank::Account_ptr account = (Bank::Account_ptr) _accounts.get(name);
         if(account == Bank::Account::_nil()) {
            // 0 ～ 1000 ドルの範囲で口座に残高を設定します。
            float balance = abs(rand()) % 100000 / 100.0;
            // その残高で口座のインプリメンテーションを作成します。
            AccountImpl *accountServant = new AccountImpl(balance);
            try {
               // デフォルト POA（このサーバントのルート POA）でアクティブ化します。
                        servant
               PortableServer::POA_var rootPOA = _default_POA();
               CORBA::Object_var obj =
                        rootPOA->servant_to_reference(accountServant);
               account = Bank::Account::_narrow(obj);
            } catch(const CORBA::Exception& e) {
               cerr << "_narrow caught exception: " << e << endl;
            }
            // 新しい口座を出力します。
            cout << "Created " << name << "'s account: " << account << endl;
            // 口座を account ディレクトリに保存します。
            _accounts.put(name, account);
         }
         // 口座を返します。
         return Bank::Account::_duplicate(account);
      }
};
```

サーバントの作成およびアクティブ化

AccountManager インプリメンテーションはサーバーのコード内で作成して，アクティブ化する必要があります。このサンプルで，AccountManager は activate_object_with_id でアクティブ化され，**アクティブオブジェクトマップ**にオブジェクト ID を渡して記録します。アクティブオブジェクトマップは，ID をサーバントにマッピングするテーブルです。この方法では，POA がアクティブなときにいつでもこのオブジェクトを使用できます。この方法を**明示的なオブジェクトのアクティブ化**と呼びます。

```
// サーバントを作成します。

AccountManagerImpl managerServant;
// サーバントの ID を決定します。
PortableServer::ObjectId_var managerId =
      PortableServer::string_to_ObjectId("BankManager");
// その ID を使って myPOA でサーバントをアクティブ化します。
myPOA->activate_object_with_id(managerId,&managerServant);
```

POA のアクティブ化

最後に行う操作として，POA に関連付けられている POA マネージャをアクティブ化します。デフォルトの POA マネージャは，作成された時点で**停止状態**になります。この状態ではすべての要求が停止キューに入れられ，すぐには処理されません。要求を送るには，POA に関連付けられている **POA マネージャ**を停止状態からアクティブな状態に変更する必要があります。POA マネージャは POA の状態を制御するオブジェクトです。POA の状態には要求がキューに入る状態，処理される状態，および破棄される状態があります。POA マネージャは，POA の作成時に POA に関連付けられます。使用する POA マネージャは指定できます。また，`create_POA()` で POA マネージャ名に `null` を指定して，システムに新しい POA マネージャの作成を任せることもできます。

```
// POA マネージャをアクティブ化
rootPOA.the_POAManager().activate();
```

オブジェクトのアクティブ化

前の節では明示的なオブジェクトのアクティブ化について簡単に説明しましたが，実際にオブジェクトをアクティブ化するには，次のような方法があります。

- **明示的**：POA の呼び出しを介して，サーバーの起動時にすべてのオブジェクトがアクティブ化されます。
- **オンデマンド**：オブジェクト ID に関連付けられていないサーバントへの要求を受け取ると，サーバントマネージャがオブジェクトをアクティブ化します。
- **暗黙的**：任意のクライアント要求ではなく，POA によるオペレーションへの応答の中でサーバーが暗黙的にオブジェクトをアクティブ化します。
- **デフォルトサーバント**：POA がデフォルトサーバントでクライアント要求を処理します。

オブジェクトのアクティブ化の詳細については，第 9 章「POA の使い方」を参照してください。ここでは，オブジェクトは，いくつかの方法でアクティブ化できることだけ気に留めておいてください。

クライアント要求の待機

POA の設定が完了したら，`orb.run()` でクライアント要求を待機します。このプロセスは，サーバーが終了するまで実行を続けます。

```
// 着信要求を待機します。
orb.run();
```

完全なサンプルコード

次にサンプルコード全体を示します。

```
// Server.C
#include "Bank_s.hh"
```

```
#include <math.h>
class Dictionary {
   private:
      struct Data {
         const char* name;
         void* value;
      };
      unsigned _count;
      Data* _data;
   public:
      Dictionary() {
         _count = 0;
      }
      void put(const char* name, void* value) {
         Data* oldData = _data;
         _data = new Data[_count + 1];
         for(unsigned i = 0; i < _count; i++) {
            _data[i] = oldData[i];
         }
         _data[_count].name = strdup(name);
         _data[_count].value = value;
         _count++;
      }
      void* get(const char* name) {
         for(unsigned i = 0; i < _count; i++) {
            if(!strcmp(name, _data[i].name)) {
               return _data[i].value;
            }
         }
         return 0;
      }
};
class AccountImpl : public POA_Bank::Account {
   private:
      float _balance;
   public:
      AccountImpl(float balance) {
         _balance = balance;
      }
      virtual float balance() {
         return _balance;
      }
};
class AccountManagerImpl : public POA_Bank::AccountManager {
   private:
      Dictionary _accounts;
   public:
      virtual Bank::Account_ptr open(const char* name) {
         // account ディレクトリ内で口座を検索します。
         Bank::Account_ptr account = (Bank::Account_ptr) _accounts.get(name);
         if(account == Bank::Account::_nil()) {
            // 0 〜 1000 ドルの範囲で口座に残高を設定します。
            float balance = abs(rand()) % 100000 / 100.0;
            // その残高で口座のインプリメンテーションを作成します。
            AccountImpl *accountServant = new AccountImpl(balance);
            try {
            // デフォルト POA（このサーバントのルート POA）でアクティブ化します。
               servant
               PortableServer::POA_var rootPOA = _default_POA();
               CORBA::Object_var obj =
                        rootPOA->servant_to_reference(accountServant);
               account = Bank::Account::_narrow(obj);
            } catch(const CORBA::Exception& e) {
               cerr << "_narrow caught exception: " << e << endl;
            }
            // 新しい口座を出力します。
```

```
            cout << "Created " << name << "'s account: " << account << endl;
            // 口座を account ディレクトリに保存します。
            _accounts.put(name, account);
         }
         // 口座を返します。
         return Bank::Account::_duplicate(account);
      }
};
int main(int argc, char* const* argv) {
   try {
      // ORB を初期化します。
      CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
      // ルート POA へのリファレンスを取得します。
      CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
      // オブジェクトリファレンスを POA へのリファレンスにナローイングします。
      PortableServer::POA_var rootPOA = PortableServer::POA::_narrow(obj);
      CORBA::PolicyList policies;
      policies.length(1);
      policies[(CORBA::ULong)0] = rootPOA->create_lifespan_policy(
            PortableServer::PERSISTENT
      );
      // 適切なポリシーで myPOA を作成します。
      PortableServer::POAManager_var rootManager = rootPOA->the_POAManager();
      PortableServer::POA_var myPOA = rootPOA->create_POA( "bank_agent_poa",
            rootManager, policies );
      // サーバントを作成します。
      AccountManagerImpl managerServant;
      // サーバントの ID を決定します。
      PortableServer::ObjectId_var managerId =
            PortableServer::string_to_ObjectId("BankManager");
      // その ID を使って myPOA でサーバントをアクティブ化します。
      myPOA->activate_object_with_id(managerId,&managerServant);
      // POA マネージャをアクティブ化します。
      rootPOA->the_POAManager()->activate();
      cout << myPOA->servant_to_reference(&managerServant) << " is ready" << endl;
      // 着信要求を待機します。
      orb->run();
   } catch(const CORBA::Exception& e) {
      cerr << e << endl;
   }
}
```

第 9 章

POA の使い方

ポータブルオブジェクトアダプタの概要

ポータブルオブジェクトアダプタ（POA）は，サーバー側に移植性を提供するために開発され，基本オブジェクトアダプタに代わるオブジェクトアダプタです。

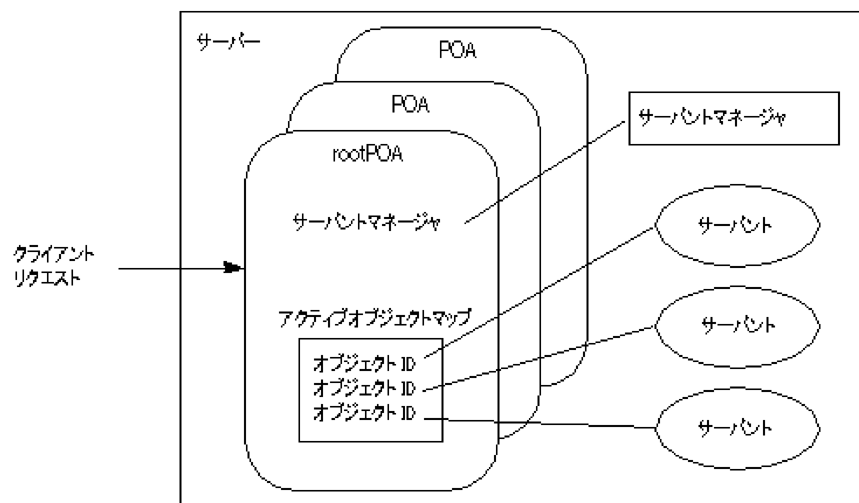
POA は，オブジェクトのインプリメンテーションと VisiBroker ORB を仲介します。POA は，仲介役として，要求をサーバントに送ります。その結果，サーバントが実行され，必要に応じて子 POA が作成されます。

サーバーは複数の POA をサポートできます。POA は少なくとも 1 つ存在する必要があり，これはルート POA と呼ばれます。ルート POA は自動的に作成されます。POA の集合は階層構造で，すべての POA の上位にルート POA があります。

サーバントマネージャは，POA のためにサーバントを探し，それをオブジェクトに割り当てます。サーバントに割り当てられた抽象オブジェクトをアクティブオブジェクトと呼びます。このサーバントは，アクティブオブジェクトを具現化していると言えます。各 POA は 1 つのアクティブオブジェクトマップを持ち，そこでアクティブオブジェクトのオブジェクト ID および関連するサーバントの最新情報を管理します。

メモ　VisiBroker 6.0 より前のバージョンをご使用だったユーザーは，継承階層の変更点に注意してください。これは，ローカルインターフェースを必要とする CORBA 仕様 2.6 をサポートするためです。たとえば，ServantLocator インプリメンテーションは，`org.omg.PortableServer.ServantLocatorPOA` ではなく，`org.omg.PortableServer._ServantLocatorLocalBase` から拡張されるようになります。

図 9.1 POA の概要



POA の用語

この章を読み進める上で必要な各用語の定義を次に示します。

表 9.1 ポータブルオブジェクトアダプタ（POA）の用語

用語	説明
アクティブオブジェクトマップ	オブジェクト ID を介して，アクティブな VisiBroker CORBA オブジェクトをサーバントにマッピングするためのテーブルです。POA ごとに 1 つのアクティブオブジェクトマップがあります。
アダプタアクティベータ	存在しない子 POA への要求を受け取ったとき，オンデマンドでオブジェクトを作成できるオブジェクトです。
気化	サーバントと抽象 CORBA オブジェクトの関連を削除することです。
具現化	サーバントを抽象 CORBA オブジェクトに関連付けることです。
ObjectID	オブジェクトアダプタ内で CORBA オブジェクトを識別する手段になります。オブジェクト ID は，オブジェクトアダプタまたはアプリケーションによって割り当てられ，作成されたオブジェクトアダプタ内でのみ一意です。サーバントは，オブジェクト ID を介して抽象オブジェクトに関連付けられます。
永続的オブジェクト	作成元になったサーバープロセスの外部でも有効な CORBA オブジェクトです。
POA マネージャ	POA の状態を制御するオブジェクトです。たとえば，POA が，着信した要求を受信するか，それとも破棄するかを制御します。
ポリシー	関連する POA，およびその POA が管理するオブジェクトの動作を制御するオブジェクトです。
rootPOA	作成された VisiBroker ORB は，ルート POA と呼ばれる POA をそれぞれ 1 つずつ持ちます。必要に応じて，ルート POA から追加の POA を作成することもできます。
servant	CORBA オブジェクトのメソッドを実装するコードです。CORBA オブジェクトそのものではありません。
サーバントマネージャ	サーバントとオブジェクトの関連付けを管理したり，オブジェクトの存在を確認する役割を持つオブジェクトです。複数のサーバントマネージャが存在できます。
一時的オブジェクト	作成元になったプロセスの内部でのみ有効な CORBA オブジェクトです。

POA の作成と使用の手順

手順の細部は異なりますが，ここでは，POA の存続期間内に行う基本的な手順を示します。

1. POA ポリシーを定義します。
2. POA を作成します。
3. POA マネージャを使って POA をアクティブ化します。
4. サーバントを作成し，アクティブ化します。
5. サーバントマネージャを作成し，使用します。
6. アダプタアクティベータを使用します。

目的に応じて，これらの手順の一部は省略できます。たとえば，POA で要求を処理する場合は，その POA をアクティブ化するだけです。

POA ポリシー

各 POA は，その特性を定義するポリシーのセットを持ちます。新しい POA の作成時には，デフォルトのポリシーセットも使用できますが，必要に応じて別の値も使用できます。ポリシーは，POA の作成時にだけ設定でき，既存の POA のポリシーは変更できません。POA は，親 POA のポリシーを継承しません。

次に，POA ポリシーとその値，およびルート POA が使用するデフォルト値を示します。

スレッドポリシー スレッドポリシーは，POA が使用するスレッドモデルを指定します。

ThreadPolicy の値は，次のいずれかになります。

ORB_CTRL_MODEL : デフォルトです。POA は，要求をスレッドに割り当てる役割を果たします。マルチスレッド環境では，同時に複数の要求があった場合，それらに複数のスレッドを提供します。VisiBroker はマルチスレッドモデルを使用します。

SINGLE_THREAD_MODEL : POA は要求を逐次処理します。マルチスレッド環境では，POA からサーバントやサーバントマネージャへの呼び出しは，すべてスレッドセーフです。

MAIN_THREAD_MODEL : 呼び出しは，識別された「メイン」スレッドで処理されます。すべてのメインスレッド POA に対する要求は，順番に処理されます。マルチスレッド環境では，このポリシーを持つ POA によって処理された呼び出しは，すべてスレッドセーフです。アプリケーションプログラマは，ORB::run() または ORB::perform_work() を呼び出してメインスレッドを指定します。以上のメソッドの詳細については，98 ページの「オブジェクトのアクティブ化」を参照してください。

存続期間ポリシー 存続期間ポリシーには，POA に実装されているオブジェクトの存続期間を指定します。

LifespanPolicy の値は，次のいずれかになります。

TRANSIENT : デフォルトです。POA がアクティブ化した一時的オブジェクトは，作成元の POA より長く存続することはできません。POA が非アクティブ化された後で，その POA によって生成されたオブジェクトリファレンスを使用しようとすると，OBJECT_NOT_EXIST 例外が生成されます。

PERSISTENT : POA がアクティブ化した永続的オブジェクトは，最初に作成されたプロセスより長く存続できます。永続的オブジェクトに対して要求を行うと，プロセス，POA，およびそのオブジェクトを実装するサーバントが暗黙的にアクティブ化されます。

オブジェクト ID の一意性ポリシー オブジェクト ID の一意性ポリシーを利用すると，複数の抽象オブジェクトが単一のサーバントを共有できます。

IdUniquenesspolicy の値は，次のいずれかになります。

UNIQUE_ID : デフォルトです。アクティブ化されたサーバントは，1 つのオブジェクト ID だけをサポートします。

MULTIPLE_ID : アクティブ化されたサーバントは，1 つ以上のオブジェクト ID を持つことができます。オブジェクト ID は，実行時に起動されるメソッド内で決定する必要があります。

ID の割り当てポリシー ID の割り当てポリシーには，オブジェクト ID をサーバーアプリケーションと POA のどちらから生成するかを指定します。

IdAssignmentPolicy の値は，次のいずれかになります。

USER_ID : オブジェクトは，アプリケーションからオブジェクト ID を割り当てられます。

SYSTEM_ID : デフォルトです。オブジェクトは，POA からオブジェクト ID を割り当てられます。同時に PERSISTENT ポリシーが設定されている場合，オブジェクト ID は，同じ POA のすべてのインスタンス化で一意である必要があります。

通常，USER_ID は永続的オブジェクト用であり，SYSTEM_ID は一時的オブジェクト用です。永続的オブジェクトに SYSTEM_ID を使用する場合は，サーバントまたはオブジェクトリファレンスからオブジェクトを抽出できます。

サーバント管理ポリシー サーバント管理ポリシーには，POA がアクティブオブジェクトマップでアクティブなサーバントを保持するかどうかを指定します。

ServantRetentionPolicy の値は，次のいずれかになります。

RETAIN : デフォルトです。POA は，アクティブオブジェクトマップでオブジェクトのアクティブ化を追跡します。RETAIN は，通常，サーバントアクティベータまたは POA の明示的なアクティブ化メソッドとともに使用します。

NON_RETAIN : POA は，アクティブオブジェクトマップでアクティブなサーバントを保持しません。NON_RETAIN は，サーバントロケータとともに使用する必要があります。

サーバントアクティベータとサーバントロケータは，サーバントマネージャの一種です。サーバントマネージャの詳細については，102 ページの「サーバントとサーバントマネージャの使い方」を参照してください。

要求処理ポリシー 要求処理ポリシーには，POA が要求を処理する方法を指定します。

USE_ACTIVE_OBJECT_MAP_ONLY : デフォルトです。アクティブオブジェクトマップ内のリストにオブジェクト ID がない場合は，OBJECT_NOT_EXIST 例外が返されます。この値には，RETAIN ポリシーと組み合わせて使用する必要があります。

USE_DEFAULT_SERVANT : アクティブオブジェクトマップのリストにオブジェクト ID がない場合，または NON_RETAIN ポリシーが設定されている場合は，デフォルトのサーバントに要求が送られます。デフォルトのサーバントが登録されていない場合は，OBJ_ADAPTER 例外が返されます。この値は，MULTIPLE_ID ポリシーと組み合わせて使用する必要があります。

USE_SERVANT_MANAGER : アクティブオブジェクトマップのリストにオブジェクト ID がない場合，または NON_RETAIN ポリシーが設定されている場合は，サーバントマネージャでサーバントを取得します。

暗黙的アクティブ化ポリシー 暗黙的アクティブ化ポリシーは，POA が暗黙的なサーバントのアクティブ化をサポートするかどうかを指定します。

ImplicitActivationPolicy の値は，次のいずれかになります。

IMPLICIT_ACTIVATION : POA が暗黙的なサーバントのアクティブ化をサポートします。サーバントをアクティブ化するには，次の 2 とおりの方法があります。

- `POA::servant_to_reference()` を使用して，サーバントをオブジェクトリファレンスに変換する。
- サーバントの `_this()` を呼び出す。

この値は，SYSTEM_ID ポリシーおよび RETAIN ポリシーと組み合わせて使用する必要があります。

NO_IMPLICIT_ACTIVATION : デフォルトです。POA は，暗黙的なサーバントのアクティブ化をサポートしません。

バインドサポートポリシー バインドサポートポリシー（VisiBroker 固有のポリシー）は，VisiBroker osagent による POA とアクティブオブジェクトの登録を制御します。数千ものオブジェクトがある場合，それらのすべてを osagent に登録することは困難です。このような場合は，POA を osagent に登録します。クライアントが要求を行うとき，POA の名前とオブジェクト ID がバインド要求に入れられるため，osagent は正しく要求を転送することができます。

BindSupportPolicy の値は，次のいずれかになります。

BY_INSTANCE : すべてのアクティブオブジェクトが osagent に登録されます。この値は，PERSISTENT ポリシーおよび RETAIN ポリシーと組み合わせて使用します。

BY_POA : デフォルトです。POA だけが osagent に登録されます。この値は，PERSISTENT ポリシーと組み合わせて使用します。

None : POA とアクティブオブジェクトのどちらも osagent に登録されません。

メモ rootPOA は，`NONE` アクティブ化ポリシーを使って作成されます。

POA の作成

POA でオブジェクトを実装するには, 少なくとも 1 つの POA オブジェクトがサーバー上に存在する必要があります。POA の存在を確実にするため，ORB の初期化中にルート POA が提供されます。この POA は，先に説明したデフォルトの POA ポリシーを使用します。

ルート POA を取得した後は，サーバー側の特定のポリシーセットを実装した子 POA を作成できます。

POA の命名規則

各 POA は，その名前と完全な POA 名（すべての階層を明示したパス名）を認識しています。階層は，スラッシュ（/）で表されます。たとえば，`/A/B/C` は，POA C が POA B の子 POA であり，POA B が POA A の子 POA であることを意味します。この例の最初のスラッシュは，ルート POA を表します。POA C に BindSupport:BY_POA ポリシーが設定されている場合，osagent には `/A/B/C` が登録され，クライアントは `/A/B/C` にバインドします。

POA 名にエスケープ文字やデリミタが含まれている場合, これらの文字を内部的に記録するとき，VisiBroker はその直前に 2 つのバックスラッシュ（¥¥）を付けます。たとえば，2 つの POA を次の階層でコーディングしたとします。

```
PortableServer::POA_var myPOA1 = rootPOA->create_POA("A/B",
        poa_manager,
        policies);
PortableServer::POA_var myPOA2 = myPOA1->create_POA("¥t",
        poa_manager,
        policies);
```

クライアントは，次のようにバインドします。

```
Bank::AccountManager_var manager = Bank::AccountManager::_bind("/A¥¥/B/¥t", managerId);
```

ルート POA の取得

次のサンプルコードに，サーバーアプリケーションがルート POA を取得する手続きを示します。

```
// ORB を初期化します。
CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
// ルート POA へのリファレンスを取得します。
PortableServer::POA_var rootPOA = PortableServer::POA::_narrow(obj);
```

メモ　resolve_initial_references メソッドは，CORBA::Object 型の値を返します。返されるオブジェクトリファレンスは，目的の型にナローイングする必要があります。前の例の，PortableServer::POA です。

POA ポリシーの設定

親 POA のポリシーは継承されません。POA に固有の特性を持たせる場合は，デフォルト値とは異なるすべてのポリシーを指定する必要があります。POA ポリシーの詳細については，95 ページの「POA ポリシー」を参照してください。

```
CORBA::PolicyList policies;
policies.length(1);
policies[(CORBA::ULong)0] = rootPOA->create_lifespan_policy(PortableServer::PERSISTENT);
```

POA の作成およびアクティブ化

POA は create_POA で親 POA に作成されます。POA には任意の名前を付けることができますが，同じ親を持つ POA どうしに同じ名前を付けることはできません。2 つの POA に同じ名前を付けようとすると，CORBA 例外（AdapterAlreadyExists）が発生します。

新しい POA を作成するには，次のように create_POA を使用します。

```
POA create_POA(POA_Name, POAManager, PolicyList);
```

POA マネージャは，POA の状態（要求の処理中であるかどうかなど）を制御します。POA マネージャの名前として null が create_POA に指定されると，新しい POA マネージャオブジェクトが作成され，POA に関連付けられます。通常，すべての POA に同じ POA マネージャを関連付けます。POA マネージャの詳細については，107 ページの「POA マネージャによる POA の管理」を参照してください。

POA マネージャと POA は，作成後，自動的にはアクティブ化されません。POA に関連付けられた POA マネージャをアクティブ化するには，activate() を使用します。次のサンプルコードは，POA を作成する例です。

```
CORBA::PolicyList policies;
policies.length(1);
policies[(CORBA::ULong)0] = rootPOA->create_lifespan_policy(PortableServer::PERSISTENT);
// 適切なポリシーで myPOA を作成します。
PortableServer::POAManager_var rootManager = rootPOA->the_POAManager();
PortableServer::POA_var myPOA =
 rootPOA->create_POA("bank_agent_poa", rootManager, policies);
```

オブジェクトのアクティブ化

CORBA オブジェクトがアクティブなサーバントに関連付けられ，POA のサーバント管理ポリシーが RETAIN であれば，関連付けられているオブジェクト ID がアクティブオブ

ジェクトマップに記録され，オブジェクトがアクティブ化されます。アクティブ化は，次のいずれかの方法で行われます。

99 ページの「オブジェクトの明示的なアクティブ化」	activate_object または activate_object_with_id を呼び出すことにより，サーバーアプリケーション自身が明示的にオブジェクトをアクティブ化します。
99 ページの「オンデマンドのオブジェクトのアクティブ化」	ユーザーが提供するサーバントマネージャを介してオブジェクトをアクティブ化するように，サーバーアプリケーションから POA に指示します。最初に，set_servant_manager で POA にサーバントマネージャを登録する必要があります。
100 ページの「オブジェクトの暗黙的なアクティブ化」	サーバーは，何らかの処理への応答としてのみ，オブジェクトをアクティブ化します。サーバントがアクティブでない場合，クライアントがこれをアクティブ化する手段はありません。たとえば，アクティブでないオブジェクトを要求しても，それをアクティブ化することはできません。
100 ページの「デフォルトサーバントによるアクティブ化」	POA が単一のサーバントを使ってすべてのオブジェクトを実装します。

オブジェクトの明示的なアクティブ化

POA に IdAssignmentPolicy::SYSTEM_ID を設定すると，オブジェクト ID を指定しなくても，オブジェクトを明示的にアクティブ化できます。サーバーが POA で activate_object を呼び出すと，オブジェクトがアクティブ化され，オブジェクト ID が割り当てられて返されます。一時的オブジェクトには，このアクティブ化方法がよく使用されます。オブジェクトとサーバントのどちらも長期間必要になることがないので，サーバントマネージャは不要です。

オブジェクト ID を使用して，明示的にオブジェクトをアクティブ化することもできます。たとえば，サーバーの初期化中に，ユーザーが activate_object_with_id を呼び出し，そのサーバーに管理されるすべてのオブジェクトをアクティブ化する操作は一般的です。すべてのオブジェクトがアクティブ化されるので，サーバントマネージャは不要です。存在しないオブジェクトに対する要求を受け取ると，OBJECT_NOT_EXIST 例外が生成されます。サーバーが大量のオブジェクトを管理している場合，この例外は明らかに悪影響を及ぼします。

次のサンプルコードは，activate_object_with_id を使った明示的アクティブ化の例です。

```
// サーバントを作成します。
AccountManagerImpl managerServant;
// サーバントの ID を決定します。
PortableServer::ObjectId_var managerId =
      PortableServer::string_to_ObjectId("BankManager");
// その ID を使って myPOA でサーバントをアクティブ化します。
myPOA->activate_object_with_id(managerId,&managerServant);
// POA マネージャをアクティブ化します。
PortableServer::POAManager_var rootManager = rootPOA->the_POAManager();
rootManger->activate();
```

オンデマンドのオブジェクトのアクティブ化

クライアントが，関連付けられたサーバントを持たないオブジェクトを要求すると，オンデマンドのアクティブ化が行われます。POA は，要求を受け取るとアクティブオブジェクトマップから，オブジェクト ID に関連付けられているアクティブなサーバントを探します。該当するサーバントが見つからない場合は，サーバントマネージャの incarnate を呼び出して，サーバントマネージャにそのオブジェクト ID 値を渡します。サーバントマネージャは，次の 3 つのいずれかの処理を行います。

- 要求に対して適切なオペレーションを実行するサーバントを探します。
- OBJECT_NOT_EXIST 例外を生成します。これはクライアントに返されます。
- 要求を別のオブジェクトに転送します。

POA ポリシーによっては，その他の処理も行われます。たとえば，RequestProcessingPolicy::USE_SERVANT_MANAGER と ServantRetentionPolicy::RETAIN が有効な場合，アクティブオブジェクトマップはサーバントとオブジェクト ID の関連付けによって更新されます。

次に，オンデマンドアクティブ化の例を示します。

オブジェクトの暗黙的なアクティブ化

POA が ImplicitActivationPolicy::IMPLICIT_ACTIVATION, IdAssignmentPolicy::SYSTEM_ID，および ServantRetentionPolicy::RETAIN で作成済みの場合，サーバントは一定のオペレーションで暗黙的にアクティブ化できます。暗黙的なアクティブ化は次のように実行します。

- POA::servant_to_reference メンバー関数
- POA::servant_to_id メンバー関数
- _this() サーバントメンバー関数

POA に IdUniquenessPolicy::UNIQUE_ID が設定されている場合は，アクティブでないサーバントで上記のいずれかのオペレーションが実行されると，暗黙的なアクティブ化が行われます。

POA に IdUniquenessPolicy::MULTIPLE_ID が設定されている場合は，サーバントがアクティブになっていても，servant_to_reference オペレーションと servant_to_id オペレーションが常に暗黙的アクティブ化を実行します。

デフォルトサーバントによるアクティブ化

RequestProcessing::USE_DEFAULT_SERVANT ポリシーを使用すると，オブジェクト ID に関係なく，POA は常に同じサーバントを呼び出すようになります。この方法は，各オブジェクトに関連付けられているデータがほとんどない場合に便利です。

次に，同じサーバントですべてのオブジェクトをアクティブ化する例を示します。

```
int main(int argc, char* const* argv) {
   try {
      // ORB を初期化します。
      CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
      PortableServer::Current_var cur = PortableServer::Current::_instance();
      // ルート POA へのリファレンスを取得します。
      CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
      PortableServer::POA_var rootPOA = PortableServer::POA::_narrow(obj);

      CORBA::PolicyList policies;
      policies.length(3);
      // 永続的 POA のポリシーを作成します。
      policies[(CORBA::ULong)0] =
            rootPOA->create_lifespan_policy(PortableServer::PERSISTENT);
      policies[(CORBA::ULong)1] =
   rootPOA>create_request_processing_policy(PortableServer::USE_DEFAULT_SERVANT);
      policies[(CORBA::ULong)2] =
            rootPOA->create_id_uniqueness_policy(PortableServer::MULTIPLE_ID);
      // 適切なポリシーで myPOA を作成します。
      PortableServer::POAManager_var rootManager = rootPOA->the_POAManager();
      PortableServer::POA_var myPOA =
            rootPOA->create_POA("bank_default_servant_poa", rootManager,policies);
      // デフォルトサーバントを設定します。
```

```
        AccountManagerImpl managerServant(cur);
        myPOA->set_servant( &managerServant );
        // POA マネージャをアクティブ化します。
        rootManager->activate();

        // 当座と貯金のために，2 つのリファレンスを生成します。
        // ここではサーバントを作成するのではなく，
        // サーバントの裏付けのない参照を生成しているだけ
        // であることに注意してください
        PortableServer::ObjectId_var an_oid =
              PortableServer::string_to_ObjectId("CheckingAccountManager");
        CORBA::Object_var cref = myPOA->create_reference_with_id(an_oid.in(),
              "IDL:Bank/AccountManager:1.0");
        an_oid = PortableServer::string_to_ObjectId("SavingsAccountManager");
        CORBA::Object_var sref = myPOA->create_reference_with_id(an_oid.in(),
              "IDL:Bank/AccountManager:1.0");
        // チェック用のリファレンスを書き出します。
        CORBA::String_var string_ref = orb->object_to_string(cref.in());
        ofstream crefFile("cref.dat");
        crefFile << string_ref << endl;
        crefFile.close();
        // 貯金用のリファレンスを書き出します。
        string_ref = orb->object_to_string(sref.in());
        ofstream srefFile("sref.dat");
        srefFile << string_ref << endl;
        srefFile.close();
        cout << "Bank Manager is ready" << endl;

 // 着信要求を待機します。

orb->run();
    }
    catch(const CORBA::Exception& e) {
    cerr << e << endl;
    }
    return 1;
}
```

オブジェクトの非アクティブ化

POA では，アクティブオブジェクトマップから任意のサーバントを削除できます。このような処理は，たとえば，ガベージコレクション方式で行われます。マップから削除されたサーバントは，非アクティブになります。オブジェクトは deactivate_object() で非アクティブ化できます。オブジェクトを非アクティブ化しても，そのオブジェクトが永久に失われるわけではありません。後でいつでも再アクティブ化できます。

これは，オブジェクトを非アクティブ化する例です。

```
// DeActivatorThread
class DeActivatorThread: public VISThread {
   private :
      PortableServer::ObjectId _oid;
      PortableServer::POA_ptr  _poa;
   public :
      virtual ~DeActivatorThread(){}
      // コンストラクタ
      DeActivatorThread(const PortableServer::ObjectId& oid,
            PortableServer::POA_ptr poa ): _oid(oid), _poa(poa) {
         // スレッドを起動します。
               run();
      }
      // begin() コールバックを実装します。
```

```
        void begin() {
           // 15 秒待ちます。
           VISPortable::vsleep(15);
           CORBA::String_var s = PortableServer::ObjectId_to_string (_oid);
           // オブジェクトを非アクティブ化します。
           cout << "¥nDeActivating the object with ID =" << s << endl;
           if ( _poa )
              _poa->deactivate_object( _oid );
        }
  };
  // サーバントアクティベータ
  class AccountManagerActivator : public PortableServer::ServantActivator {
     public:
        virtual PortableServer::Servant incarnate (const
                    PortableServer::ObjectId& oid,
                    PortableServer::POA_ptr poa) {
           CORBA::String_var s = PortableServer::ObjectId_to_string (oid);
           cout << "¥nAccountManagerActivator.incarnate called with ID = " << s <<
           endl;
           PortableServer::Servant servant;
           if ( VISPortable::vstricmp( (char *)s,  "SavingsAccountManager" ) == 0 )
              // CheckingAccountManager サーバントを作成します。
              servant = new SavingsAccountManagerImpl;
           else if ( VISPortable::vstricmp( (char *)s,
                             "CheckingAccountManager")==0)
              // CheckingAccountManager サーバントを作成します。
              servant = new CheckingAccountManagerImpl;
           else
              throw CORBA::OBJECT_NOT_EXIST();
           // ディアクティベータのスレッドを作成します。
           new DeActivatorThread( oid, poa );
           // サーバントを返します。
           return servant;
        }
        virtual void etherealize (const PortableServer::ObjectId& oid,
              PortableServer::POA_ptr adapter,
              PortableServer::Servant servant,
              CORBA::Boolean cleanup_in_progress,
              CORBA::Boolean remaining_activations) {
           // アクティベーション（サーバントに関連付けられた ObjectId）が
           // 残っていない場合は，サーバントを削除します。
           CORBA::String_var s = PortableServer::ObjectId_to_string (oid);
           cout << "¥nAccountManagerActivator.etherealize called with ID = " << s
              << endl;
           if (!remaining_activations)
              delete servant;
        }
  };
```

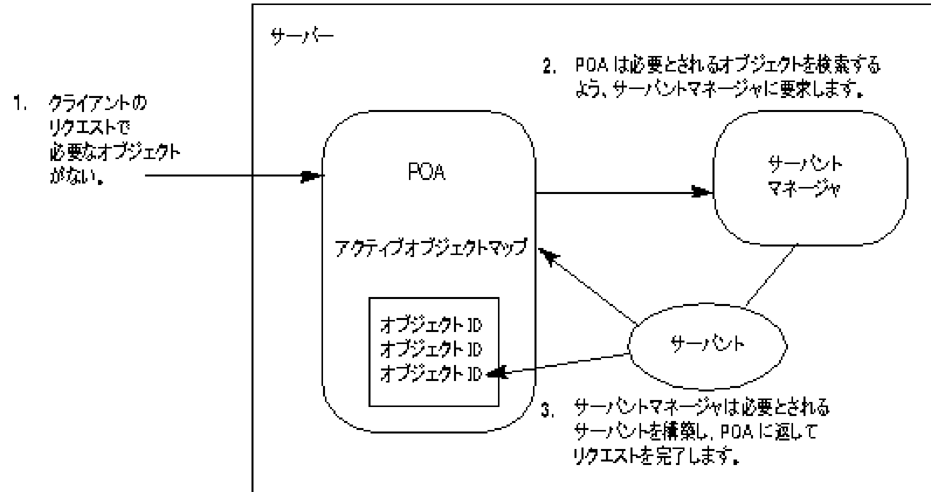
サーバントとサーバントマネージャの使い方

サーバントマネージャでは，サーバントの検索と返し，およびサーバントの非アクティブ化という 2 種類のオペレーションを実行します。サーバントマネージャにより，POA は，アクティブでないオブジェクトへの要求を受信したときに，オブジェクトをアクティブ化できます。サーバントマネージャはオプションです。たとえば，起動時にサーバーがすべてのオブジェクトをロードする場合，サーバントマネージャは不要です。サーバントマネージャは，ForwardRequest 例外で別のオブジェクトに要求を転送するようにクライアントに指示することもできます。

サーバントとは，あるインプリメンテーションのアクティブなインスタンスです。POA は，アクティブなサーバントとそれらのサーバントのオブジェクト ID のマップを管理します。

POA は，クライアント要求を受け取ると，まずこのマップをチェックし，クライアント要求に埋め込まれているオブジェクト ID が記録されているかどうかを確認します。オブジェクト ID が見つかった場合，POA は，要求をサーバントに転送します。オブジェクト ID がマップに見つからなかった場合は，適切なサーバントを見つけてアクティブ化するように，サーバントマネージャに要求します。これは，あくまでも 1 つの例です。実際の処理の流れは，使用している POA ポリシーによって異なります。

図 9.2　サーバントマネージャの動作例



サーバントマネージャには，*ServantActivator* と *ServantLocator* の 2 種類があります。どちらの種類のサーバントマネージャが使用されるかは，現在設定されているポリシーによって決まります。POA ポリシーの詳細については，95 ページの「POA ポリシー」を参照してください。通常，サーバントアクティベータは永続的オブジェクトをアクティブ化し，サーバントロケータは一時的オブジェクトをアクティブ化します。

サーバントマネージャを使用するには，サーバントマネージャの種類を定義するポリシー（サーバントアクティベータの場合は `ServantRetentionPolicy::RETAIN`，サーバントロケータの場合は `ServantRetentionPolicy::NON_RETAIN`）とともに `RequestProcessingPolicy::USE_SERVANT_MANAGER` を設定する必要があります。

ServantActivators

ServantActivator は，`ServantRetentionPolicy::RETAIN` および `RequestProcessingPolicy::USE_SERVANT_MANAGER` が設定されている場合に使用されます。

このサーバントマネージャによってアクティブ化されたサーバントは，アクティブオブジェクトマップに記録されます。

サーバントアクティベータで要求を処理している間に，次の動作が実行されます。

1. クライアント要求が受信されます。クライアント要求は，POA 名やオブジェクト ID などを保持しています。
2. まず，POA はアクティブオブジェクトマップをチェックします。ここでオブジェクト ID が見つかった場合は，処理がサーバントに渡され，クライアントに応答が返されます。
3. アクティブオブジェクトマップにオブジェクト ID が見つからなかった場合，POA は，サーバントマネージャの `incarnate` を呼び出します。`incarnate` は，オブジェクト ID，およびオブジェクトがアクティブ化される POA を渡します。
4. サーバントマネージャが適切なサーバントを探します。
5. アクティブオブジェクトマップにサーバント ID が入力され，クライアントに応答が返されます。

メモ etherealize メソッドインプリメンテーションと incarnate メソッドインプリメンテーションは，ユーザー定義コードです。

サーバントは，後で非アクティブ化されることがあります。これには，deactivate_object オペレーション，POA に関連付けられている POA マネージャの非アクティブ化など，いくつかの場合が考えられます。オブジェクトの非アクティブ化の詳細については，101 ページの「オブジェクトの非アクティブ化」を参照してください。

このサンプルコードは，サーバントアクティベータタイプのサーバントマネージャです。

```
int main(int argc, char* const* argv) {
   try {
      // ORB を初期化します。
      CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
      // ルート POA へのリファレンスを取得します。
      CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");

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

      CORBA::PolicyList policies;
      policies.length(2);
      policies[(CORBA::ULong)0] =
            rootPOA->create_lifespan_policy(PortableServer::PERSISTENT);
      policies[(CORBA::ULong)1] =
            rootPOA->create_request_processing_policy(
                  PortableServer::USE_SERVANT_MANAGER);
      // 適切なポリシーで myPOA を作成します。
      PortableServer::POAManager_var rootManager = rootPOA->the_POAManager();
      PortableServer::POA_var myPOA =
            rootPOA->create_POA("bank_servant_activator_poa", rootManager,
                  policies);
      // サーバントアクティベータを作成します。
      AccountManagerActivator servant_activator_impl;
      // サーバントアクティベータを設定します。
      myPOA->set_servant_manager(&servant_activator_impl);
      // 当座と貯金のために，2 つのリファレンスを生成します。
      // ここではサーバントを作成するのではなく，
      // サーバントの裏付けのない参照を生成しているだけ
      // であることに注意してください
      PortableServer::ObjectId_var an_oid =
            PortableServer::string_to_ObjectId("CheckingAccountManager");
      CORBA::Object_var cref = myPOA->create_reference_with_id(an_oid.in(),
            "IDL:Bank/AccountManager:1.0");
      an_oid = PortableServer::string_to_ObjectId("SavingsAccountManager");
      CORBA::Object_var sref = myPOA->create_reference_with_id(an_oid.in(),
            "IDL:Bank/AccountManager:1.0");
      // POA マネージャをアクティブ化します。
      rootManager->activate();

      // チェック用のリファレンスを書き出します。
      CORBA::String_var string_ref = orb->object_to_string(cref.in());
      ofstream crefFile("cref.dat");
      crefFile << string_ref << endl;
      crefFile.close();
      // 貯金用のリファレンスを書き出します。
      string_ref = orb->object_to_string(sref.in());
      ofstream srefFile("sref.dat");
      srefFile << string_ref << endl;
      srefFile.close();
      // 要求の着信を待機します。
      cout << " BankManager Server is ready" << endl;
      orb->run();
   }
   catch(const CORBA::Exception& e) {
```

```
      cerr << e << endl;
    }
    return 1;
  }
```

次に，このサーバントアクティベータの例でのサーバントマネージャを示します。

```
// サーバントアクティベータ
class AccountManagerActivator : public PortableServer::ServantActivator {

  public:
    virtual PortableServer::Servant incarnate (const
        PortableServer::ObjectId& oid,
        PortableServer::POA_ptr poa) {
      CORBA::String_var s = PortableServer::ObjectId_to_string (oid);
      cout << "¥nAccountManagerActivator.incarnate called with ID = " << s <<
    endl;
      PortableServer::Servant servant;
    if ( VISPortable::vstricmp( (char *)s,  "SavingsAccountManager" ) == 0 )
      // CheckingAccountManager サーバントを作成します。
      servant = new SavingsAccountManagerImpl;
    else if ( VISPortable::vstricmp( (char *)s,  "CheckingAccountManager" ) ==
          0 )
      // CheckingAccountManager サーバントを作成します。
      servant = new CheckingAccountManagerImpl;
    else
      throw CORBA::OBJECT_NOT_EXIST();
    // ディアクティベータのスレッドを作成します。
    new DeActivatorThread( oid, poa );
    // サーバントを返します。
    return servant;
  }
  virtual void etherealize   (const PortableServer::ObjectId& oid,
      PortableServer::POA_ptr adapter,
      PortableServer::Servant servant,
      CORBA::Boolean cleanup_in_progress,
      CORBA::Boolean remaining_activations) {
    // アクティベーション（サーバントに関連付けられた ObjectId）が
    // 残っていない場合は，サーバントを削除します。
    CORBA::String_var s = PortableServer::ObjectId_to_string (oid);
    cout << "¥nAccountManagerActivator.etherealize called with ID = " << s <<
  endl;
    if (!remaining_activations)
      delete servant;
  }
};
```

ServantLocators

一般に，POA のアクティブオブジェクトマップのサイズはかなり大きくなり，メモリに負担がかかります。メモリ消費を削減するため，POA の作成時に RequestProcessingPolicy::USE_SERVANT_MANAGER と ServantRetentionPolicy::.NON_RETAIN を使用します。この場合，サーバントとオブジェクトの関連付けは，アクティブオブジェクトマップに保存されません。関連付けが保存されていないので，要求があるたびに ServantLocator サーバントマネージャが呼び出されます。

サーバントロケータを使って要求を処理している間に，次の動作が実行されます。

1. クライアント要求が受信されます。クライアント要求は，POA 名とオブジェクト ID を保持しています。

2 ServantRetentionPolicy::NON_RETAIN を使用しているので，POA は，アクティブオブジェクトマップからオブジェクト ID を検索しません。

3 POA は，サーバントマネージャで preinvoke を呼び出します。preinvoke は，オブジェクト ID，オブジェクトを起動する POA，2，3 その他パラメータを渡します。

4 サーバントロケータが適切なサーバントを探します。

5 サーバントで処理が行われ，クライアントに応答が返されます。

6 POA は，サーバントマネージャで postinvoke を呼び出します。

メモ preinvoke メソッドと postinvoke メソッドは，ユーザー定義コードです。

これは，サーバントロケータタイプのサーバントマネージャを使ったサーバーコードの例です。

```
int main(int argc, char* const* argv) {
   try {
      // ORB を初期化します。
      CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
      // ルート POA へのリファレンスを取得します。
      CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
      PortableServer::POA_var rootPOA = PortableServer::POA::_narrow(obj);

      CORBA::PolicyList policies;
      policies.length(3);
      // PERSISTENT 存続期間ポリシーで子 POA を作成します
      // NON_RETAIN 管理ポリシー（アクティブオブジェクトマップがない）
      // を持つサーバントマネージャを使用するため，POA が
      // サーバントロケータを使用します
      policies[(CORBA::ULong)0] =
            rootPOA->create_lifespan_policy(PortableServer::PERSISTENT);
      policies[(CORBA::ULong)1] =
            rootPOA->create_servant_retention_policy(PortableServer::NON_RETAIN);
      policies[(CORBA::ULong)2] =
   rootPOA->create_request_processing_policy(PortableServer::USE_SERVANT_MANAGER);
      PortableServer::POAManager_var rootManager = rootPOA->the_POAManager();
      PortableServer::POA_var myPOA =
            rootPOA->create_POA("bank_servant_locator_poa",rootManager,policies);
      // サーバントロケータを作成します。
      AccountManagerLocator servant_locator_impl;
      myPOA->set_servant_manager(&servant_locator_impl);
      // 当座と貯金のために，2 つのリファレンスを生成します。
      // ここではサーバントを作成するのではなく，
      // サーバントの裏付けのない参照を生成しているだけ
      // であることに注意してください
      PortableServer::ObjectId_var an_oid =
            PortableServer::string_to_ObjectId("CheckingAccountManager");
      CORBA::Object_var cref = myPOA->create_reference_with_id(an_oid.in(),
            "IDL:Bank/AccountManager:1.0");
      an_oid = PortableServer::string_to_ObjectId("SavingsAccountManager");
      CORBA::Object_var sref = myPOA->create_reference_with_id(an_oid.in(),
            "IDL:Bank/AccountManager:1.0");
      // POA マネージャをアクティブ化します。
      rootManager->activate();

      // チェック用のリファレンスを書き出します。
      CORBA::String_var string_ref = orb->object_to_string(cref.in());
      ofstream crefFile("cref.dat");
      crefFile << string_ref << endl;
      crefFile.close();
      // 貯金用のリファレンスを書き出します。
      string_ref = orb->object_to_string(sref.in());
      ofstream srefFile("sref.dat");
```

```
      srefFile << string_ref << endl;
      srefFile.close();
      // 着信要求を待機します。
      cout << "Bank Manager is ready" << endl;
      orb->run();
   }
   catch(const CORBA::Exception& e) {
      cerr << e << endl;
   }
   return 1;
}
```

次に，この例でのサーバントマネージャを示します。

```
// サーバントロケータ
class AccountManagerLocator : public PortableServer::ServantLocator {
   public:
      AccountManagerLocator (){}
      // preinvoke は ServantActivators の incarnate メソッドと似ていますが，
      // 要求が着信するたびに呼び出されます。これに対して，incarnate() は
      // POA がアクティブオブジェクトマップにサーバントを発見できないときにだけ呼び出さ
れます。
      virtual PortableServer::Servant preinvoke (const PortableServer::ObjectId& oid,
            PortableServer::POA_ptr adapter,
            const char* operation,
            PortableServer::ServantLocator::Cookie& the_cookie) {
         CORBA::String_var s = PortableServer::ObjectId_to_string (oid);
         cout << "¥nAccountManagerLocator.preinvoke called with ID = " << s << endl;
         PortableServer::Servant servant;
         if ( VISPortable::vstricmp( (char *)s,  "SavingsAccountManager" ) == 0 )
            // CheckingAccountManager サーバントを作成します。
            servant = new SavingsAccountManagerImpl;
         else if ( VISPortable::vstricmp( (char *)s,  "CheckingAccountManager" ) == 0 )
         // CheckingAccountManager サーバントを作成します。
            servant = new CheckingAccountManagerImpl;
         else
            throw CORBA::OBJECT_NOT_EXIST();
         // サーバントアクティベータとは異なり，ここではオブジェクトを明示的に非アクティブ化
         // するスレッドを起動しません。これは，POA 自身が，要求の完了後に postinvoke
         // を呼び出すからです。サーバントアクティベータでは，poa->de_activateobject を
         // 呼び出してオブジェクトが非アクティブ化されたか，POA 自身が
         // 破棄された場合にだけ，POA は etherealize() を呼び出します
         // サーバントを返します。
         return servant;
      }
   virtual void postinvoke (const PortableServer::ObjectId& oid,
         PortableServer::POA_ptr adapter,
         const char* operation,
         PortableServer::ServantLocator::Cookie the_cookie,
         PortableServer::Servant the_servant) {
      CORBA::String_var s = PortableServer::ObjectId_to_string (oid);
      cout << "¥nAccountManagerLocator.postinvoke called with ID = " << s <<
   endl;
      delete the_servant;
   }
};
```

POA マネージャによる POA の管理

POA マネージャは，POA の状態（要求をキューに入れるか，破棄するか）を制御します。また，POA を非アクティブ化することもできます。POA は，それぞれ 1 つの POA マ

ネージャオブジェクトに関連付けられており，POA マネージャは，1 つ以上の POA を制御できます。

POA マネージャは，POA の作成時に POA に関連付けられます。POA マネージャを使用するか，null を指定して，新しい POA マネージャを作成するかはユーザーが選択します。

次に，POA とその POA マネージャを指定する例を示します。

```
PortableServer::POAManager_var rootManager = rootPOA->the_POAManager();
PortableServer::POA_var myPOA =
      rootPOA->create_POA("bank_servant_locator_poa", rootManager, policies);
PortableServer::POA_var myPOA = rootPOA->create_POA(
"bank_servant_locator_poa",null,policies );
```

関連付けられている POA がすべて破棄されると，POA マネージャも「破棄」されます。

POA マネージャには，次の 4 種類の状態があります。

- 停止
- アクティブ
- 破棄
- 非アクティブ

これらの状態によって POA の状態も決まります。これらの状態の詳細については，以下の節で説明します。

現在の状態の取得

POA マネージャの現在の状態を取得するには，次の構文を使用してください。

```
enum State{HOLDING, ACTIVE, DISCARDING, INACTIVE};
State get_state();
```

停止状態

POA マネージャは，作成時にデフォルトで停止状態になります。POA マネージャが停止状態の場合，POA は，着信した要求をすべてキューに入れます。

POA マネージャが停止状態の場合は，アダプタアクティベータを必要とする要求もキューに入ります。

POA マネージャを停止状態にするには，次の構文を使用してください。

```
void hold_requests (in boolean wait_for_completion)
   raises (AdapterInactive);
```

wait_for_completion は Boolean です。FALSE の場合，このオペレーションは POA マネージャを停止状態にし，すぐに戻ります。TRUE の場合，このオペレーションは，状態の変更前に開始したすべての要求が完了するか，POA マネージャが停止以外の状態に変化しないと戻りません。AdapterInactive は，このオペレーションを呼び出した時点で POA マネージャがすでに非アクティブ状態の場合に生成される例外です。

メモ　非アクティブ状態の POA マネージャは，停止状態に変更できません。

キューに入っており，まだ起動されていない要求は，停止状態の間，そのままキューの中に保持されます。

アクティブ状態

POA マネージャがアクティブ状態の場合，関連する POA は，要求を処理します。

POA マネージャをアクティブ状態にするには，次の構文を使用してください。

```
void activate()
   raises (AdapterInactive);
```

AdapterInactive は，このオペレーションを呼び出した時点で POA マネージャがすでに非アクティブ状態の場合に生成される例外です。

メモ　非アクティブ状態の POA マネージャは，アクティブ状態に変更できません。

破棄状態

POA マネージャが破棄状態の場合，関連する POA は，まだ起動されていない要求をすべて破棄します。このとき，POA に登録されているアダプタアクティベータは呼び出されません。POA が受け取る要求の数が多すぎる場合は，この状態が便利です。その場合は，破棄された要求を再送信するようにクライアントに通知する必要があります。POA が受け取る要求の数が多すぎるかどうかを判定するための機能は用意されていません。必要に応じて，ユーザー自身でスレッドの監視機能を設定してください。

POA マネージャを破棄状態にするには，次の構文を使用してください。

```
void discard_requests(in boolean wait_for_completion)
   raises (AdapterInactive);
```

wait_for_completion オプションはブール値です。FALSE の場合，このオペレーションは POA マネージャを停止状態にし，すぐに戻ります。TRUE の場合，このオペレーションは，状態の変更前に開始したすべてのリクエストが完了するか, POA マネージャが破棄以外の状態に変化しないと戻りません。AdapterInactive は，このオペレーションを呼び出した時点で POA マネージャがすでに非アクティブ状態の場合に生成される例外です。

メモ　非アクティブ状態の POA マネージャは，破棄状態に変更できません。

非アクティブ状態

POA マネージャが非アクティブ状態の場合，関連する POA は，着信した要求を受け付けません。この状態は，関連する POA をシャットダウンするときに使用します。

メモ　非アクティブ状態の POA マネージャは，ほかの状態に変更できません。

POA マネージャを非アクティブ状態にするには，次の構文を使用してください。

```
void deactivate (in boolean etherealize_objects, in boolean wait_for_completion)
   raises (AdapterInactive);
```

状態の変更後に，etherealize_objects が TRUE の場合，Servant RetentionPolicy::RETAIN と RequestProcessingPolicy::USE_SERVANT_MANAGER が設定されているすべての関連する POA は，すべてのアクティブオブジェクトについてサーバントマネージャの etherealize を呼び出します。etherealize_objects が FALSE の場合，etherealize は呼び出されません。wait_for_completion オプションはブール値です。FALSE の場合，このオペレーションは，状態を非アクティブ化してすぐに戻ります。TRUE の場合，このオペレーションは，状態の変更前に開始されたすべての要求が完了するか，すべての関連する POA（ServantRetentionPolicy::RETAIN と RequestProcessingPolicy::USE_SERVANT_MANAGER が設定）で etherealize が呼び出されるまで戻りません。AdapterInactive は，このオペレーションを呼び出した時点で POA マネージャがすでに非アクティブ状態の場合に生成される例外です。

リスナーとディスパッチャ：サーバーエンジン，サーバー接続マネージャ，およびそれらのプロパティ

メモ　POA には，これまで BOA によってサポートされていたリスナー機能とディスパッチャ機能に関するポリシーがありません。これらの機能を提供するため，VisiBroker 固有のポリシー（ServerEnginePolicy）を使用できます。

Visibroker では，Visibroker サーバーのエンドポイントを定義および調整するために，たいへん柔軟性のあるメカニズムが提供されています。この場合のエンドポイントとは，クライアントがサーバーと通信するための通信チャネルの接続先です。サーバーエンジンは，設定可能なプロパティのセットとして提供される接続エンドポイントのための仮想抽象コンポーネントです。

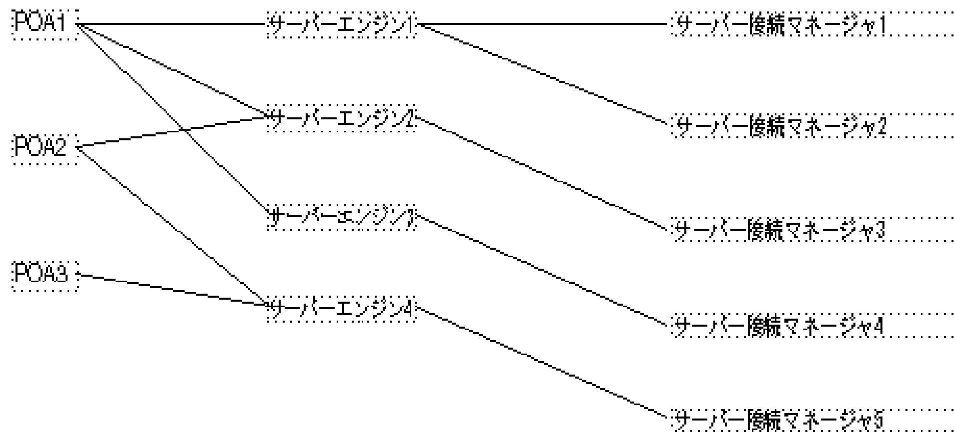
抽象 ServerEngine は，次の項目を制御できます。

- 接続リソースの種類
- 接続管理
- スレッドモデルと要求のディスパッチ

サーバーエンジンと POA

Visibroker の POA は，ServerEngine と多対多の関係を持つことができます。1 つの POA を複数の ServerEngine に，また 1 つの ServerEngine を複数の POA に関連付けることができます。そのため，POA（および POA 上の CORBA オブジェクト）は，複数の通信チャネルをサポートできます。

図 9.3　サーバーエンジンの概要



最も単純な例は，POA がそれぞれに固有のサーバーエンジンを 1 つだけ持つ場合です。この場合，各 POA への要求は，それぞれ異なるポートで受信されます。また，1 つの POA が複数のサーバーエンジンを持つこともできます。この場合は，その POA が複数の入力ポートから着信する要求をサポートします。

POA は，サーバーエンジンを共有できます。サーバーエンジンが共有されている場合は，複数の POA が同じポートを監視します。複数の POA への要求が同じポートに着信しても，それらの要求は，埋め込まれている POA 名を利用して正しくディスパッチされます。このような状況は，デフォルトのサーバーエンジンを使用し，新しいサーバーエンジンを指定しないで複数の POA を作成する場合などに起こります。

サーバーエンジンは名前によって識別され，その名前が最初に組み込まれるときに定義されます。Visibroker では，デフォルトで次の 3 つのサーバーエンジン名が定義されています。

- iiop_tp：スレッドプールディスパッチャを使用した TCP トランスポート
- iiop_ts：セッションごとスレッドディスパッチャを使用した TCP トランスポート
- iiop_tm：メインスレッドディスパッチャを使用した TCP トランスポート

さらに，VisiBroker for C++ は次のサーバーエンジンを定義します。

- liop_tp：スレッドプールディスパッチャを使用した TCP トランスポート
- liop_ts：セッションごとのスレッドディスパッチャを使用した TCP トランスポート

- liop_tm: Local ICP：メインスレッドディスパッチャを使用した TCP トランスポート

BOA の下位互換性を保持するために，さらに 2 つのサーバーエンジン boa_tp と boa_ts を使用できます。

POA とサーバーエンジンの関連付け

POA に関連付けられているデフォルトのサーバーエンジンを変更するには，プロパティ vbroker.se.default を使用します。たとえば，次のように設定します。

```
vbroker.se.default=MySE
```

これは，MySE という名前の新しいサーバーエンジンを定義しています。ルート POA と，作成されたすべての子 POA は，デフォルトでこのサーバーエンジンに関連付けられます。

また，SERVER_ENGINE_POLICY_TYPE POA ポリシーを使用すると，POA を特定のサーバーエンジンに明示的に関連付けることができます。次に例を示します。

```
// ServerEngine ポリシー値を作成します。
CORBA::Any_var se(new CORBA::Any);
CORBA::StringSequence_var engines =
  new CORBA::StringSequence(1UL);
engines->length(1UL);
engines[(CORBA::ULong)0] = CORBA::string_dup("MySE");
se <<= engines;

// POA のポリシーを作成します。
CORBA::PolicyList_var policies =
  new CORBA::PolicyList(2UL);
policies->length(2UL);
policies[(CORBA::ULong)0] =
  orb->create_policy(
  PortableServerExt::SERVER_ENGINE_POLICY_TYPE,
  se);
policies[(CORBA::ULong)1] =
  rootPOA->create_lifespan_policy(
  PortableServer::PERSISTENT);

// ポリシー付きで POA を作成します。
PortableServer::POA_var myPOA = rootPOA->create_POA(
  "bank_se_policy_poa", manager,
  policies);
```

POA は IOR テンプレートを持ち，そのプロファイルは，POA に関連付けられているサーバーエンジンから取得されます。

サーバーエンジンポリシーを指定しないと，POA は，サーバーエンジン名が iiop_tp であるとみなして，次のデフォルト値を使用します。

```
vbroker.se.iiop_tp.host=null
vbroker.se.iiop_tp.proxyHost=null
vbroker.se.iiop_tp.scms=iiop_tp
vbroker.se.liop_tp.host=null
vbroker.se.liop_tp.proxyHost=null
vbroker.se.liop_tp.scms=liop_tp
```

デフォルトのサーバーエンジンポリシーを変更するには，vbroker.se.default プロパティを使って新しいサーバーエンジンポリシー名を入力し，新しいサーバーエンジンのすべての要素に値を定義してください。次に例を示します。

```
vbroker.se.default=abc,def
vbroker.se.abc.host=cob
vbroker.se.abc.proxyHost=null
vbroker.se.abc.scms=cobscm1,cobscm2
vbroker.se.def.host=gob
vbroker.se.def.proxyHost=null
vbroker.se.def.scms=gobscm1
```

サーバーエンジンのエンドポイントのホストの定義

サーバーエンジンは接続のエンドポイントの定義に使用されるため，エンドポイントのホストを指定するために次のプロパティが提供されています。

- vbroker.se.<se-name>.host=<host-URL>: vbroker.se.mySE.host=host.borland.com（例）
- vbroker.se.<se-name>.proxyHost=<proxy-host-URL-or-IP-address>: vbroker.se.mySE.proxyHost=proxy.borland.com（例）

proxyHost プロパティの値には，IP アドレスを指定することもできます。その場合は，IOR 内のデフォルトホスト名がその IP アドレスに置き換えられます。

ServerEngine の抽象エンドポイントは，サーバー接続マネージャ（SCM）と呼ばれる設定可能な一連のエンティティによってさらに詳細に設定できます。ServerEngine は，複数の SCM を持つことができます。SCM は，複数の ServerEngine で共有できません。SCM も名前によって識別され，ServerEngine に対して次のように定義されます。

```
vbroker.se.<se-name>.scms=<SCM-name>[,<SCM-name>,...]
```

メモ　iiop_tp および liop_tp の Server Engine には，それぞれ iiop_tp および liop_tp という SCM が指定されています。

サーバー接続マネージャ

サーバー接続マネージャ（SCM）は，エンドポイントの設定可能なコンポーネントを定義します。SCM は，接続リソースを管理し，要求を監視し，関連付けられている POA に要求をディスパッチします。これらの機能を実行するため，プロパティグループを介して定義される次の 3 つの論理エンティティが SCM によって提供されます。

- マネージャ
- リスナー
- ディスパッチャ

各 SCM は，マネージャ，リスナー，ディスパッチャを 1 つずつ持ちます。この 3 つがすべて定義されている場合に，単一のエンドポイント定義が形成され，クライアントはサーバーと通信できるようになります。

マネージャ

マネージャは，接続リソースの設定可能部分を定義する一連のプロパティです。VisiBroker は，Socket 型のマネージャを提供します。

さらに，VisiBroker for C++ は，Local 型の別のマネージャを定義します。Local 型は Local IPC 接続に対応し，Socket マネージャ型は TCP 接続に対応します。Local または Socket を選択するには，次のプロパティを設定します。

```
vbroker.se.<se-name>.scm.<scm-name>.manager.type=Local|Socket
```

サーバーのエンドポイントで受け入れることができる最大同時接続数を指定するには，connectionMax プロパティを使用します。

```
vbroker.se.<se-name>.scm.<scm-name>.manager.connectionMax=<integer>
```

connectionMax を 0 に設定すると，接続数に制限がないことを示します。これはデフォルトの設定です。

最大アイドル時間を指定するには，connectionMaxIdle プロパティを使用します。

```
vbroker.se.<se-name>.scm.<scm-name>.manager.connectionMaxIdle=<seconds>
```

connectionMaxIdle を 0 に設定すると，タイムアウトがないことを示します。これはデフォルトの設定です。

マネージャがアイドル状態の接続を回収するためのガベージコレクション時間を指定することもできます。connectionMaxIdle 時間の経過後も，ガベージコレクションによって回収

されるまでの間，接続はアイドル状態のままです。garbageCollectTimer プロパティを使用して，ガベージコレクションの周期を秒単位で指定できます。

```
vbroker.se.<se-name>.scm.<scm-name>.manager.garbageCollectTimer=<seconds>
```

0 を指定すると，接続はガベージコレクションによって回収されません。

リスナー

リスナーは，SCM がメッセージを監視する方法を決定する SCM コンポーネントです。マネージャと同様に，リスナーも一連のプロパティで構成されます。VisiBroker では，TCP 接続に対して IIOP リスナーが定義されています。

さらに，VisiBroker for C++ は，ローカル IPC 接続用に LIOP リスナーを定義します。次のプロパティで，どちらのタイプのリスナーを使用するかを指定します。

```
vbroker.se.<se-name>.scm.<scm-name>.listener.type=LIOP|IIOP
```

リスナーは，実際の基底のトランスポートメカニズムに密接しているため，異なるリスナータイプ間ではリスナーのプロパティに可搬性がありません。次に定義されるように，各リスナータイプが独自のプロパティセットを持ちます。

LIOP リスナーのプロパティ

共有メモリのローカル IPC を使用しているシステムの場合は，shmSize プロパティを使用して，共有メモリサイズをバイト単位で制御します。

```
vbroker.se.<se-name>.scm.<scm-name>.listener.shmSize=<bytes>
```

共有メモリマップファイルを，このユーザーだけがアクセスできるディレクトリに隠す必要がある場合は，次の論理プロパティを設定する必要があります。

```
vbroker.se.<se-name>.scm.<scm-name>.listener.userConstrained=true|false
```

IIOP リスナーのプロパティ

IIOP リスナーでは，ホストと組み合わせて，ポートと（必要に応じて）プロキシポートを定義する必要があります。これらは，port プロパティと proxyPort プロパティを使用して，次のように設定されます。

```
vbroker.se.<se-name>.scm.<scm-name>.listener.port=<port>
vbroker.se.<se-name>.scm.<scm-name>.listener.proxyPort=<proxy-port>
```

メモ　port プロパティを設定しない場合，または 0 に設定した場合は，ポートが無作為に選択されます。proxyPort プロパティの値を 0 にすると，listener.port プロパティによって定義されるか，システムによって無作為に選択された実際のポートが IOR に含まれます。実際のポートを宣言する必要がない場合は，プロキシポートを正数（0 以外）に設定してください。

送信／受信のバッファサイズ，ソケット遅延時間，アクティブでないソケットを存続させるかどうかなど，標準の TCP ソケットオプションを定義するためのプロパティ設定もサポートされています。これらのメカニズムのために，次のプロパティが提供されています。

```
vbroker.se.<se-name>.scm.<scm-name>.listener.rcvBuffSize=<bytes>
vbroker.se.<se-name>.scm.<scm-name>.listener.sendBuffSize=<bytes>
vbroker.se.<se-name>.scm.<scm-name>.listener.socketLinger=<seconds>
vbroker.se.<se-name>.scm.<scm-name>.connection.keepAlive=true|false
```

何らかの理由で，TCP ソケットのプロパティとして単にシステムのデフォルト値を使用する場合は，該当するプロパティの値を 0 に設定します。

ディスパッチャ

ディスパッチャは，SCM がスレッドに要求をディスパッチする方法を決定する一連のプロパティを定義します。ThreadPool，ThreadSession，MainThread の 3 つのタイプのディスパッチャが提供されています。ディスパッチャのタイプは，次のように type プロパティを使って設定します。

```
vbroker.se.<se-name>.scm.<scm-name>.dispatcher.type=ThreadPool|ThreadSession|MainThread
```

ディスパッチャタイプが ThreadPool の場合は，SCM を介してさらに詳細な制御が提供されます。ThreadPool は，スレッドプール内に作成できる最小スレッド数と最大スレッド数，およびアイドル状態のスレッドが破棄されるまでの最大時間（秒）を定義します。これらの値は，次のプロパティで制御されます。

```
vbroker.se.<se-name>.scm.<scm-name>.dispatcher.threadMin=<integer>
vbroker.se.<se-name>.scm.<scm-name>.dispatcher.threadMax=<integer>
vbroker.se.<se-name>.scm.<scm-name>.dispatcher.threadMaxIdle=<seconds>
```

ThreadPool ディスパッチャで，「冷却時間」を設定できます。接続の作成時または要求の到着時に，サービスを提供されている GIOP 接続が読み取り可能な場合，スレッドは「ホット」です。冷却時間（秒）が経過すると，スレッドはスレッドプールに戻すことができます。

次のプロパティを使って冷却時間を設定します。

```
vbroker.se.<se-name>.scm.<scm-name>.dispatcher.coolingTime=<seconds>
```

以上のプロパティを使用するタイミング

サーバーエンジンのプロパティの一部は，何度も変更する必要があります。これらのプロパティを変更する方法は，目的に応じて異なります。たとえば，ポート番号を変更する場合は，次の方法があります。

- デフォルトの listener.port プロパティを変更する。
- 新しいサーバーエンジンを作成する。

デフォルト listener.port プロパティの変更が最も簡単ですが，デフォルトサーバーエンジンを使用するすべての POA に影響が出ます。そして，それで問題がない場合と，問題になる場合があります。

特定の POA においてポート番号を変更する場合は，新しいサーバーエンジンを作成し，そのプロパティを定義して，POA の作成時にそのサーバーエンジンを参照する必要があります。前の節では，サーバーエンジンのプロパティを更新する方法を示しました。ここでは，サーバーエンジンのプロパティを定義し，ユーザー定義のサーバーエンジンポリシーを使って POA を作成する方法を示します。次のコードを参照してください。

```
         // 静的初期化
         AccountRegistry AccountManagerImpl::_accounts;
         int main(int argc, char* const* argv)
         {
            try {
            // orb を初期化します。
            CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
            // プロパティマネージャを取得します。戻り値は var 型には
            // 格納されません
            VISPropertyManager_ptr pm = orb->getPropertyManager();
            pm->addProperty("vbroker.se.mySe.host", "");
            pm->addProperty("vbroker.se.mySe.proxyHost", "");
            pm->addProperty("vbroker.se.mySe.scms", "scmlist");
            pm->addProperty("vbroker.se.mySe.scm.scmlist.manager.type", "Socket");
            pm->addProperty("vbroker.se.mySe.scm.scmlist.manager.connectionMax",
100UL);
            pm->addProperty("vbroker.se.mySe.scm.scmlist.manager.connectionMaxIdle",
                  300UL);
            pm->addProperty("vbroker.se.mySe.scm.scmlist.listener.type", "IIOP");
            pm->addProperty("vbroker.se.mySe.scm.scmlist.listener.port", 55000UL);
            pm->addProperty("vbroker.se.mySe.scm.scmlist.listener.proxyPort", 0UL);
            pm->addProperty("vbroker.se.mySe.scm.scmlist.dispatcher.type",
"ThreadPool");
            pm->addProperty("vbroker.se.mySe.scm.scmlist.dispatcher.threadMax", 100UL);
            pm->addProperty("vbroker.se.mySe.scm.scmlist.dispatcher.threadMin", 5UL);
            pm->addProperty("vbroker.se.mySe.scm.scmlist.dispatcher.threadMaxIdle",
                  300UL);
```

```
        // ルート POA へのリファレンスを取得します。
        CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
        PortableServer::POA_var rootPOA = PortableServer::POA::_narrow(obj);
        // Create the policies
        CORBA::Any_var seAny(new CORBA::Any);
        // 1 つのエンジンを指定する場合でも，SERVER_ENGINE_POLICY_TYPE では
        // シーケンスを指定する必要があります。
        CORBA::StringSequence_var engines = new CORBA::StringSequence(1UL);
        engines->length(1UL);
        engines[0UL] = CORBA::string_dup("mySe");
        seAny <<= engines;
        CORBA::PolicyList_var policies = new CORBA::PolicyList(2UL);
        policies->length(2UL);
        policies[0UL] = orb->create_policy(
                          PortableServerExt::SERVER_ENGINE_POLICY_TYPE, seAny);
        policies[1Ul] = rootPOA-
>create_lifespan_policy(PortableServer::PERSISTENT);
        // 独自のポリシーで独自の POA を作成します。
        PortableServer::POAManager_var manager = rootPOA->the_POAManager();
        PortableServer::POA_var myPOA = rootPOA->create_POA(
                          "bank_se_policy_poa", manager, policies);

        // サーバントを作成します。
        AccountManagerImpl* managerServant = new AccountManagerImpl();
        // サーバントをアクティブ化します。
        PortableServer::ObjectId_var oid =
                PortableServer::string_to_ObjectId("BankManager");
        myPOA->activate_object_with_id(oid ,managerServant);
        // リファレンスを取得します。
        CORBA::Object_var ref = myPOA->servant_to_reference(managerServant);
        CORBA::String_var string_ref = orb->object_to_string(ref.in());
        ofstream refFile("ref.dat");
        refFile << string_ref << endl;
        refFile.close();
        // POA マネージャをアクティブ化
        manager->activate();
        // 要求の着信を待ちます。
        cout << "AccountManager Server ready" << endl;
        orb->run();
        }
        catch(const CORBA::Exception& e) {
        cerr << e << endl;
        return (1);
        }
    return (0);
    }
```

アダプタアクティベータ

アダプタアクティベータは，POA に関連付けられており，オンデマンドで子 POA を作成する機能を提供します。これは，find_POA オペレーションの間か，特定の子 POA を指定する要求が受信されたときに行われます。

POA は，アダプタアクティベータを利用して，オンデマンドで子 POA を作成できます。オンデマンドで子 POA が作成されるのは，子 POA（またはその 1 つ）を指定する要求を受信したときの処理過程か，アクティブ化パラメータ値が TRUE の状態で find_POA が呼び出されたときです。実行の開始時に，必要な POA をすべて作成するようなアプリケーションサーバーでは，アダプタアクティベータの使用や提供は不要です。アダプタアクティベータは，要求の処理中に POA を作成する場合にだけ必要です。

POA からアダプタアクティベータへの要求が処理されている間，新しい POA（または子以下の POA）によって管理されるオブジェクトへの要求は，すべてキューに入れられます。このシリアライゼーションにより，新しい POA に要求が配信される前に，アダプタアクティベータは POA の初期化を完了できます。

アダプタアクティベータの使用例については，製品に付属している POA の adaptor_activator サンプルを参照してください。

要求の処理

要求では，ターゲットオブジェクトのオブジェクト ID，およびそのオブジェクトリファレンスを作成した POA を保持します。クライアントが要求を送信すると，まず VisiBroker ORB が適切なサーバーを探すか，必要に応じてサーバーを起動します。次に，そのサーバー内の適切な POA を探します。

VisiBroker ORB は，適切な POA を見つけると，その POA に要求を配信します。この時点で要求がどのように処理されるかは，POA のポリシーおよびオブジェクトのアクティブ化状態によって異なります。オブジェクトのアクティブ化状態については，98 ページの「オブジェクトのアクティブ化」を参照してください。

- POA に ServantRetentionPolicy::RETAIN がある場合は，POA はアクティブオブジェクトマップを参照して，要求のオブジェクト ID に関連付けられたサーバントを探します。サーバントが見つかった場合は，そのサーバントの適切なメソッドを起動します。
- POA に ServantRetentionPolicy::NON_RETAIN または ServantRetentionPolicy::RETAIN があり，適切なサーバントが見つからなかった場合は，次の処理に続きます。
    - POA に RequestProcessingPolicy::USE_DEFAULT_SERVANT がある場合，POA はデフォルトサーバントで適切なメソッドを呼び出します。
    - POA に RequestProcessingPolicy::USE_SERVANT_MANAGER がある場合は，サーバントマネージャの incarnate または preinvoke を呼び出します。
    - POA に RequestProcessingPolicy::USE_OBJECT_MAP_ONLY がある場合，例外が発生します。

サーバントマネージャを起動しても，オブジェクトを具現化できない場合は，サーバントマネージャが ForwardRequest 例外を生成することがあります。

第 10 章

スレッドと接続の管理

ここでは，クライアントプログラムやオブジェクトインプリメンテーションで複数のスレッドを使用する方法について説明します。さらに，VisiBroker スレッドおよび接続のモデルについて理解を深めます。

スレッドの使い方

スレッドは，プロセス内の一連の制御の流れのことで，軽量プロセスとも呼ばれます。スレッドでは，ほかのスレッドと基本要素を共有して，オーバーヘッドを減らします。また，軽量なので，1 つのプロセス内に多数のスレッドが共存できます。

マルチスレッドでは，単一のアプリケーション内で同期が可能になるので，パフォーマンスが向上します。また，複数の独立した計算を同時に行うスレッドを使用すれば，効率的なアプリケーションを構成できます。たとえば，複数のファイル操作やネットワークオペレーションを同時に実行しながら，多くのユーザーと対話するようなデータベースシステムが考えられます。

ある要求から別の要求へ非同期に移動する 1 つの制御スレッドとしてソフトウェアを記述することもできますが，各要求を独立したシーケンスとして記述し，さまざまなオペレーションの同期インターリーブは基底のシステムで処理した方が，コードはシンプルになります。

マルチスレッドは，次のような場合に有効です。

- ほかの処理には必ずしも依存しない複数の長いオペレーショングループ（ウィンドウの描画，ドキュメントの印刷，マウスクリックに対する反応，スプレッドシートの計算，シグナル処理など）がある場合。
- データがほとんどロックされてない場合。つまり，共有データの量が明らかで，少量の場合。
- タスクを複数の役割に分割できる場合。たとえば，シグナルを処理するスレッドとユーザーインターフェースを処理するスレッドに分割できる場合。

スレッドと接続の管理は，サーバーエンジンと呼ばれるエンティティのスコープ内で行います。複数のデフォルトのサーバーエンジンが VisiBroker によって自動的に作成されます。これには，IIOP 向けや LIOP 向けのスレッドプールエンジンなどが含まれます。アプリケーションにより，VisiBroker サーバー内で追加のサーバーエンジンを使用したり作成することができます。次のディレクトリにあるサンプルを参照してください。

```
<install_dir>/examples/vbe/poa/server_engine_policy/Server.C
```

サーバーエンジンは個別に作成，設定，および使用することができます。サーバーエンジンの作成や設定を行っても，同じサーバー内のほかのサーバーエンジンには影響しません。通常，各サーバーエンジンには，*監視ポイント／ソケット*と呼ばれる 1 つのトランスポートエンドポイントがあります。

サーバーエンジンと POA の関係は多対多です。各サーバーエンジンを複数の POA が使用できます。また，各 POA は複数のサーバーエンジンを使用できます。

サーバーエンジンは，複数のサーバー接続マネージャ（SCM）で構成されます。SCM は，マネージャ，リスナー，およびディスパッチャで構成されます。マネージャ，リスナー，およびディスパッチャのプロパティを設定して，SCM の機能を決定できます。これらのプロパティについては，127 ページの「接続管理プロパティの設定」を参照してください。

リスナースレッド，ディスパッチャスレッド，および作業スレッド

各サーバーエンジンには，リスナースレッドとディスパッチャスレッドがあります。リスナースレッドは次の役割を分担します。

- 新しい接続の受け付け。したがって，リスナースレッドは監視エンドポイントを監視します。
- アイドル状態の GIOP 接続の可読性の監視
- 監視リストの更新
- プロパティ設定に基づくアイドル状態の接続のガベージコレクション

ディスパッチャは，要求の送信先のスレッドを決定します。

各サーバーエンジンは，一定数の作業スレッドを使用して，要求の受け取りと処理を行います。要求は，それぞれ異なる作業スレッドによって処理されます。特定の要求の読み取り，処理（サーバー側インターセプタのインターセプトなど），および応答は，すべて同じスレッドによって処理されます。サーバーエンジンが使用する作業スレッドの数は，次の要素によって異なります。

- スレッドモデル
- 同時処理する要求または接続の数
- プロパティ設定

スレッドポリシー

VisiBroker がサポートする主な 2 つのスレッドモデルは，スレッドプール（要求ごとのスレッドまたは `TPool` とも呼ばれる）とセッションごとのスレッド（接続ごとのスレッドまたは `TSession` とも呼ばれる）です。シングルスレッドモデルとメインスレッドモデルについては，ここでは扱いません。スレッドプールとセッションごとのスレッドのモデルは，次の点で基本的に異なっています。

- 作成される状況
- 同じクライアントから複数の要求を同時に受け取った場合の処理方法
- スレッドを解放するタイミングと方法

デフォルトのスレッドポリシーは，スレッドプールです。セッションごとのスレッドの設定やスレッドプールモデルのプロパティの変更については，125 ページの「ディスパッチのポリシーとプロパティの設定」を参照してください。

スレッドプールポリシー

サーバーがスレッドプールポリシーを使用する場合，クライアント要求を処理するために割り当てることができる最大のスレッド数が定義されます。各クライアント要求ごとに 1 つの作業スレッドが割り当てられますが，この作業スレッドはその要求が存続している間のみ有効です。要求が完了すると，その要求に割り当てられていた作業スレッドは，使用可能なスレッドのプールに入れられ，その後，任意のクライアントから受け取る要求の処理に割り当てられます。

このモデルでは，サーバーオブジェクトへの要求トラフィックの量に基づいてスレッドが割り当てられます。つまり，サーバーに同時に多くの要求を送る非常にアクティブなクライアントは，要求をすばやく実行できるように複数のスレッドによって処理されます。一方，あまりアクティブでないクライアントは，別のクライアントと 1 つのスレッドを共有することになりますが，それでも要求はすぐに処理されます。さらに，スレッドを破棄しないで再利用し，複数の新しい接続に割り当てることができるので，作業スレッドの作成や破棄にかかるオーバーヘッドを削減できます。

VisiBroker は，デフォルトの同時クライアント要求の数に基づき，スレッドプール内のスレッド数を動的に割り当てことにより，システムリソースの消費を抑えます。クライアントがビジーになると，それに応じて新しいスレッドが割り当てられます。アクティブでないスレッドは，VisiBroker によって解放され，現在のクライアントの要求に必要な数だけスレッドを保持します。このように，サーバーでは常に最適な数のスレッドをアクティブ化しておくことができます。

スレッドプールのサイズは，サーバーのアクティビティに応じて変化します。ただし，特定の分散システムのニーズに合わせ，サーバーの実行前または実行中でも，完全に設定が可能です。スレッドプールモデルでは，次の項目を設定できます。

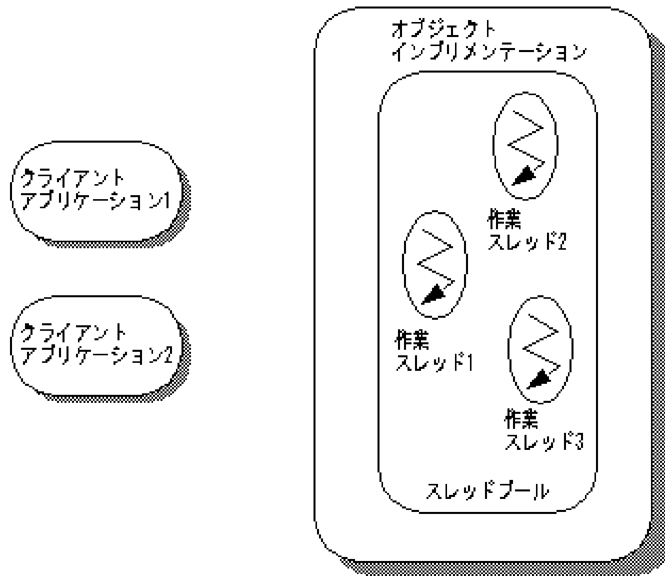
- 最大および最小のスレッド数
- 最大アイドル時間

クライアント要求が受信されるたびに，その要求を処理するためにスレッドプールからスレッドの割り当てが行われます。それが最初のクライアント要求で，プールが空の場合は，スレッドが 1 つ作成されます。同様に，すべてのスレッドがビジーである場合も，新しいスレッドが作成されて，要求を処理します。

サーバーでは，クライアント要求の処理を割り当てることができる最大のスレッド数を定義できます。使用可能なスレッドがプール内になく，すでに最大数のスレッドが作成されている場合は，現在使用されているスレッドが解放されてプールに戻されるまで，要求はブロックされます。

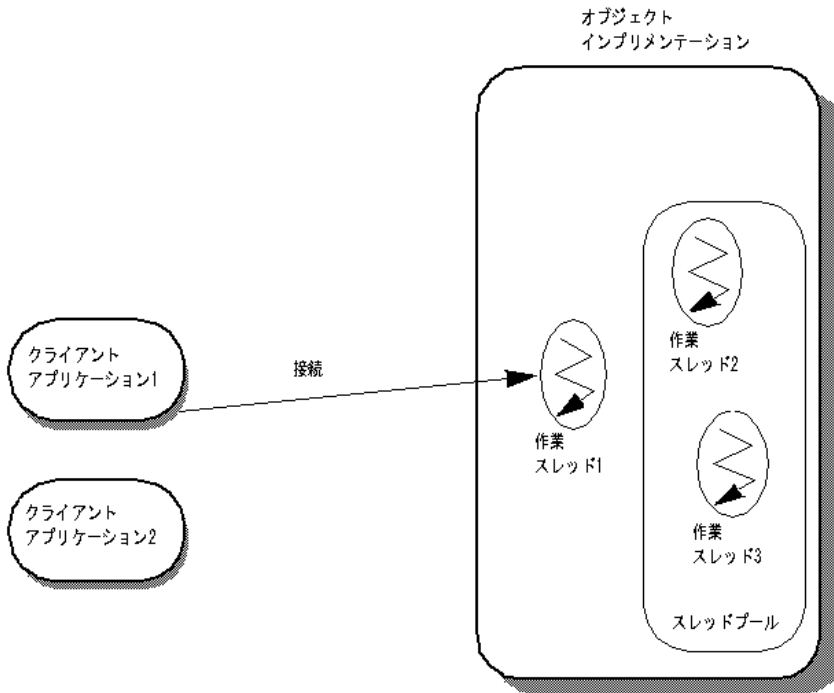
スレッドプールは，デフォルトのスレッドポリシーです。この環境を定義するために必要な設定はありません。スレッドプールのプロパティを設定する場合は，125 ページの「ディスパッチのポリシーとプロパティの設定」を参照してください。

図 10.1　使用可能なスレッドプール



上の図は，スレッドプールポリシーによるオブジェクトインプリメンテーションです。名前からわかるように，このポリシーには，作業スレッドとして使用できるプールがあります。

図 10.2　クライアントアプリケーション＃1による要求の送信

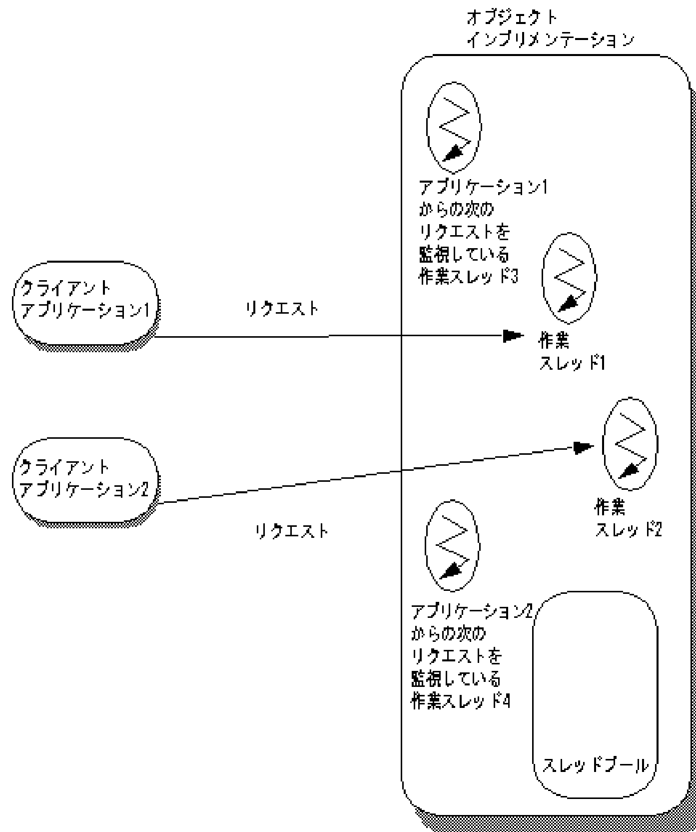


上の図で，クライアントアプリケーション＃1は，オブジェクトインプリメンテーションとの接続を確立し，要求を処理するスレッドが作成されます。スレッドプールでは，クライアントごとに1つずつ接続が確立され，接続ごとに1つのスレッドが作成されます。着信した要求を作業スレッドが受け取ると，その作業スレッドはプールから削除されます。

スレッドプールから除去された作業スレッドは，常時要求を監視しています。要求が着信すると，作業スレッドは要求を読み取り，適切なオブジェクトインプリメンテーションに

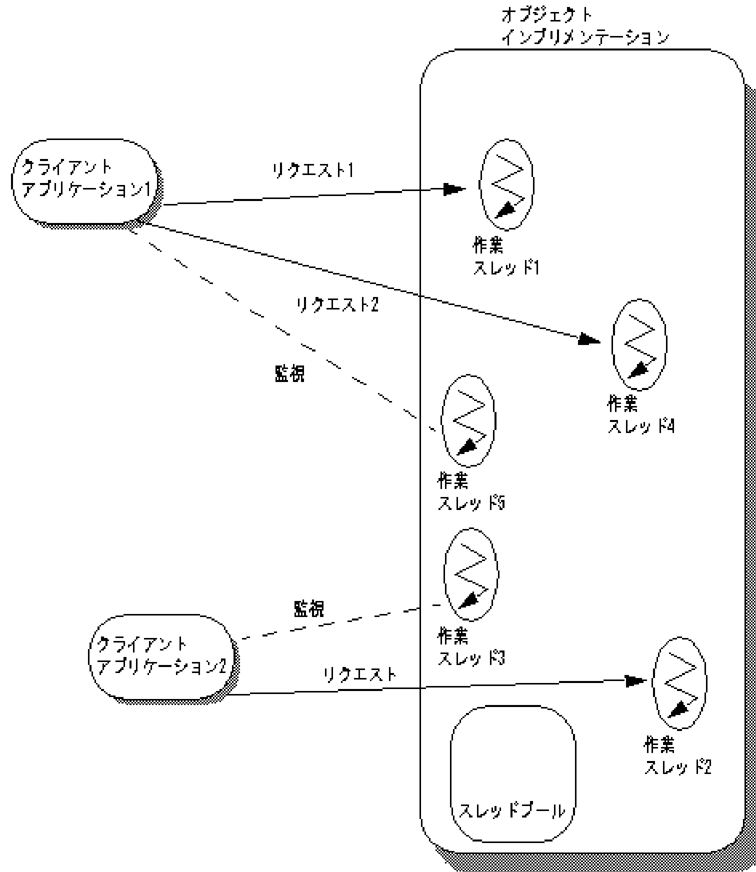
その要求を送ります。作業スレッドが要求をオブジェクトインプリメンテーションに送る前に，その作業スレッドは次の要求を監視する別の作業スレッドを起動します。

図 10.3　クライアントアプリケーション #2 による要求の送信



上の図では，クライアントアプリケーション #2 が接続を確立し，要求を送信します。その結果，2 番めの作業スレッドが作成されます。作業スレッド #3 は，要求の着信を監視している状態です。

図 10.4 クライアントアプリケーション＃1による第 2 の要求の送信

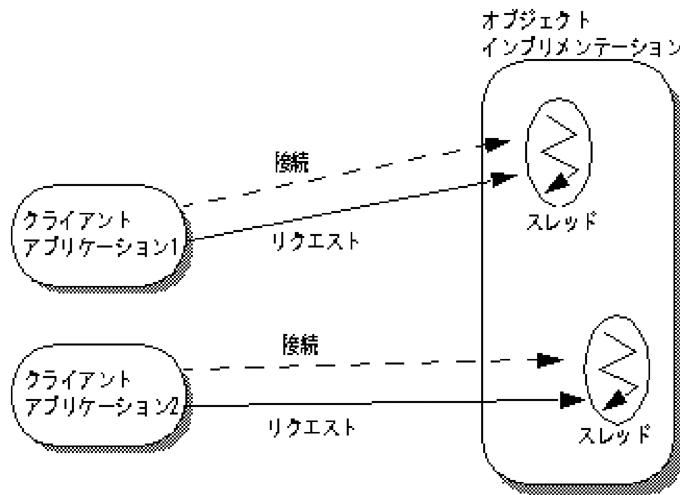


上の図では，クライアントアプリケーション＃1から2番めの要求が着信し，作業スレッド＃4が使用されています。このとき，新しい要求を監視する作業スレッド＃5が作成されます。クライアントアプリケーション＃1からさらに要求が着信すると，監視中のスレッドが要求を受信するたびに新しいスレッドが生成され，要求の処理に割り当てられます。作業スレッドは，タスクを完了するとプールに戻り，再びクライアントからの要求を処理できる状態になります。

セッションごとのスレッドポリシー

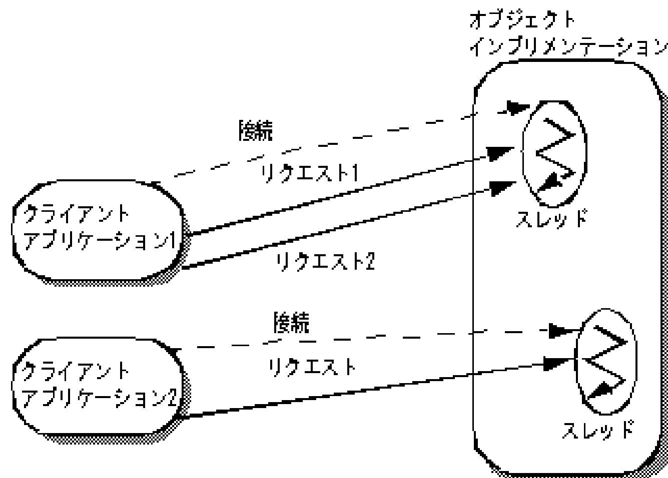
セッションごとのスレッド（TSession）ポリシーでは，クライアントとサーバーのプロセス間の接続によってスレッドが駆動されます。サーバーでセッションごとのスレッドポリシーを選択すると，新しいクライアントがサーバーに接続するたびに，新しいスレッドが割り当てられます。特定のクライアントから受信したすべての要求を処理するために，1つのスレッドが割り当てられます。このため，セッションごとのスレッドポリシーは，接続ごとのスレッドポリシーとも呼ばれます。クライアントがサーバーから切断すると，スレッドは破棄されます。クライアント接続に割り当てることのできる最大のスレッド数を制限する場合は，vbroker.se.iiop_ts.scm.iiop_ts.manager.connectionMax プロパティを設定します。

図 10.5　セッションごとのスレッドポリシーを使用するオブジェクトインプリメンテーション



上の図は，セッションごとのスレッドポリシーの使い方です。まず，クライアントアプリケーション＃1 がオブジェクトインプリメンテーションとの接続を確立します。クライアントアプリケーション＃2 とオブジェクトインプリメンテーションの間には，別の接続が確立されています。クライアントアプリケーション＃1 からオブジェクトインプリメンテーションに要求が着信した場合は，作業スレッドがこの要求を処理します。クライアントアプリケーション＃2 から要求が着信した場合は，別の作業スレッドが割り当てられ，その要求を処理します。

図 10.6　同じクライアントから第 2 の要求が着信



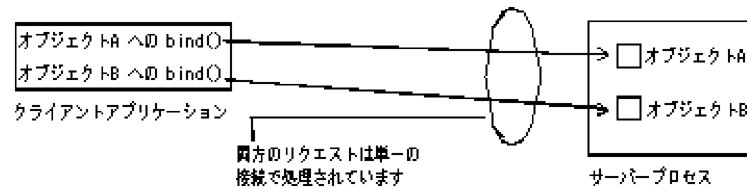
上の図では，クライアントアプリケーション＃1 からオブジェクトインプリメンテーションに 2 番めの要求が着信しています。この要求 2 は，要求 1 を処理したスレッドと同じスレッドによって処理されます。このため，このスレッドは，要求 1 の処理を完了するまで，要求 2 をブロックします。セッションごとのスレッドポリシーでは，同じクライアントからの要求は同時に処理されません。要求 1 が完了すると，スレッドはクライアントアプリケーション＃1 からの要求 2 を処理します。クライアントアプリケーション＃1 から複数の要求を着信することができます。この複数の要求は着信順に処理され，クライアントアプリケーション＃1 に別のスレッドが割り当てられることはありません。

接続管理

VisiBroker の接続管理は，クライアントからサーバーへの接続数を最小限に抑えます。言い換えれば，共有されるサーバープロセスにつき 1 つの接続があるだけです。1 つのクライアントアプリケーションからのすべての要求は，異なるスレッドから生じたものであっても，同じ接続上に多重化されます。また，解放されたクライアント接続は，その後，同じサーバーに再び接続するときに再利用されます。したがって，クライアントが同じサーバーに新しく接続するためのオーバーヘッドの必要性がなくなります。

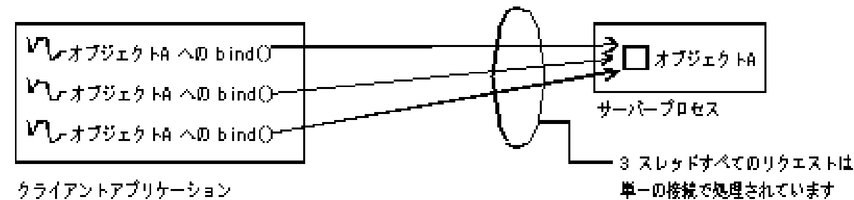
次の例では，サーバープロセス内の 2 つのオブジェクトに 1 つのクライアントアプリケーションがバインドされています。bind() ごとに対象となるサーバープロセス内のオブジェクトは異なりますが，bind() は，サーバープロセスへの接続は共有しています。

図 10.7　同じサーバープロセス内の 2 つのオブジェクトへのバインディング



次の図は，複数のスレッドを使用するクライアントの接続を示します。このクライアントが持つ複数のスレッドは，サーバーの単一のオブジェクトにバインドされています。

図 10.8　1 つのサーバープロセス内の 1 つのオブジェクトへのバインディング



上の図のように，すべてのスレッドからのすべての呼び出しは，同じ接続で処理します。その場合，最も効率的なマルチスレッドモデルは，スレッドプールモデルです（これがデフォルトです）。この例でセッションごとのスレッドモデルを使用すると，サーバーの 1 つのスレッドがクライアントアプリケーションのすべてのスレッドからのすべての要求を処理することになるので，通常，パフォーマンスが低下します。

サーバーへの接続またはクライアントからの接続の最大数を設定できます。最大接続数に達すると，非アクティブな接続が再利用されるので，リソースを節約できます。

ServerEngines

サーバー側のスレッドと接続の管理は ServerEngine によって実行され，ServerEngine は 1 つ以上のサーバー接続マネージャ（SCM）で構成されます。SCM は，マネージャ，リスナー，およびディスパッチャのプロパティの集まりです。

ServerEngine は，プロパティファイルに一連のプロパティを指定することで定義されます。たとえば，UNIX では，myprops.properties というプロパティファイルがホームディレクトリにあり，コマンドラインで次のように指定します。

```
prompt> vbj -DORBpropStorage=~/myprops.properties myServer
```

ServerEngine のプロパティ

```
vbroker.se.<srvr_eng_name>.scms=<srvr_connection_mngr_name1>,<srvr_connection_mngr_name2>
```

ServerEngine に関連付けられるサーバー接続マネージャは，このプロパティによって定義されます。上のプロパティで <svr_eng_name> として指定される名前は，ServerEngine の名前です。ここにリストされる SCM は，関連するサーバーエンジンの SCM のリストです。SCM は ServerEngine 間で共有できません。ただし，ServerEngine は複数の POA で共有できます。

ほかのプロパティは次のとおりです。

```
vbroker.se.<se>.host
```

host プロパティは，サーバーエンジンがメッセージを監視するための IP アドレスです。

```
vbroker.se.<se>.proxyHost
```

proxyHost プロパティは，サーバーが実際のホスト名を公開しない場合にクライアントに送信するプロキシの IP アドレスを指定します。

ディスパッチのポリシーとプロパティの設定

マルチスレッドオブジェクトのサーバーにある各 POA は，2 つのディスパッチモデル（*セッションごとのスレッド*または*スレッドプール*）から選択できます。ディスパッチポリシーを選択するには，ServerEngine の dispatcher.type プロパティを設定します。

```
vbroker.se.<srvr_eng_name>.scm.<srvr_connection_mngr_name>.dispatcher.type=
    ThreadPool
vbroker.se.<srvr_eng_name>.scm.<srvr_connection_mngr_name>.dispatcher.type=
    ThreadSession
```

以上のプロパティの詳細については，第 9 章「POA の使い方」と『*VisiBroker プログラマーズリファレンス*』を参照してください。

スレッドプールディスパッチポリシー

ServerEnginePolicy を指定しないで POA を作成した場合は，ThreadPool（スレッドプール）がデフォルトのディスパッチポリシーになります。

ThreadPool には，次のプロパティを設定できます。

- vbroker.se.default.dispatcher.tp.threadMax

  このプロパティは，TPool サーバーエンジンにスレッドプール内の作業スレッドの最大数を設定します。このプロパティは，サーバーの起動時に静的に設定するか，プロパティ API で動的に再設定できます。たとえば，次の起動時プロパティがあるとします。

  ```
  vbroker.se.default.dispatcher.tp.threadMax=32
  ```

  または

  ```
  vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.threadMax=32
  ```

  これは，デフォルトの TPool サーバーエンジンに，作業スレッドの最大数の初期値として 32 を設定します。このプロパティのデフォルト値は 0（無制限）です。使用可能なスレッドがプール内になく，すでに最大数のスレッドが作成されている場合は，現在使用されているスレッドが解放されてプールに戻されるまで，要求はブロックされます。

- vbroker.se.default.dispatcher.tp.threadMin

  このプロパティは，TPool サーバーエンジンにスレッドプール内の作業スレッドの最小数を設定します。このプロパティは，サーバーの起動時に静的に設定するか，プロパティ API で動的に再設定できます。たとえば，次の起動時プロパティがあるとします。

  ```
  vbroker.se.default.dispatcher.tp.threadMin=8
  ```

または

```
vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.threadMin=8
```

これは，デフォルトの TPool サーバーエンジンに，作業スレッドの最小数の初期値として 8 を設定します。このプロパティのデフォルト値は 0（作業スレッドなし）です。

- vbroker.se.default.dispatcher.tp.threadMaxIdle

  このプロパティは，TPool サーバーエンジンのアイドルスレッドのチェック間隔を設定します。このプロパティは，サーバーの起動時に静的に設定するか，プロパティ API で動的に再設定できます。たとえば，次の起動時プロパティがあるとします。

  ```
  vbroker.se.default.dispatcher.tp.threadMaxIdle=120
  ```

  または

  ```
  vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.threadMaxIdle=120
  ```

  これは，デフォルトの TPool サーバーエンジンに，アイドル状態の作業スレッドのチェック間隔として 120 秒を設定します。このプロパティのデフォルト値は 300 秒です。この設定の場合，サーバーエンジンは，各作業スレッドのアイドル状態を 120 秒ごとにチェックします。2 回のチェックで連続してアイドル状態の作業スレッドは，2 回めのチェックで再利用（終了）されます。したがって，上の設定の場合，アイドルスレッドの実際のガベージコレクション時間は，120 秒ちょうどではなく 120 ～ 240 秒になります。

- vbroker.se.default.dispatcher.tp.coolingTime

  ThreadPool ディスパッチャで，「冷却時間」を設定できます。接続の作成時または要求の到着時に，サービスを提供されている GIOP 接続が読み取り可能な場合，スレッドは「ホット」です。冷却時間（秒）が経過すると，スレッドはスレッドプールに戻すことができます。このプロパティは，サーバーの起動時に静的に設定するか，プロパティ API で動的に再設定できます。たとえば，次の起動時プロパティがあるとします。

  ```
  vbroker.se.default.dispatcher.tp.coolingTime=6
  ```

  または

  ```
  vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.coolingTime=6
  ```

  これは，デフォルトのエンジン（IIOP TPool サーバーエンジン）の冷却時間の初期値として 6 秒を設定します。

  このプロパティのデフォルト値は 3 秒です。最大値は 10 秒です。

メモ vbroker.se.default.xxx.tp.xxx プロパティは，vbroker.se.default=iiop_tp の場合に使用することをお勧めします。ThreadSession で使用する場合は，vbroker.se.iiop_ts.scm.iiop_ts.xxx プロパティの使用をお勧めします。

セッションごとのスレッドのディスパッチポリシー

ディスパッチャタイプとして ThreadSession を使用する場合は，se.default プロパティを iiop_ts に設定してください。

```
vbroker.se.default=iiop_ts
```

メモ セッションごとのスレッドには，threadMin，threadMax，threadMaxIdle，coolingTime の各ディスパッチャプロパティがありません。ThreadSession では，接続とマネージャのプロパティだけが有効なプロパティです。

コーディングにおける留意点

VisiBroker ORB オブジェクトを実装するサーバー内のすべてのコードは，スレッドセーフである必要があります。オブジェクトインプリメンテーション内でシステム全体のリソースにアクセスする場合は，特に注意が必要です。たとえば，多くのデータベースアク

セスメソッドは，スレッドセーフではありません。オブジェクトインプリメンテーションからそのようなリソースにアクセスする前に，同期ブロックを使用して，そのリソースへのアクセスを最初にロックしておく必要があります。

オブジェクトへのシリアライゼーションアクセスが必要な場合は，`ThreadPolicy` の `SINGLE_THREAD_MODEL` 値に設定して，このオブジェクトをアクティブ化する POA を作成する必要があります。

接続管理プロパティの設定

次のプロパティを使用して，接続管理を設定します。名前が `vbroker.se` で始まるプロパティは，サーバー側のプロパティです。クライアント側のプロパティは，名前が `vbroker.ce` で始まります。

メモ　VisiBroker 3.x 下位互換のコマンドラインオプションは，オプションがクライアント側であるかサーバー側であるかの表現が不明確です。ただし，プレフィクス `-ORB` で始まる接続とスレッドの管理オプションはクライアント側のオプションで使用され，プレフィクス `-OA` で始まるオプションはサーバー側のオプションで使用されます。クライアント側とサーバー側の両方で使用されるスレッドと接続の管理の共通プロパティはありません。

コールバックまたは双方向の GIOP が使用される場合，クライアントとサーバーの区別はなくなります。

- `vbroker.se.default.socket.manager.connectionMax`

  このプロパティは，サーバーエンジンにクライアント接続の最大許容数を設定します。このプロパティは，サーバーの起動時に静的に設定するか，プロパティ API で動的に再設定できます。たとえば，次の起動時プロパティがあるとします。

  ```
  -Dvbroker.se.default.socket.manager.connectionMax=128
  ```

  または

  ```
  -Dvbroker.se.iiop_tp.scm.iiop_tp.manager.connectionMax=128
  ```

  これは，このサーバーエンジンの最大接続数の初期値として `128` を設定します。このプロパティのデフォルト値は `0`（無制限）です。新しいクライアント接続を受け入れる前にサーバーエンジンがこの制限値に達した場合，サーバーエンジンはアイドル状態の接続を再利用する必要があります。これは接続スワップと呼ばれます。新しい接続がサーバーに到達すると，サーバーは，最も古い未使用の接続を解除しようとします。すべての接続がビジーである場合，新しい接続は無視されます。クライアントは，タイムアウトになるまで再試行できます。

- `vbroker.se.default.socket.manager.connectionMaxIdle`

  このプロパティは，アイドル状態の接続をサーバーエンジンで開いたままにできる最大時間を設定します。このプロパティは，サーバーの起動時に静的に設定するか，プロパティ API で動的に再設定できます。たとえば，次の起動時プロパティがあるとします。

  ```
  -Dvbroker.se.default.socket.manager.connectionMaxIdle=300
  ```

  または

  ```
  -Dvbroker.se.iiop_tp.scm.iiop_tp.manager.connectionMaxIdle=300
  ```

  これは，アイドル状態の接続の最大存続期間の初期値として 300 秒を設定します。このプロパティのデフォルト値は 0（無制限）です。クライアント接続がこの値より長くアイドル状態を続けると，ガベージコレクションの候補になります。

- `vbroker.ce.iiop.ccm.connectionMax`

  クライアントごとの接続総数の最大数を指定します。これは，アクティブな接続の数とキャッシュされている接続の数の合計です。デフォルト値の 0 の場合，クライアントは，既存のアクティブな接続とキャッシュされている接続のどちらも閉じようとしません。新しいクライアント接続がこのプロパティで設定された制限値を超えると，VisiBroker for C++ は，キャッシュされた接続の 1 つを解放しようとします。キャッシュされた接

続がない場合は，最も古いアイドル状態の接続を閉じようとします。両方の試行がともに失敗すると，CORBA::NO_RESOURCE 例外が発生します。

適用できるプロパティの有効値

次のプロパティには，有効な値がいくつかに固定されているか，有効な値の範囲があります。

- vbroker.ce.iiop.ccm.type=Pool

  現時点では，Pool だけがサポートされるタイプです。

以下のプロパティで，xxx はサーバーエンジンの名前，yyy はサーバー接続マネージャの名前です。

- vbroker.se.xxx.scm.yyy.manager.type=Socket

  ほかの有効な値には，Local（LIOP）と BIDIR（双方向 SCM）があります。

- vbroker.se.xxx.scm.yyy.listener.type=IIOP

  LIOP（ローカル IPC）と SSL（セキュリティ）も使用できます。

- vbroker.se.xxx.scm.yyy.disptacher.type=ThreadPool

  ほかの有効な値は，ThreadSession と MainThread です。

- vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.coolingTime

  デフォルト値は 3，最大値は 10 です。したがって，10 を超える値は 10 とみなされます。

プロパティの変更の影響

プロパティ値の変更の影響は，プロパティに関連付けられた動作によって異なります。ほとんどの動作は，システムリソースの利用に直接または間接的に関係しています。CORBA アプリケーションに対するシステムリソースの可用性と制限は，システムやアプリケーションの性質によって異なります。

たとえば，ガベージコレクタタイマーの値を減らすと，ガベージコレクタが頻繁に実行されるようになり，システムの動作が増加します。一方，この値を増やすと，アイドルスレッドが回収されないままシステムに長時間残るようになります。

動的に変更できるプロパティ

次のプロパティは動的に変更できます。特に断らない限り，変更内容はすぐに反映されます。

```
vbroker.ce.iiop.ccm.connectionCacheMax=5
vbroker.ce.iiop.ccm.connectionMax=0
vbroker.ce.iiop.ccm.connectionMaxIdle=360
vbroker.ce.iiop.connection.rcvBufSize=0
vbroker.ce.iiop.connection.sendBufSize=0
vbroker.ce.iiop.connection.tcpNoDelay=false
vbroker.ce.iiop.connection.socketLinger=0
vbroker.ce.iiop.connection.keepAlive=true
vbroker.ce.liop.ccm.connectionMax=0
vbroker.ce.liop.ccm.connectionMaxIdle=360
vbroker.ce.liop.connection.rcvBufSize=0
vbroker.ce.liop.connection.sendBufSize=0
vbroker.se.iiop_tp.scm.iiop_tp.manager.connectionMax=0
vbroker.se.iiop_tp.scm.iiop_tp.manager.connectionMaxIdle=0
vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.threadMin=0
vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.threadMax=100
```

ディスパッチャの新しい threadMax 関連のプロパティは，次のガベージコレクタの実行後に反映されます。

```
vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.threadMaxIdle=300
vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.coolingTime=3
vbroker.se.iiop_tp.scm.iiop_tp.manager.garbageCollectTimer=30
vbroker.se.liop_tp.scm.liop_tp.listener.userConstrained=false
```

プロパティ値の変更が有効かどうかの確認

これを確認するには，プロパティ vbroker.orb.enableServerManager=true を使ってサーバーマネージャを有効にし，コンソールまたはコマンドラインユーティリティのいずれかでサーバーマネージャに照会してプロパティを取得します。

プロパティ値の変更による影響

プロパティの値をデフォルト以外の値に変更した場合の影響を判断することは，非常に困難です。スレッドと接続の制限の場合，使用できるシステムリソースは，コンピュータの設定と実行中のほかのプロセスの数によって異なります。プロパティを設定することで，各システムのパフォーマンスを調整できます。

ガベージコレクション

VisiBroker のディスパッチャのスレッドプールには，アイドルタイムアウト vbroker.se.xxx.scm.xxx.dispatcher.threadMaxIdle があります。デフォルト値は 300 秒です。アイドル時間がタイムアウトになると，ディスパッチャがスレッドプール内のアイドル状態の作業スレッドを削除します。

サーバー接続マネージャ（SCM）には，独自にガベージコレクションのタイムアウト vbroker.se.xxx.scm.xxx.manager.garbageCollectTimer があります。デフォルト値は 30 秒です。タイムアウトになると，アイドル状態の接続にガベージコレクションが実行されます。

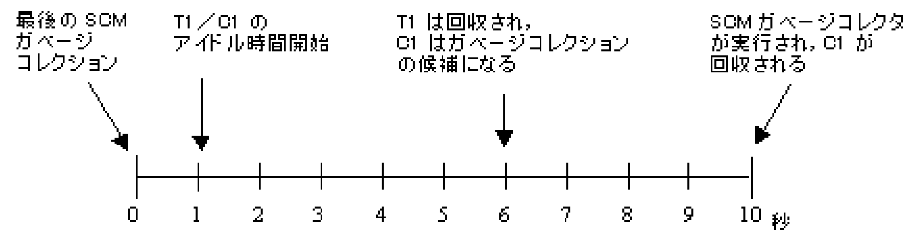
SCM はアイドル状態の接続にガベージコレクションを実行するだけなので，接続がアイドル状態に移行するように，プロパティ vbroker.se.xxx.scm.xxx.manager.connectionMaxIdle を 0 より大きい値に設定する必要があります。デフォルト値は 0 です。これは，SCM のガベージコレクションタイムアウトが発生しても，ガベージコレクションが実行されるアイドル状態の接続が存在しないことを示します。

ディスパッチャと SCM は個別にガベージコレクションを実行し，ORB 自体が実行するガベージコレクションはありません。したがって，次のように値を指定します。

```
vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.threadMaxIdle=5
vbroker.se.iiop_tp.scm.iiop_tp.manager.connectionMaxIdle=5
vbroker.se.iiop_tp.scm.iiop_tp.manager.garbageCollectTimer=10
```

スレッドプールの作業スレッド T1 は，5 秒間アイドル状態になると，ディスパッチャのスレッドプールからすぐに削除されます。5 秒間アイドル状態にある接続 C1 には，10 秒後に SCM によってガベージコレクションが実行されます。

図 10.9 SCM GC によるリソースの回収



クライアント側でクライアント接続マネージャ（CCM）のキャッシュされた接続にアイドルタイムアウトを指定するには，プロパティ vbroker.ce.xxx.ccm.connectionMaxIdle を設定します。デフォルト値は 0 で，キャッシュされた接続がアイドルタイムアウトにならないことを示します。アイドルタイムアウトを指定すると，接続プール／キャッシュ内にキャッシュされたアイドル状態の接続がガベージコレクションの候補としてマークされます。SCM とは異なり，CCM にはガベージコレクションタイマーがありません。ただし，接続がキャッシュされるたびに，CCM は，候補としてマークされた接続にガベージコレクションを実行しようとします。

第 11 章

tie メカニズムの使い方

ここでは，tie メカニズムの使い方について説明します。tie メカニズムを使用して，既存の C++ コードを分散オブジェクトシステムに統合できます。この節を通して，デリゲーションインプリメンテーションを作成する方法やインプリメンテーションを継承する方法を学びます。

tie メカニズムのしくみ

オブジェクトインプリメンテーションクラスは，`idl2cpp` コンパイラで生成されるサーバントクラスを継承しています。一方，サーバントクラスは，`PortableServer.Servant::Servant` を継承します。既存のクラスを変更して VisiBroker サーバントクラスを継承することが難しい場合は，かわりに *tie* メカニズムを使用できます。

tie メカニズムは，`PortableServer::Servant` クラスを継承する*デリゲータインプリメンテーション*クラスをオブジェクトサーバーに提供します。デリゲータインプリメンテーションが自身のセマンティクスを提供することはありません。デリゲータインプリメンテーションは，受け取る要求に対して，別個に実装する実際のインプリメンテーションクラスを代理するだけです。実際のインプリメンテーションクラスが `PortableServer::Servant` を継承する必要はありません。

tie メカニズムを使用した場合は，IDL コンパイラによって 2 つの追加ファイルが生成されます。

- `<interface_name>POATie` は，すべての IDL 定義メソッドのインプリメンテーションをデリゲートに任せます。デリゲートは，インターフェース `<interface_name>Operations` を実装します。レガシーインプリメンテーションを少し拡張してオペレーションインターフェースを実装すると，実際のインプリメンテーションを代理できます。
- `<interface_name>Operations` は，オブジェクトインプリメンテーションで実装が必要なすべてのメソッドを定義します。tie メカニズムを使用する場合，このインターフェースは，関連する `<interface_name>POATie` クラスのデリゲートオブジェクトとして機能します。

サンプルプログラム

tie メカニズムを使用するサンプルプログラムの場所

tie メカニズムを使用する Bank サンプルのバージョンは，次の場所にあります。

```
<install_dir>¥vbe¥examples¥basic¥bank_tie
```

tie テンプレートの概要

idl2cpp コンパイラは，次のサンプルコードに示す _tie_Account テンプレートクラスを自動的に生成します。POA_Bank_Account_tie クラスは，オブジェクトサーバーでインスタンス化され，AccountImpl のインスタンスで初期化されます。POA_Bank_Account_tie クラスは，受信した各オペレーションリクエストを実際のインプリメンテーションクラス AccountImpl にデリゲートします。この例では，クラス AccountImpl は，POA_Bank::Account クラスを継承しません。

```
...
template <class T> class POA_Bank_Account_tie :
    public POA_Bank::Account {
  private:
    CORBA::Boolean _rel;
    PortableServer::POA_ptr _poa;
    T *_ptr;
    POA_Bank_Account_tie(const POA_Bank_Account_tie&) {}
    void operator=(const POA_Bank_Account_tie&) {}
  public:
    POA_Bank_Account_tie (T& t): _ptr(&t), _poa(NULL),
      _rel((CORBA::Boolean)0) {}
    POA_Bank_Account_tie (T& t,
      PortableServer::POA_ptr poa): _ptr(&t),
      _poa(PortableServer::_duplicate(poa)),
      _rel((CORBA::Boolean)0) {}
    POA_Bank_Account_tie (T *p, CORBA::Boolean release= 1) : _ptr(p),
      _poa(NULL), _rel(release) {}
    POA_Bank_Account_tie (T *p, PortableServer::POA_ptr poa,
      CORBA::Boolean release =1): _ptr(p),
      _poa(PortableServer::_duplicate(poa)), _rel(release) {}
    virtual ~POA_Bank_Account_tie() {
      CORBA::release(_poa);
      if (_rel) {
        delete _ptr;
      }
    }
    T* _tied_object() { return _ptr; }
    void _tied_object(T& t) {
      if (_rel) {
        delete _ptr;
      }
      _ptr = &t;
      _rel = 0;
    }
    void _tied_object(T *p, CORBA::Boolean release=1) {
      if (_rel) {
        delete _ptr;
      }
      _ptr = p;
      _rel = release;
    }
    CORBA::Boolean _is_owner() { return _rel; }
```

```
    void _is_owner(CORBA::Boolean b) { _rel = b; }
    CORBA::Float balance() {
      return _ptr->balance();
    }
    PortableServer::POA_ptr _default_POA() {
      if ( !CORBA::is_nil(_poa) ) {
        return _poa;
      } else {
        return PortableServer_ServantBase::_default_POA();
      }
    }
};
```

_tie_account クラスを使用するためのサーバーの変更

次のサンプルコードは，_tie_account クラスを使用するために必要な Server.C ファイルに対する変更を示しています。

```
#include "Bank_s.hh"
#include <math.h>
. . .
int main(int argc, char* const* argv) {
  try {
    // ORB を初期化します。
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
    // ルート POA へのリファレンスを取得します。
    PortableServer::POA_var rootPOA =
      PortableServer::POA::_narrow(
        orb->resolve_initial_references("RootPOA"));
    CORBA::PolicyList policies;
    policies.length(1);
    policies[(CORBA::ULong)0] = rootPOA->create_lifespan_policy(
      PortableServer::PERSISTENT);
    // POA マネージャを取得します。
    PortableServer::POAManager_var poa_manager =
      rootPOA->the_POAManager();
    // 適切なポリシーで myPOA を作成します。
    PortableServer::POA_var myPOA = rootPOA->create_POA(
      "bank_agent_poa", poa_manager, policies);
    // サーバントを作成します。
    AccountManagerImpl managerServant(rootPOA);
    // デリゲータを作成します。
    POA_Bank_AccountManager_tie<AccountManagerImpl>
      tieServer(managerServant);
    // サーバントの ID を決定します。
    PortableServer::ObjectId_var managerId =
      PortableServer::string_to_ObjectId("BankManager");
    // その ID を使って myPOA でサーバントをアクティブ化します。
    myPOA->activate_object_with_id(managerId, &tieServer);
    // POA マネージャをアクティブ化します。
    poa_manager->activate();
    cout << myPOA->servant_to_reference(&tieServer) <<
      " is ready" << endl;
    // 着信要求を待機します。
    orb->run();
  } catch(const CORBA::Exception& e) {
    cerr << e << endl;
    return 1;
  }
  return 0;
}
```

tie サンプルのビルド

「「VisiBroker を使ったサンプルアプリケーションの開発」」で説明されている手順は，tie サンプルのビルドにも適用できます。

第 12 章

クライアントの基礎

この節では，クライアントプログラムから分散オブジェクトにアクセスして使用する方法について説明します。

VisiBroker ORB の初期化

Object Request Broker（ORB）は，クライアントとサーバーの間の通信リンクを提供します。クライアントから要求があると，VisiBroker ORB がそのオブジェクトインプリメンテーションを検索し，必要に応じてそのオブジェクトをアクティブ化し，要求をそのオブジェクトに渡し，応答をクライアントに返します。オブジェクトがクライアントと同じマシン上にあるか，ネットワーク上にあるかは，クライアントにはわかりません。

VisiBroker ORB が行う作業の多くは開発者に透過的ですが，作成するクライアントプログラムの中では，VisiBroker ORB を明示的に初期化する必要があります。『*VisiBroker プログラマーズリファレンス*』の第 4 章「C++ 対応プログラマツール」で説明されている VisiBroker ORB オプションは，コマンドライン引数として指定できます。これらのオプションを有効にするには，argc 引数と argv 引数を ORB_init に渡す必要があります。次のサンプルコードは，VisiBroker ORB の初期化の具体例です。

```
#include <fstream.h>
#include "Bank_c.hh"

int main(int argc, char* const* argv) {
  CORBA::ORB_var orb;
  CORBA::Float balance;
  try {
    // ORB を初期化します。
    orb = CORBA::ORB_init(argc, argv);
    . . .
}
```

オブジェクトへのバインド

クライアントプログラムは，リモートオブジェクトへのリファレンスを取得することによってリモートオブジェクトを使用します。通常，オブジェクトリファレンスは，<interface> _bind() メソッドを使って取得します。VisiBroker ORB は，オブジェクトを

実装しているサーバーの検索やそのサーバーへの接続の確立など，オブジェクトリファレンスの取得に関する大部分の処理をクライアントから隠します。

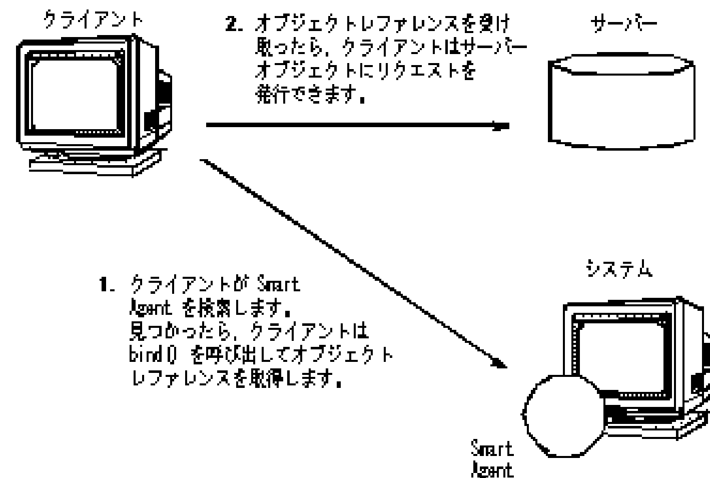
バインド処理中に実行される動作

サーバーのプロセスが起動すると，サーバーは，`CORBA::ORB.init()` を実行して，自分自身をネットワーク上のスマートエージェントに通知します。

クライアントプログラムが `_bind()` メソッドを呼び出すと，VisiBroker ORB は，プログラムにかわっていくつかの機能を実行します。

- VisiBroker ORB は，スマートエージェントにコンタクトして，要求されたインターフェースを提供するオブジェクトインプリメンテーションを検索します。オブジェクト名を指定して `_bind()` が呼び出された場合は，その名前を使用して，ディレクトリサービスの検索をさらに限定します。サーバーオブジェクトが Object Activation Daemon（OAD）を使って登録されている場合は，OAD がこの処理にかかわることがあります。OAD については，第 20 章「オブジェクトアクティベーションデーモン（OAD）の使い方」を参照してください。
- オブジェクトインプリメンテーションが見つかった場合，VisiBroker ORB は，そのオブジェクトインプリメンテーションとクライアントプログラムとの間に接続を確立しようとします。
- 正しく接続が確立されると，VisiBroker ORB はプロキシオブジェクトを作成し，そのオブジェクトへのリファレンスを返します。クライアントによってプロキシオブジェクトのメソッドが呼び出されると，プロキシオブジェクトがサーバーオブジェクトと対話します。

図 12.1　クライアントとスマートエージェントとの対話



メモ　クライアントプログラムがサーバークラスのコンストラクタを呼び出すことはありません。かわりに，静的 `_bind()` メソッドを呼び出して，オブジェクトリファレンスを取得します。

```
PortableServer::ObjectId_var manager_id =
PortableServer::string_to_ObjectId("BankManager");
Bank::AccountManager_var =
  Bank::AccountManager::_bind("/bank_agent_poa", manager_id);
```

オブジェクトのオペレーションの呼び出し

クライアントプログラムは，オブジェクトリファレンスを使用して，オブジェクトのオペレーションを呼び出したり，オブジェクトが保持するデータを参照します。オブジェクトリファレンスの操作方法については，137 ページの「オブジェクトリファレンスの操作」を参照してください。

次のサンプルは，オブジェクトリファレンスを使ってオペレーションを呼び出します。

```
// balance オペレーションを呼び出します。
balance = account->balance();
cout << "Balance is $" << balance << endl;
```

オブジェクトリファレンスの操作

_bind() メソッドは，クライアントプログラムに CORBA オブジェクトのリファレンスを返します。クライアントプログラムは，このオブジェクトリファレンスを使用して，オブジェクトの IDL インターフェース仕様で定義されているオブジェクトのオペレーションを呼び出すことができます。さらに，すべての VisiBroker ORB オブジェクトが CORBA::Object クラスから継承するメソッドがあります。これらのメソッドを使用して，オブジェクトを操作できます。

nil リファレンスをチェックする

次に示す CORBA クラスのメソッド is_nil() を使用すると，オブジェクトリファレンスが nil であるかどうかを判定できます。このメソッドは，渡されたオブジェクトリファレンスが nil の場合は 1 を返し，nil でない場合は 0 を返します。

```
class CORBA {
  . . .
  static Boolean is _nil(CORBA::Object_ptr obj);
  . . .
};
```

nil リファレンスを取得する

nil オブジェクトリファレンスは，CORBA::Object クラスの _nil() メンバー関数を使って取得できます。このメンバー関数は，Object_ptr にキャストされた NULL 値を返します。

```
class Object {
  . . .
  static CORBA::Object_ptr _nil();
  . . .
};
```

オブジェクトリファレンスを複製する

クライアントプログラムが _duplicate メンバー関数を呼び出すと，オブジェクトリファレンスのリファレンスカウントが 1 だけインクリメントされ，同じオブジェクトリファレンスが返されます。クライアントプログラムでオブジェクトリファレンスをデータ構造に保存したり，パラメータとして渡すには，_duplicate メンバー関数を使用して，そのオブジェクトリファレンスのリファレンスカウントを増やします。リファレンスカウントを増やすことにより，リファレンスカウントが 0 になるまで，そのオブジェクトリファレンスに関連付けられているメモリは解放されません。

IDL コンパイラは，指定されたオブジェクトインターフェースごとに _duplicate メンバー関数を生成します。_duplicate メンバー関数は，汎用の Object_ptr を受け取って返します。

```
class Object {
  . . .
  static CORBA::Object_ptr _duplicate(CORBA::Object_ptr obj);
  . . .
};
```

メモ　POA や VisiBroker ORB はリファレンスカウントをサポートしないため，これらのオブジェクトに対して _duplicate メンバー関数を使用することは意味がありません。

オブジェクトリファレンスを解放する

必要なくなったオブジェクトリファレンスは，解放する必要があります。オブジェクトリファレンスを解放する方法の 1 つとして，CORBA::Object クラスの _release メンバー関数の呼び出しがあります。

注意：　必ず release メンバー関数を使用してください。オブジェクトリファレンスに対して delete 演算子を呼び出さないでください。

```
class CORBA {
  class Object {
    . . .
    void _release();
    . . .
  };
};
```

CORBA クラスの release メンバー関数を使用することもできます。この関数は，CORBA 仕様との互換性を保つために用意されています。

```
class CORBA {
  . . .
  static void release();
  . . .
};
```

リファレンスカウントを取得する

オブジェクトリファレンスは，それぞれのリファレンスカウントを持ちます。リファレンスカウントを使用すると，オブジェクトリファレンスが何回複製されたかを判定できます。最初に _bind() を呼び出してオブジェクトリファレンスを取得したとき，リファレンスカウントは 1 に設定されます。オブジェクトリファレンスを解放すると，リファレンスカウントは 1 だけ減らされます。リファレンスカウントが 0 になると，VisiBroker は自動的にそのオブジェクトリファレンスを削除します。次のサンプルコードは，リファレンスカウントを取得するための _ref_count メンバー関数を示しています。

メモ　リモートクライアントがオブジェクトリファレンスを複製または解放しても，サーバーのオブジェクトリファレンスのカウントは影響を受けません。

```
class Object {
  . . .
  CORBA::Long _ref_count() const;
  . . .
};
```

リファレンスを文字列に変換する

VisiBroker では，オブジェクトリファレンスを文字列に変換したり，文字列を元のオブジェクトリファレンスに再変換できるメソッドが VisiBroker ORB クラス用に用意されています。CORBA 仕様では，この処理のことを文字列化と呼んでいます。

表 12.1　リファレンスの文字列化とその復元のためのメソッド

メソッド	説明
object_to_string	オブジェクトリファレンスを文字列に変換します。
string_to_object	文字列をオブジェクトリファレンスに変換します。

クライアントプログラムは，object_to_string メソッドを使ってオブジェクトリファレンスを文字列に変換し，その文字列を別のクライアントプログラムに渡すことができます。その後，その 2 番めのクライアントが string_to_object メソッドを使ってオブジェクトリファレンスを復元すると，オブジェクトに明示的にバインドする必要なく，そのオブジェクトリファレンスを使用できます。

返された文字列に対して，object_to_string の呼び出し元が CORBA::string_free() を呼び出す必要があります。

メモ　VisiBroker ORB や POA などのローカルスコープ付きのオブジェクトリファレンスは文字列化できません。文字列化しようとすると，マイナーコード 4 とともに MARSHAL 例外が生成されます。

オブジェクト名とインターフェース名を取得する

次の表に，Object クラスによって提供されるメソッドを示します。これらのメソッドは，オブジェクト名とインターフェース名を取得するほか，オブジェクトリファレンスに関連付けられたリポジトリ ID を取得するために使用できます。インターフェースリポジトリの詳細については，第 21 章「インターフェースリポジトリの使い方」を参照してください。

メモ　オブジェクト名を指定しないで _bind() を呼び出した場合は，取得したオブジェクトリファレンスを使って _object_name() メソッドを呼び出すと，NULL が返されます。

表 12.2　インターフェース名とオブジェクト名を取得するためのメソッド

メソッド	説明
_interface_name	このオブジェクトのインターフェース名を返します。
_object_name	このオブジェクトの名前を返します。
_repository_id	リポジトリの型識別子を返します。

オブジェクトリファレンスの型を判定する

_hash() メンバー関数を使用すると，オブジェクトリファレンスのハッシュ値を取得できます。この値の一意性は保証されませんが，オブジェクトリファレンスの存続期間を通して整合性が保たれ，ハッシュテーブルに保存できます。

_is_a() メソッドを使用すると，オブジェクトリファレンスが特定の型であるかどうかをチェックできます。最初に，_repository_id() メソッドを使用して，チェックする型のリポジトリ ID を取得する必要があります。このメソッドは，オブジェクトが _repository_id() で表された型のインスタンスであるか，そのサブタイプである場合，1 を返します。このメンバー関数は，オブジェクトが指定された型でない場合，0 を返します。型の判定にはリモート呼び出しが必要であることに注意してください。

_is_equivalent() を使用して，2 つのオブジェクトリファレンスが同じオブジェクトインプリメンテーションを参照するかどうかをチェックできます。2 つのオブジェクトリファレンスが等しい場合，このメソッドは 1 を返します。オブジェクトリファレンスが等しくない場合は，0 を返しますが，これは必ずしもこの 2 つのオブジェクトリファレンスが別の

オブジェクトであることを示しません。これは簡易メソッドであり，実際にサーバーオブジェクトとは通信しません。

表 12.3　オブジェクトリファレンスの型を判定するためのメソッド

メソッド	説明
_hash	オブジェクトリファレンスのハッシュ値を返します。
_is_a	指定されたインターフェースをオブジェクトが実装しているかどうかを判定します。
_is_equivalent	2 つのオブジェクトが同一のインターフェースインプリメンテーションを参照している場合は，true を返します。

バインドされたオブジェクトの場所と状態を判定する

有効なオブジェクトリファレンスを取得できた場合，クライアントプログラムは _is_bound() を使用して，オブジェクトがバインドされているかどうかを判定できます。このメソッドは，オブジェクトがバインドされている場合は 1 を返し，オブジェクトがバインドされていない場合は 0 (zero) を返します。

_is_local() メソッドは，クライアントプログラムとオブジェクトインプリメンテーションが同じプロセス内に存在する（メソッドの呼び出し元のアドレス空間に存在する）場合に，1 を返します。

_is_remote() メソッドは，クライアントプログラムとオブジェクトインプリメンテーションが異なるプロセスに存在する場合に，1 を返します。この場合，それらのプロセスは，同じホスト上にある場合もそうでない場合もあります。

表 12.4　オブジェクトリファレンスの場所と状態を判定するためのメソッド

メソッド	説明
_is_bound	このオブジェクトに対する接続が現在アクティブであるかどうかを判定します。
_is_local	このオブジェクトがローカルのアドレス空間で実装されているかどうかを判定します。
_is_remote	このオブジェクトのインプリメンテーションがローカルのアドレス空間に存在していないかどうかを判定します。

存在しないオブジェクトをチェックする

_non_existent() メンバー関数を使用すると，オブジェクトリファレンスに関連付けられているオブジェクトインプリメンテーションが存在するかどうかを確認できます。このメソッドは，オブジェクトを実際に「ping」して，オブジェクトが存在しているかどうかを判定し，存在している場合は 1 を返します。

オブジェクトリファレンスをナローイングする

オブジェクトリファレンスの型を汎用のスーパータイプからより具体的なサブタイプに変換する処理を「ナローイング」と呼びます。

_narrow() メンバー関数は，新しい C++ オブジェクトを生成し，このオブジェクトへのポインタを返します。オブジェクトが不要になった場合は，_narrow() から返されたオブジェクトリファレンスを解放する必要があります。

各オブジェクトの narrow() メソッドを使ってナローイングを実行できるように，VisiBroker には，各オブジェクトインターフェースのタイプグラフが用意されています。

narrow メンバー関数は，要求された型にオブジェクトをナローイングできないと判定した場合，NULL を返します。

```
Account *acct;
Account *acct2;
Object *obj;
acct = Account::_bind();
```

```
obj = (CORBA::Object *)acct;
acct2 = Account::_narrow(obj);
```

オブジェクトリファレンスをワイドニングする

オブジェクトリファレンスの型をスーパータイプに変換する処理を「ワイドニング」と呼びます。次のサンプルコードは，Account ポインタを Object ポインタにワイドニングする例を示します。Account クラスは Object クラスを継承しているので，ポインタ acct は Object のポインタにキャストできます。

```
. . .
Account *acct;
CORBA::Object *obj;
acct = Account::_bind();
obj  = (CORBA::Object *)acct;. . .
```

Quality of Service（QoS）の使用

Quality of Service（QoS）は，ポリシーを使用して，クライアントアプリケーションとその接続先のサーバーとの間の接続を定義および管理します。

Quality of Service（QoS）の概要

QoS ポリシー管理は，次のような状況で利用できる操作を通して実行されます。

- VisiBroker ORB レベルのポリシーは，局所性制約付きの PolicyManager によって処理されます。この PolicyManager を介して，ポリシーを設定したり，現在の Policy オーバーライドを取得することができます。VisiBroker ORB レベルで設定されたポリシーは，システムデフォルトを上書きします。
- スレッドレベルのポリシーは，PolicyCurrent を介して設定されます。PolicyCurrent には，スレッドレベルの Policy オーバーライドを取得および設定するためのオペレーションがあります。スレッドレベルで設定されたポリシーは，システムデフォルト，および VisiBroker ORB レベルで設定された値をオーバーライドします。

メモ　VisiBroker for C++ は，スレッドレベルのポリシーをまだサポートしていません。

- オブジェクトレベルのポリシーを適用するには，ベースオブジェクトのインターフェースの QoS 操作を利用します。オブジェクトレベルで適用されたポリシーは，システムデフォルト，および VisiBroker ORB とスレッドレベルで設定された値をオーバーライドします。

メモ　ORB レベルでインストールされた QoS ポリシーは，ポリシーをインストールする前にメソッドが呼び出されていないオブジェクトだけに影響します。たとえば，non_existent 呼び出しは，内部的にサーバーオブジェクトに対して呼び出しを行います。non_existent 呼び出しの後に ORB レベルの QoS ポリシーがインストールされても，ポリシーは適用されません。

ポリシーオーバーライドと有効なポリシー

有効なポリシーとは，適用可能なポリシーオーバーライドがすべて適用された上で，要求に適用されるポリシーです。有効なポリシーは，IOR によって指定されたポリシーを有効なオーバーライドと比較することによって判定されます。有効なポリシーは，有効なオーバーライドと IOR 指定の Policy が許容する値の共通部分になります。共通する値がない場合は，org.omg.CORBA.INV_POLICY 例外が生成されます。

QoS のインターフェース

QoS ポリシーの取得および設定には，次のインターフェースが使用されます。

CORBA::Object

次のメソッドを使用して，有効なポリシーを取得したり，ポリシーオーバーライドを取得または設定します。

- _get_policy は，このオブジェクトリファレンスの有効なポリシーを返します。
- _set_policy_override は，新規のオブジェクトリファレンスとともに，オブジェクトレベルの Policy オーバーライドのリストを返します。

CORBA::Object

- _get_client_policy は，このオブジェクトリファレンスの有効な Policy を返します。サーバー側のポリシーとの共通部分は調べません。有効なオーバーライドは，オブジェクトレベル，スレッドレベル，VisiBroker ORB レベルの順序で，指定されたオーバーライドをチェックすることによって取得されます。要求対象の PolicyType に対してオーバーライドを指定しない場合，PolicyType のシステムデフォルト値が使用されます。
- _get_policy_overrides は，オブジェクトレベルで設定され，指定されたポリシータイプを持つ Policy オーバーライドのリストを返します。指定されたシーケンスが空の場合は，オブジェクトレベルのすべてのオーバーライドが返されます。オブジェクトレベルでオーバーライドされるポリシータイプがない場合は，空のシーケンスが返されます。
- _validate_connection は，オブジェクトの現在有効なポリシーが呼び出しを実行できるかどうかを表すブール値を返します。オブジェクトリファレンスがバインドされていない場合は，バインドされます。オブジェクトリファレンスがすでにバインドされており，現在のポリシーオーバーライドが変更されている場合，またはバインドが有効でない場合は，RebindPolicy オーバーライドの設定にかかわらず，リバインドが試みられます。現在有効なポリシーが INV_POLICY 例外を生成する場合は，false が返されます。現在有効なポリシーどうしに互換性がない場合は，PolicyList のシーケンスに互換性のないポリシーがリストされて返されます。

CORBA::PolicyManager

PolicyManager は，VisiBroker ORB レベルの Policy オーバーライドを取得および設定するメソッドを提供するインターフェースです。

- get_policy_overrides は，要求された PolicyTypes のオーバーライドされたポリシーのシーケンスを示す PolicyList を返します。指定されたシーケンスが空の場合は，現在のコンテキストレベルのすべての Policy オーバーライドが返されます。要求された PolicyTypes が，目的の PolicyManager でオーバーライドされていない場合は，空のシーケンスが返されます。
- set_policy_overrides は，要求された Policy オーバーライドのリストを使用して，現在のオーバーライドセットを変更します。最初の入力パラメータ policies は，複数の Policy オブジェクトを示すリファレンスのシーケンスです。2 番めのパラメータ set_add は，SetOverrideType 型です。このパラメータに ADD_OVERRIDE を使用して，PolicyManager にすでに存在するほかのオーバーライドに指定したポリシーを追加するように指定できます。または，SET_OVERRIDES を使用すると，オーバーライドを含まない PolicyManager に指定したポリシーを追加するように指定できます。ポリシーの空のシーケンスと SET_OVERRIDES モードを使って set_policy_overrides を呼び出すと，PolicyManager からすべてのオーバーライドが除去されます。クライアントに適用されないポリシーを上書きしようとすると，NO_PERMISSION が生成されます。この要求によって指定された PolicyManager に矛盾が起こる場合，ポリシーはまったく変更または追加されず，InvalidPolicies 例外が生成されます。

QoSExt::DeferBindPolicy

DeferBindPolicy は，VisiBroker ORB がリモートオブジェクトにコンタクトするタイミングを決定します。VisiBroker ORB は，リモートオブジェクトが初めて作成されたときにそれにコンタクトするか，オブジェクトが初めて呼び出されるまでコンタクトを遅延します。DeferBindPolicy の値は，true と false です。DeferBindPolicy を true に設定すると，

バインド先のインスタンスが初めて呼び出されるまですべてのバインドが遅延されます。デフォルト値は，false です。

クライアントオブジェクトを作成し，DeferBindPolicy を true に設定した場合は，サーバーの起動を最初の呼び出しまで遅延できます。このオプションは以前からあり，当初は，生成されるヘルパークラスの Bind メソッドに対するオプションでした。

次のサンプルコードは，DeferBindPolicy を作成し，VisiBroker ORB でポリシーを設定する例を示します。

```
// フラグとリファレンスを初期化します。
CORBA::Boolean deferMode = (CORBA::Boolean) 1;
CORBA::Any policy_value;
policy_value <<= CORBA::Any::from_boolean(deferMode);

CORBA::Policy_var policy =
  orb->create_policy(QoSExt::DEFER_BIND_POLICY_TYPE, policy_value);

CORBA::PolicyList policies;
policies.length(1);
policies[0] = CORBA::Policy::_duplicate(policy);

// スレッドマネージャへのリファレンスを取得します。
CORBA::Object_var obj = orb->resolve_initial_references("ORBPolicyManager");
CORBA::PolicyManager_var orb_mgr = CORBA::PolicyManager::_narrow(obj);

// ORB レベルでポリシーを設定します。
orb_mgr->set_policy_overrides(policies, CORBA::SET_OVERRIDE);
```

QoSExt::RelativeConnectionTimeoutPolicy

RelativeConnectionTimeoutPolicy には，利用可能なエンドポイントの 1 つからオブジェクトへの接続の試行が打ち切られるまでのタイムアウト値を指定できます。タイムアウトは，ファイアウォールで保護されているため接続方法が HTTP トンネリングに限られるようなオブジェクトで使用する場合が大半です。

Messaging::RebindPolicy

RebindPolicy は，ORB がターゲットに正しくバインドした後で透過的に再バインドできるかどうかを指定するために使用されます。LocateRequest メッセージの結果として OBJECT_HERE というステータスの LocateReply メッセージが戻される状態になると，オブジェクトリファレンスはバインドされているとみなされます。RebindPolicy は，Messaging::RebindMode 型の値を受け取り，クライアント側でのみ設定されます。これは，オブジェクトリファレンスがバインドされた後の接続の切断，オブジェクト転送要求，またはオブジェクト障害の場合の動作を決定する 6 つの値の 1 つになります。次の値を指定できます。

- Messaging::TRANSPARENT を使用すると，ORB は，リモート要求を行う間に，オブジェクト転送および必要な再接続を暗黙的に処理します。
- Messaging::NO_REBIND を使用すると，VisiBroker ORB は，リモート要求の実行中に，閉じた接続の再オープンを無条件に処理します。ただし，クライアントに可視の有効な QoS ポリシーを変更するような透過的なオブジェクト転送は許可しません。RebindMode が NO_REBIND に設定されている場合は，明示的なリバインドだけを実行できます。
- Messaging::NO_RECONNECT を使用すると，VisiBroker ORB は，オブジェクト転送または閉じた接続の再オープンを暗黙的に処理できません。RebindMode を NO_RECONNECT に設定している場合は，リバインドや再接続を明示的に行う必要があります。
- QoSExt::VB_TRANSPARENT はデフォルトのポリシーです。このポリシーは，暗黙的バインディングと明示的バインディングのどちらの場合でも透過的なリバインドを可能にすることにより，TRANSPARENT の機能を拡張しています。VB_TRANSPARENT の目的は，VisiBroker 3.x のオブジェクトフェイルオーバーインプリメンテーションとの互換性を維持することです。

- このポリシーのデフォルト値は QoSExt::VB_NOTIFY_REBIND です。クライアントはこの例外をキャッチし，2 度めの呼び出しでバインドを行います。CloseConnection メッセージを事前に受け取っているクライアントは，閉じた接続も再確立します。
- QoSExt::VB_NO_REBIND はフェイルオーバーを無効にします。クライアントの VisiBroker ORB は，閉じた接続を同じサーバーに向けて再オープンできるだけで，どのようなオブジェクト転送も許可されません。

メモ　クライアントの有効なポリシーが VB_TRANSPARENT であり，クライアントがサーバーの状態データを保持している場合は，次の点に注意が必要です。VB_TRANSPARENT を使用すると，クライアントは，サーバーの変更に気付かないまま新しいサーバーに接続し，元のサーバーが保持していた状態データは失われます。

メモ　クライアントが RebindPolicy を設定し，RebindMode がデフォルト（VB_TRANSPARENT）以外に設定されている場合，RebindPolicy は，CORBA 仕様にしたがって特別な ServiceContext で伝達されます。ServiceContext の伝達は，クライアントが GateKeeper または RequestAgent を使ってサーバーを呼び出す場合にだけ行われます。この伝達は，通常のクライアント／サーバーのシナリオでは行われません。

NO_REBIND または NO_RECONNECT を使用する場合，閉じた接続の再オープンおよび転送を明示的に可能にするには，CORBA::Object インターフェースの _validate_connection を呼び出します。

次の表で，さまざまな RebindMode 型の動作について説明します。

表 12.5　RebindMode ポリシー

RebindMode 型	同じオブジェクトに対する閉じた接続の再確立	オブジェクトの転送	オブジェクトのフェイルオーバー
NO_RECONNECT	いいえ。REBIND 例外が生成されます。	いいえ。REBIND 例外が生成されます。	いいえ
NO_REBIND	はい	はい（ポリシーと一致する場合） いいえ。REBIND 例外が生成されます。	いいえ
TRANSPARENT	はい	はい	いいえ
VB_NO_REBIND	はい	いいえ。REBIND 例外が生成されます。	いいえ
VB_NOTIFY_REBIND	いいえ。例外が生成されます。	はい	はい。VB_NOTIFY_REBIND は，障害の検出後に例外を生成し，それ以降に要求があるとフェイルオーバーを試みます。
VB_TRANSPARENT	はい	はい	はい。透過的に行われます。

通信で障害が発生したり，オブジェクトでエラーが発生した場合は，該当する CORBA 例外が生成されます。

QoS のポリシーとタイプの詳細については，CORBA 仕様のメッセージングに関する節を参照してください。

Messaging::RelativeRequestTimeoutPolicy

RelativeRequestTimeoutPolicy には，要求またはその応答を送信する相対的な時間を指定します。この時間を過ぎると，要求はキャンセルされます。このポリシーは同期および非同期呼び出しの両方に適用されます。指定されたタイムアウトの時間内に要求が完了すると，タイムアウトのために応答が破棄されることはありません。タイムアウト値は，100 ナノ秒（ns）単位で指定されます。このポリシーは確立された接続だけに有効で，接続の確立には適用されません。

Messaging::RelativeRoundTripTimeoutPolicy

RelativeRoundTripTimeoutPolicy には，要求またはその応答を送信する相対的な時間を表します。この時間を過ぎても応答が送信されない場合，要求はキャンセルされます。また，要求がすでに送信済みで応答が送信先から返されている場合，この時間が過ぎると応答は破棄されます。このポリシーは同期および非同期呼び出しの両方に適用されます。指定されたタイムアウトの時間内に要求が完了すると，タイムアウトのために応答が破棄されることはありません。タイムアウト値は，100 ナノ秒（ns）単位で指定されます。このポリシーは確立された接続だけに有効で，接続の確立には適用されません。

Messaging::SyncScopePolicy

SyncScopePolicy では，要求の対象に関する要求の同期レベルを定義します。SyncScope 型の値を SyncScopePolicy と組み合わせて使用して，一方向オペレーションの動作を制御します。

デフォルトの SyncScopePolicy は SYNC_WITH_TRANSPORT です。OAD を介して一方向オペレーションを実行するには，SyncScopePolicy=SYNC_WITH_SERVER を使用する必要があります。SyncScopePolicy の有効な値は OMG によって定義されます。

メモ　アプリケーションでは，VisiBroker ORB レベルの SyncScopePolicy を明示的に設定して，VisiBroker ORB のインプリメンテーション全体で可搬性を確保する必要があります。SyncScopePolicy のインスタンスを作成すると，Messaging::SyncScope 型の値が CORBA::ORB::create_policy に渡されます。このポリシーは，クライアント側オーバーライドとしてのみ適用できます。

例外

表 12.6　例外

例外	説明
CORBA::INV_POLICY	Policy オーバーライドどうしが矛盾する場合に生成されます。
CORBA::REBIND	RebindPolicy の値が NO_REBIND，NO_RECONNECT，または VB_NO_REBIND の場合に，バインドされているオブジェクトリファレンスを使って呼び出しを行った結果，オブジェクト転送メッセージまたはロケーション転送メッセージが出されると，この例外が生成されます。
CORBA::PolicyError	要求された Policy がサポートされていない場合に生成されます。
CORBA::InvalidPolicies	オペレーションに PolicyList シーケンスが渡された場合に生成されます。例外本体には，ポリシーのシーケンスから有効でないポリシーが格納されています。有効でないポリシーは，現在のスコープ内ですでにオーバーライドされているか，要求されたほかのポリシーと両立しないかのどちらかだからです。
CORBA::COMM_FAILURE	処理中に通信が失われると，クライアント側の ORB によって生成されます。処理は完了している可能性があります。
CORBA::TRANSIENT	オブジェクトに到達しようとして失敗すると，クライアント側の ORB によって生成されます。処理は成功していません。

第 13 章

IDL の使い方

この節では，CORBA インターフェース定義言語（IDL）の使い方について説明します。

IDL の概要

インターフェース定義言語（IDL）は，プログラミング言語ではなく，リモートオブジェクトが実装するインターフェースを記述するための*記述言語*です。IDL では，インターフェースの名前，属性やメソッドの名前などを定義します。IDL ファイルを作成すると，IDL コンパイラを使用して，C++ プログラミング言語で記述したクライアントスタブファイルとサーバースケルトンファイルを生成できます。

詳細については，『VisiBroker プログラマーズリファレンス』の第 4 章「C++ 対応プログラマツール」を参照してください。

このような言語マッピング仕様は，OMG が定義します。VisiBroker は OMG が設定した仕様に準拠しているため，このマニュアルでは言語マッピングに関する情報は取り扱っていません。言語マッピングの詳細については，OMG の Web サイト http://www.omg.org を参照してください。

メモ　CORBA 2.6 の正式な仕様は，http://www.omg.org/technology/documents/vault.htm#CORBA_IIOP を参照してください。

IDL について説明し始めるとページがいくらあっても足りなくなってしまいます。VisiBroker は OMG 定義の仕様に準拠しているので，IDL の詳細については，OMG のサイトを参照してください。

IDL コンパイラでコードを生成する方法

クライアントプログラムで使用するオブジェクトのインターフェースを定義するには，インターフェース定義言語（IDL）を使用します。`idl2cpp` コンパイラは，インターフェース定義を使ってコードを生成します。

IDL 仕様のサンプル

インターフェース定義では，オブジェクトの名前だけでなく，オブジェクトが提供するすべてのメソッドの名前も定義します。各メソッドは，メソッドに渡されるパラメータ，その種類，およびそれらが入力用か出力用か，またはその両方かを定義します。下の IDL サンプルは，example という名前のオブジェクトの IDL 仕様を示したものです。example オブジェクトには op1 メソッドがあります。

```
// サンプルオブジェクトの IDL 仕様
interface example {
  long op1(in char x, out short y);
};
```

クライアント用に生成されたコードの検討

次のサンプルコードは，IDL コンパイラが「IDL の使い方」から 2 つのクライアントファイル example_c.hh と example_c.cc を生成する方法を示します。この 2 つのファイルは，クライアントが使用する example クラスを提供します。IDL コンパイラの生成するファイルには，プログラマが自分で作成するファイルと区別するために，命名規則にしたがって .cc または .hh という拡張子が付けられます。必要に応じて，この命名規則にしたがわずに別の拡張子の付いたファイルを作成することもできます。

重要 IDL コンパイラが生成したファイルの内容は変更しないでください。

```
class example : public virtual CORBA_Object {
  protected:
    example() {}
    example(const example&) {}
  public:
    virtual ~example() {}
    static  const CORBA::TypeInfo *_desc();
    virtual const CORBA::TypeInfo *_type_info() const;
    virtual void *_safe_narrow(const CORBA::TypeInfo& ) const;
    static CORBA::Object*_factory();
    example_ptr _this();
    static example_ptr _duplicate(example_ptr _obj) { /* . . . */ }
    static example_ptr _nil() { /* . . . */ }
    static example_ptr _narrow(CORBA::Object* _obj);
    static example_ptr _clone(example_ptr _obj) { /* . . . */ }
    static example_ptr _bind(
      const char *_object_name = NULL,
      const char *_host_name = NULL,
      const CORBA::BindOptions* _opt = NULL,
      CORBA::ORB_ptr _orb = NULL);
    static example_ptr _bind(
      const char *_poa_name,
      const CORBA::OctetSequence& _id,
      const char *_host_name = NULL,
      const CORBA::BindOptions* _opt = NULL,
      CORBA::ORB_ptr _orb = NULL);
    virtual CORBA::Long op1(
      CORBA::Char _x, CORBA::Short_out _y);
};
```

IDL コンパイラが生成するメソッド（スタブ）

上のサンプルコードは，いくつかのその他のメソッドとともに，IDL コンパイラによって生成された op1 メソッドを示しています。op1 メソッドは，クライアントプログラムから呼び出されると，実際にインターフェース要求と引数を 1 つのメッセージにパッケージし，そのメッセージをオブジェクトインプリメンテーションに送信して応答を待ち，その応答をデコードして結果をプログラムに返すので，**スタブ**と呼ばれます。

example クラスは CORBA::Object クラスから派生するため，継承されたいくつかのメソッドをプログラムで使用できます。

ポインタ型 <interface_name>_ptr の定義

IDL コンパイラは常にポインタ型の定義を示します。下のサンプルコードは，example クラスの型定義を示しています。

```
typedef example *example_ptr;
```

自動メモリ管理 <interface_name>_var クラス

IDL コンパイラは，example_var という名前のクラスも生成します。これは example_ptr のかわりに使用できます。example_var クラスは，動的に割り当てられたオブジェクトリファレンスと関連するメモリを自動的に管理します。example_var オブジェクトが削除されるとき，example_ptr に関連しているオブジェクトを解放します。example_var オブジェクトに新しい値が代入されると，example_ptr がポイントしていた古いオブジェクトリファレンスは，代入の後に解放されます。キャスト演算子も提供されるため，それを使って example_var を example_ptr に代入できます。

```
class  example_var : public CORBA::_var {
  . . .
  public:
    static example_ptr _duplicate(example_ptr);
    static void _release(example_ptr);
    example_var();
    example_var(example_ptr);
    example_var(const example_var &);
    ~example_var();
    example_var& operator=(example_ptr);
    example_var& operator=(const example_var& _var) { /* . . . */ }
    operator example* () const { return _ptr; }
    . . .
};
```

次の表は，_var クラスのメソッドの説明です。

表 13.1　_var クラスのメソッド

メソッド	説明
example_var()	_ptr を NULL に初期化するコンストラクタ。
example_var(example_ptr ptr)	渡された引数に _ptr を初期化してオブジェクトを作成するコンストラクタ。var は，破棄されるときに _ptr の release() を呼び出します。_ptr のリファレンスカウントが 0 になると，そのオブジェクトは削除されます。
example_var(const example_var& var)	パラメータ var として渡されたオブジェクトのコピーを作成し，新しくコピーされたオブジェクトへのポインタを _ptr に設定するコンストラクタ。
~example()	_ptr がポイントするオブジェクトの _release() を一度呼び出すデストラクタ。
operator=(example_ptr p)	_ptr がポイントするオブジェクトの _release() を呼び出し，p を _ptr に保存する代入演算子。
operator=(const example_ptr p)	_ptr がポイントするオブジェクトの _release() を呼び出し，p の _duplicate() を _ptr に保存する代入演算子。
example_ptr operator->()	このオブジェクトに保存されている _ptr を返します。オブジェクトが正しく初期化されるまでは，この演算子を呼び出さないでください。

サーバー用に生成されたコードの検討

下のサンプルコードは，IDL コンパイラが 2 つのサーバーファイル example_s.hh と example_s.cc を生成する方法を示します。この 2 つのファイルは，サーバーがインプリメンテーションクラスを派生するために使用する POA_example クラスを提供します。PortableServer_ServantBase クラスから，POA_example クラスが派生します。

重要 IDL コンパイラが生成したファイルの内容は変更しないでください。

```
class  POA_example : public virtual PortableServer_ServantBase {
  protected:
    POA_example() {}
    virtual ~POA_example() {}
  public:
    static const CORBA::TypeInfo _skel_info;
    virtual const CORBA::TypeInfo *_type_info() const;
    example_ptr _this();
    virtual void *_safe_narrow(const CORBA::TypeInfo& ) const;
    static POA_example * _narrow(PortableServer_ServantBase *_obj);
    // 以下のオペレーションを実装する必要があります。
    virtual CORBA::Long op1(CORBA::Char _x, CORBA::Short_out _y) = 0;
    // 自動的に実装されるスケルトンオペレーション
    static void _op1(void *_obj, CORBA::MarshalInBuffer &_istrm,
    const char *_oper, VISReplyHandler& handler);
};
```

IDL コンパイラが生成するメソッド（スケルトン）

_op1 メソッドとともに，下の IDL 仕様で宣言されている op1 メソッドが生成されます。POA_example クラスは，op1 という名前の純粋仮想メソッド宣言します。POA_example から派生するインプリメンテーションクラスは，このメソッドのインプリメンテーションを提供しなければなりません。

POA_example クラスは**スケルトン**と呼ばれ，そのメソッド（_op1）は，クライアント要求を受信したときに POA が呼び出します。スケルトンの内部メソッドはクライアント要求用のすべてのパラメータをマーシャリングし，実装された op1 メソッドを呼び出した後，返されたパラメータまたは例外を応答メッセージにマーシャリングします。次に，ORB がその応答メッセージをクライアントプログラムに送信します。

コンストラクタとデストラクタはどちらもプロテクテッドメンバーで，派生されたメンバーからしか呼び出すことができません。コンストラクタは 1 つのオブジェクト名を受け取り，1 つのサーバーで複数のオブジェクトを個別にインスタンス化できます。

IDL コンパイラが生成するクラステンプレート

POA_example クラスに加えて，IDL コンパイラは _tie_example という名前のクラステンプレートを生成します。このテンプレートを使用すると，POA_example からクラスを派生させる必要がありません。テンプレートは，新しいクラスを継承するようは変更できない既存のアプリケーションのラッパークラスを提供する場合に便利です。下のサンプルは，IDL コンパイラによって example クラス用に生成されたテンプレートクラスを示しています。

```
template <class T>
class POA_example_tie : public POA_example {
  public:
    POA_example_tie (T& t): _ptr(&t),
      _poa(NULL), _rel((CORBA::Boolean)0) {}
    POA_example_tie (T& t,
      PortableServer::POA_ptr poa): _ptr(&t),
      _poa(PortableServer::_duplicate(poa)),
      _rel((CORBA::Boolean)0) {}
    POA_example_tie (T *p, CORBA::Boolean release= 1)
```

```
      : _ptr(p),_poa(NULL), _rel(release) {}
    POA_example_tie (T *p, PortableServer::POA_ptr poa,
      CORBA::Boolean release =1)
      : _ptr(p), _poa(PortableServer::_duplicate(poa)), _rel(release) {}
    virtual ~POA_example_tie() { /* . . . */ }
    T* _tied_object() { /* . . . */ }
    void _tied_object(T& t) { /* . . . */ }
    void _tied_object(T *p, CORBA::Boolean release=1) { /* . . . */ }
    CORBA::Boolean _is_owner() { /* . . . */ }
    void _is_owner(CORBA::Boolean b) { /* . . . */ }
    CORBA::Long op1(CORBA::Char _x, CORBA::Short_out _y) { /* . . . */ }
    PortableServer::POA_ptr _default_POA() { /* . . . */ }
};
```

_tie テンプレートクラスの使用方法については，第 11 章「tie メカニズムの使い方」を参照してください。

_ptie テンプレートを生成して, オブジェクトデータベースをサーバーと統合することもできます。

IDL のインターフェース属性の定義

インターフェース仕様では，インターフェースの一部としてオペレーションのほかに属性を定義できます。デフォルトでは，すべての属性が**読み取り／書き込み**用です。IDL コンパイラは，属性ごとに 2 つのメソッド（属性値を設定するメソッドと属性値を取得するメソッド）を生成します。また，**読み取り**専用の属性を指定することもできます。この場合は，取得用のメソッドだけが生成されます。

下の IDL サンプルは，2 つの属性を定義する IDL 仕様を示しています。1 つは読み書き用で，もう 1 つは読み取り専用です。

```
interface Test {
  attribute long count;
  readonly attribute string name;
};
```

このサンプルコードは，この IDL で宣言されたインターフェースに対して生成されたオペレーションクラスを示しています。

```
class test : public virtual CORBA::Object {
      ...
  // 読み取り／書き込み属性のためのメソッド
  virtual CORBA::Long count();
  virtual void count(CORBA::Long __count);
  // 読み取り専用属性のためのメソッド
  virtual char * name();
      ...
};
```

戻り値がない一方向メソッドの指定

IDL では，**一方向**メソッドと呼ばれる戻り値のないオペレーションを指定できます。これらのオペレーションには，入力パラメータだけがあります。oneway メソッドが呼び出されるとサーバーに要求が送信されますが，オブジェクトインプリメンテーションは，その要求が実際に受信されたかどうかの確認を返しません。

VisiBroker は，クライアントからサーバーへの接続に TCP ／ IP を使用します。これにより，すべてのパケットに信頼できる配信が提供されます。サーバーが使用可能な状態である限り，クライアントは確実に要求をサーバーに送信できます。ただし，クライアントは，実際にオブジェクトインプリメンテーション自身によって要求が処理されたかどうかを確認することはできません。

メモ　一方向オペレーションは，例外を生成したり値を返すことはできません。

```
interface oneway_example {
  oneway void set_value(in long val);
};
```

別のインターフェースを継承する IDL インターフェースの指定

IDL では，別のインターフェースを継承するインターフェースを指定できます。IDL コンパイラの生成したクラスは，継承関係を反映します。親インターフェースで宣言されたすべてのメソッド，データ型定義，定数，および列挙体は，派生したインターフェースに可視です。

```
interface parent {
  void operation1();
};
interface child : parent {
    . . .
  long operation2(in short s);
};
```

次のサンプルコードは，上のインターフェース仕様から生成されるコードを示します。

```
class parent : public virtual CORBA::Object {
    ...
  void operation1( );
    ...
};
class child : public virtual parent {
    ...
  CORBA::Long operation2(CORBA::Short s);
    ...
};
```

第 14 章

スマートエージェントの使い方

ここでは，スマートエージェント（osagent）について説明します。スマートエージェントは，オブジェクトインプリメンテーションを検索できるように，クライアントプログラムを登録します。カスタム VisiBroker ORB ドメインの設定方法，異なるローカルネットワーク上にあるスマートエージェントどうしの接続方法，および 1 つのホストから別のホストへのオブジェクトの移行方法について説明します。

スマートエージェントの概要

VisiBroker のスマートエージェント（osagent）は，動的な分散ディレクトリサービスであり，クライアントプログラムとオブジェクトインプリメンテーションの両方のために機能します。ローカルネットワーク内では，少なくとも 1 つのホストでスマートエージェントを起動する必要があります。クライアントプログラムがオブジェクトで bind() を呼び出すと，自動的にスマートエージェントが問い合わせを受けます。スマートエージェントは，指定されたインプリメンテーションを検索して，クライアントとそのインプリメンテーションとの間に接続を確立します。スマートエージェントとの通信は，クライアントプログラムに対して完全に透過的です。

POA で PERSISTENT ポリシーが設定されている場合，activate_object_with_id を使用すると，スマートエージェントは，クライアントプログラムがオブジェクトまたはインプリメンテーションを使用できるように，そのオブジェクトを登録します。オブジェクトまたはインプリメンテーションが非アクティブ化された場合，スマートエージェントは，使用可能なオブジェクトのリストからそのオブジェクトを削除します。スマートエージェントとの通信は，クライアントプログラムの場合と同様に，オブジェクトインプリメンテーションに対しても完全に透過的です。POA の詳細については，第 9 章「POA の使い方」を参照してください。

スマートエージェントの設定と同期に関する推奨事項

スマートエージェントがサポートするオブジェクトの数やタイプに厳しい制限はありませんが，スマートエージェントを大規模なアーキテクチャに組み込む場合にしたがうとよい推奨の方法があります。

スマートエージェントは，フラットで単純な名前空間で構成された軽量なディレクトリサービスとして設計されており，ローカルネットワーク内の少数の既知のオブジェクトをサポートできます。

すべてのオブジェクトの登録済みのサービスはメモリに格納されるため，スケーラビリティの最適化とフォールトトレランスを同時に満たすことはできません。アプリケーションは，すべてのディレクトリ要求をスマートエージェントに依存しないように，既知のオブジェクトを使って別の分散サービスにブートストラップする必要があります。必要とするサービス検索の負荷が大きい場合は，VisiBroker ネーミングサービス（VisiNaming）を使用することをお勧めします。VisiNaming には永続的ストレージ機能とクラスタ負荷分散機能がありますが，スマートエージェントでは osagent 単位の単純なラウンドロビンだけが提供されます。スマートエージェントはインメモリ設計なので，適切にシャットダウンした場合でも異常終了した場合でも，同じ ORB ドメイン内の別のスマートエージェント（同じ OSAGENT_PORT 番号）にフェイルオーバーすることがありません。これに対して，VisiNaming サービスは，そのようなフェイルオーバー機能を提供します。VisiBroker ネーミングサービスの詳細については，第 16 章「VisiNaming サービスの使い方」を参照してください。

全般的なガイドライン

スマートエージェントの使い方として推奨される一般的なガイドラインを次に示します。

- サーバー登録数は, ORB ドメインあたりのオブジェクトインスタンス数または POA 数で 100 未満に制限します。
- スマートエージェントは，CORBA サーバーだけでなくすべてのクライアントを追跡するため，スマートエージェントには各クライアントの負荷が少量ずつかかります。任意の 10 分間に，クライアント数が 100 を超えないようにします。

メモ　GateKeeper は，実際には多くのクライアントのかわりとして機能しても，1 つのクライアントとしてカウントします。

- アプリケーションは，起動時に少数の既知のオブジェクトにまとめてバインドし，これらのオブジェクトを後で検索に使用することで，スマートエージェントを分散的に使用する必要があります。スマートエージェントの通信は UDP ベースです。UDP 上に構築されたメッセージプロトコルの信頼性は高いですが, UDP は一般に信頼性が低く, ワイドエリアネットワークでは使用されません。スマートエージェントはイントラネット向けに設計されているため，ファイアウォール構成を含むワイドエリアネットワークでは使用をお勧めしません。
- スマートエージェントホストに直接接続されていないサブネット上のクライアントからスマートエージェントの実際のデフォルト IP にアクセスできる必要があります。ネットワークアドレス変換（NAT）ファイアウォールの背後からのクライアントアクセス用にスマートエージェントを設定することはできません。
- スマートエージェントは，起動時に入手できるネットワーク情報を使用して，自分自身を設定します。スマートエージェントは, ダイヤルアップ接続に関連付けられたインターフェースなど, 後で追加された新しいネットワークインターフェースを検出できません。したがって，スマートエージェントは，静的なネットワーク構成に適しています。

負荷分散とフォールトトレランスのガイドライン

- スマートエージェントは，ORB ドメインベースではなくエージェントベースで単純なラウンドロビンアルゴリズムを使用して，負荷分散を実装します。ORB ドメインに複数のスマートエージェントがある場合，複製サーバー間で負荷分散を実現するには，すべてのサーバーを同じスマートエージェントに登録します。
- ORB ランタイムはスマートエージェントへのアクセスをキャッシュするため，同じ ORB プロセスから同じサーバーオブジェクトに複数のバインドを実行しても，ラウンドロビン動作は行われません。後の方のオブジェクトへのバインドでは，スマートエージェントに新しい要求が送信されず，キャッシュが使用されます。この動作は，ORB の

プロパティで変更できます。詳細については，『*VisiBroker プログラマーズリファレンス*』の第 6 章「VisiBroker のプロパティ」を参照してください。

- スマートエージェントが終了すると，そのエージェントに登録されていたすべてのサーバーは，別のエージェントを探して登録を試みます。このプロセスは自動的に実行されますが，サーバーがこれを実行するために最大 2 分かかる場合があります。その 2 分の間，サーバーは ORB ドメインに登録されないため，新しいクライアントがサーバーを使用できなくなります。ただし，すでにバインドされているサーバーとクライアントの間で進行中の IIOP 通信には影響しません。

ロケーションサービスのガイドライン

ロケーションサービスは，スマートエージェント技術の上に構築されています。したがって，ロケーションサービスも上と同じガイドラインにしたがいます。

- ロケーションサービスのトリガーにより，アプリケーションによって登録されたトリガーハンドラとスマートエージェントの間に UDP トラフィックが発生します。この機能を使用するには，オブジェクトの数と監視プロセスの数をそれぞれ 10 未満に制限する必要があります。
- ロケーションサービスのトリガーは，オブジェクトが動作している，またはダウンしていることをスマートエージェントが確認するときに発行されます。「ダウン」トリガーの発行は，最大 4 分遅延する場合があります。このため，タイムクリティカルなアプリケーションでは，この機能の使用を控える必要があります。

ロケーションサービスの詳細については，第 15 章「ロケーションサービスの使い方」を参照してください。

スマートエージェントを使用しない場合

- ORB ドメインが多くの（6 以上）サブネットにまたがる場合。多くのサブネットにまたがる大規模な ORB ドメインでは，`agentaddr` ファイルの管理が困難です。
- 名前空間が多くの（100 を超える）既知のオブジェクトを必要とする場合。
- スマートエージェントを必要とするアプリケーション（クライアント）の数が 10 分間に常時 100 を超える場合。

メモ　上の状況では，ネーミングサービスなどの別のディレクトリが適切です。詳細については，第 16 章「VisiNaming サービスの使い方」を参照してください。

スマートエージェントの検索

VisiBroker は，クライアントプログラムまたはオブジェクトインプリメンテーションが使用するスマートエージェントを検索するために，ブロードキャストメッセージを使用します。最初に応答したスマートエージェントが使用されます。スマートエージェントが見つかると，ポイントツーポイント UDP 接続で，そのスマートエージェントに登録および検索要求を送信します。

UDP プロトコルが使用されるのは，TCP 接続より消費するネットワークリソースが少ないからです。すべての登録および検索要求は動的に行われるので，設定ファイルやマッピングを維持する必要はありません。

メモ　ブロードキャストメッセージは，スマートエージェントの検索専用です。その他のスマートエージェントとの通信では，ポイントツーポイント通信が使用されます。ブロードキャストメッセージの使い方を上書きする方法については，162 ページの「ポイントツーポイント通信の使い方」を参照してください。

スマートエージェント間の協力によるオブジェクトの検索

スマートエージェントがローカルネットワーク上の複数のホストで起動されると，各スマートエージェントは，利用できるオブジェクトのサブセットを認識します。また，ほか

のスマートエージェントと通信して，発見できないオブジェクトを検索します。スマートエージェントのプロセスの 1 つが突然終了すると，そのスマートエージェントに登録されていたすべてのインプリメンテーションがこのイベントを発見し，使用可能な別のスマートエージェントに自動的に登録されます。

OAD との協力によるオブジェクトへの接続

オブジェクトインプリメンテーションをオンデマンドで起動するために，オブジェクトインプリメンテーションをオブジェクトアクティベーションデーモン（OAD）に登録することができます。このようなオブジェクトは，OAD 内でオブジェクトが実際にアクティブであって検索が可能な場合と同様に，スマートエージェントに登録されています。クライアントがこれらのオブジェクトの 1 つを要求すると，その要求は OAD に送信されます。次に，OAD は，そのクライアント要求を*実際の*サーバーに転送します。オブジェクトインプリメンテーションが実際には OAD 内でアクティブでないことを，スマートエージェントは認識しません。OAD の詳細については，第 20 章「オブジェクトアクティベーションデーモン（OAD）の使い方」を参照してください。

スマートエージェント（osagent）の起動

ローカルネットワーク内のホストで，少なくとも 1 つのスマートエージェントのインスタンスを実行しておく必要があります。ローカルネットワークは，ブロードキャストメッセージを送信できる範囲内のサブネットワークを参照します。

Windows： スマートエージェントを起動するには，次の手順にしたがいます。

- 次のディレクトリにある osagent 実行可能ファイル osagent.exe をダブルクリックします。

  ```
  <install_dir¥bin¥
  ```

  または

- コマンドプロンプトで，「osagent [options]」と入力します。次に例を示します。

  ```
  prompt> osagent [options]
  ```

UNIX： スマートエージェントを起動するには，「osagent &」と入力します。次に例を示します。

```
prompt> osagent &
```

メモ シグナル処理が変更されたため，bourne シェルと korn シェルのユーザーは，ユーザーがログアウトする際にハングアップ（hup）シグナルによってプロセスが終了しないように，osagent の起動時に ignoreSignal hup パラメータを使用する必要があります。次に例を示します。

```
nohup $VBROKERDIR/bin/osagent ignoreSignal hup &
```

osagent コマンドは，次のコマンドライン引数を受け取ります。

オプション	説明
-a <IP_address>	デフォルトの監視アドレスを設定します。
-p <UDP_port>	OSAGENT_PORT の設定とレジストリ設定を上書きします。
-v	詳細モードにして，実行中の情報および診断メッセージを表示します。
-help または -?	ヘルプメッセージを出力します。
-l	OSAGENT_LOGGING_ON が設定されている場合はログをオフにします。
-ls <size>	調整ログサイズを 1024 KB ブロック単位で指定します。最大値は 300 です。したがって，最大ログサイズは 300 MB です。

オプション	説明
+l <options>	ログレベルを表示／有効にします。サポートされるオプションは次のとおりです。 • ログをオンにし，ログレベル "ief" (== +l oief) を有効にします。これは，OSAGENT_LOGGING_ON の設定と同等です。ログは自動的にサイズ調整され，OSAGENT_LOG_DIR ディレクトリまたは VBROKER_ADM ディレクトリに書き込まれます（設定されている場合）。設定されていない場合は，デフォルトで /tmp（UNIX）または %TEMP%（Windows）に書き込まれます。 • i - 情報 • e - エラー • w - 警告 • f - 致命的エラー • d - デバッグ • a - すべて
-n，-N	Windows のシステムトレイアイコンを無効にします。

例

osagent コマンドの次の例では，特定の UDP ポートを指定します。

```
osagent -p 17000
```

詳細出力

UNIX： UNIX では，詳細出力は stdout に送信されます。

Windows： Windows では，詳細出力は次のいずれかの場所にあるログファイルに書き込まれます。

- C:¥TEMP¥vbroker¥log¥osagent.log.
- 環境変数 VBROKER_ADM によって指定されたディレクトリ

メモ　ログファイルの書き込み先として別のディレクトリを指定するには，OSAGENT_LOG_DIR を使用します。ログオプションを設定するには，スマートエージェントのアイコンを右クリックし，［Log Options］を選択します。

エージェントを無効にする

実行時に次のプロパティを VisiBroker ORB に渡すことにより，スマートエージェントとの通信を無効にできます。

```
prompt> Server -Dvbroker.agent.enableLocator=false
```

文字列リファレンスまたはネーミングサービスを使用しているか，URL リファレンスを渡す場合，スマートエージェントは不要なので無効にできます。bind() メソッドにオブジェクト名を渡す場合は，スマートエージェントを使用する必要があります。

スマートエージェントの有効性の確認

ローカルネットワーク内の複数のホストでスマートエージェントを起動すると，スマートエージェントの 1 つが予期せずに終了しても，クライアントはオブジェクトにバインドし続けることができます。スマートエージェントが利用できなくなると，そのスマートエージェントに登録されていたすべてのオブジェクトインプリメンテーションは，別のスマートエージェントに自動的に再登録されます。ローカルネットワークでスマートエージェントが動作していない場合，オブジェクトインプリメンテーションは，新しいスマートエージェントと交信できるまで再試行を続けます。

スマートエージェントが終了すると，終了前に確立されていた任意のクライアントとオブジェクトインプリメンテーションとの間の接続は，中断なく継続されます。ただし，クライアントが任意の新しい bind() 要求を発行すると，新しいスマートエージェントが起動します。

これらのフォールトトレランス機能を利用するには，特別なコーディング技術は必要ありません。ローカルネットワークの 1 つ以上のホストで，スマートエージェントが起動されていることを確認する必要があるだけです。

クライアントの確認

スマートエージェントは,「Are You Alive」メッセージ（ハートビートメッセージともいう）を 2 分ごとにクライアントに送信して，クライアントがまだ接続されているかどうかを確認します。クライアントが応答しない場合，スマートエージェントは，クライアントが接続を終了したとみなします。

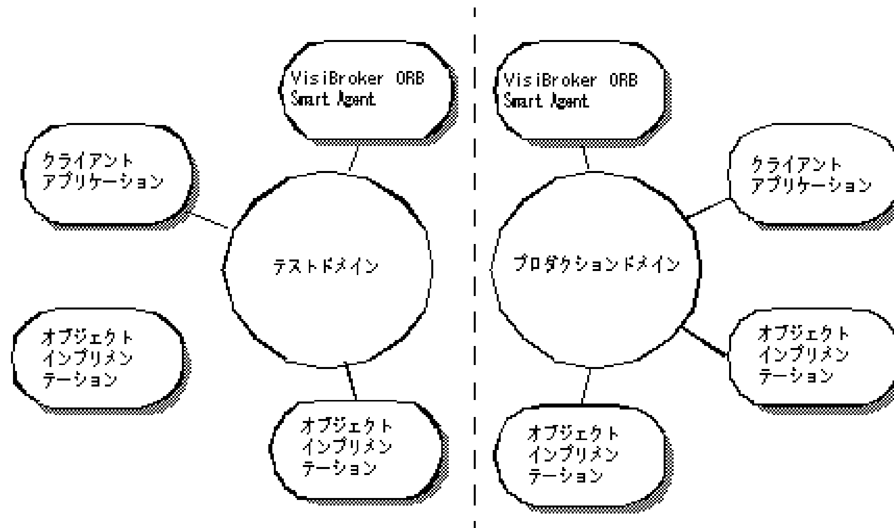
クライアントをポーリングする間隔は変更できません。

メモ　「クライアント」という用語は，必ずしもオブジェクトまたはプロセスの機能を指していません。オブジェクトリファレンスを取得するためにスマートエージェントに接続する任意のプログラムをクライアントと呼びます。

VisiBroker ORB ドメインの操作

同時に複数の VisiBroker ORB ドメインを実行すると便利なことがあります。たとえば，一方のドメインを製品版のクライアントプログラムとオブジェクトインプリメンテーションで構成し，もう一方のドメインを一般の使用に向けてまだリリースされていないテスト版で構成します。何人かの開発者が同じローカルネットワーク上で作業している場合は，テスト作業が互いに競合しないように，それぞれが独自の VisiBroker ORB ドメインを確立することができます。

図 14.1　異なる VisiBroker ORB ドメインの同時実行



VisiBroker では，各ドメインのスマートエージェントに一意の UDP ポート番号で，同じネットワーク上にある複数の ORB ドメインを区別できます。デフォルトでは，`OSAGENT_PORT` 変数は 14000 に設定されています。別のポート番号を使用する場合は，使用できるポート番号をシステム管理者に問い合わせてください。

デフォルトの設定を上書きするには，VisiBroker ORB ドメインに割り当てられているスマートエージェント，OAD，オブジェクトインプリメンテーション，およびクライアントプログラムを実行する前に，`OSAGENT_PORT` 変数を適切に設定する必要があります。たとえば，次のようにします。

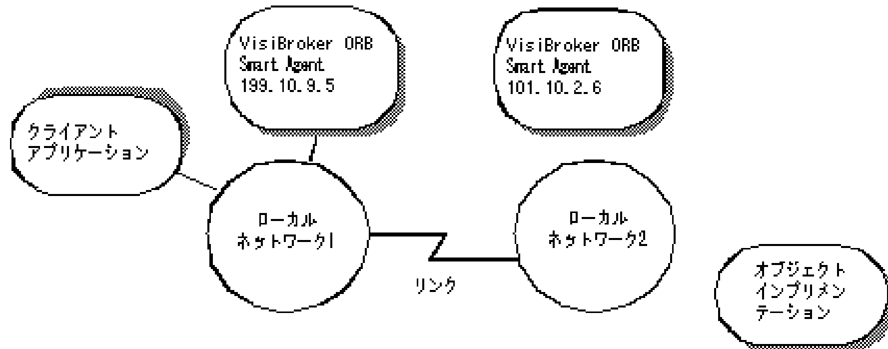
```
prompt> setenv OSAGENT_PORT 5678
prompt> osagent &
prompt> oad &
```

スマートエージェントは，TCP プロトコルと UDP プロトコルの両方に対して追加の内部ポート番号を使用します。ただし，ポート番号はともに同じです。このポート番号は，OSAGENT_CLIENT_HANDLER_PORT 環境変数で設定できます。

異なるローカルネットワーク上のスマートエージェントの接続

ローカルネットワークで複数のスマートエージェントを起動すると，各スマートエージェントは，UDP ブロードキャストメッセージで互いを見つけます。ネットワークアドミニストレータは，IP サブネットマスクでブロードキャストメッセージの範囲を指定してローカルネットワークを設定します。次の図はネットワークリンクによって接続された 2 つのローカルネットワークです。

図 14.2　異なるローカルネットワーク上の 2 つのスマートエージェント



あるネットワーク上のスマートエージェントが別のローカルネットワーク上のスマートエージェントと通信できるようにするには，次の例に示すように，OSAGENT_ADDR_FILE 環境変数を使用します。

```
setenv OSAGENT_ADDR_FILE=<path to agent addr file>
```

または，次の例に示すように vbroker.agent.addrFile プロパティを使用します。

```
vbj -Dvbroker.agent.addrFile=<path to agent addr file> ....
```

次の例は，ローカルネットワーク #1 のスマートエージェントが別のローカルネットワークのスマートエージェントに接続するための agentaddr ファイルの内容を示しています。

```
101.10.2.6
```

適切な agentaddr ファイルを使用して，ネットワーク #1 上のクライアントプログラムは，ネットワーク #2 上のオブジェクトインプリメンテーションを見つけて使用します。環境変数の詳細については，「Borland VisiBroker インストールガイド」を参照してください。

メモ　リモートネットワーク上で複数のスマートエージェントが実行されている場合は，リモートネットワーク上のスマートエージェントのすべての IP アドレスをリストしてください。

スマートエージェントが互いを検出する方法

たとえば，2 つのエージェント（エージェント 1 とエージェント 2）が，同じサブネット上の 2 つのマシンから同じ UDP ポートを監視している状況を考えてください。エージェント 1 は，エージェント 2 より先に起動します。その後のイベントは，次のようになります。

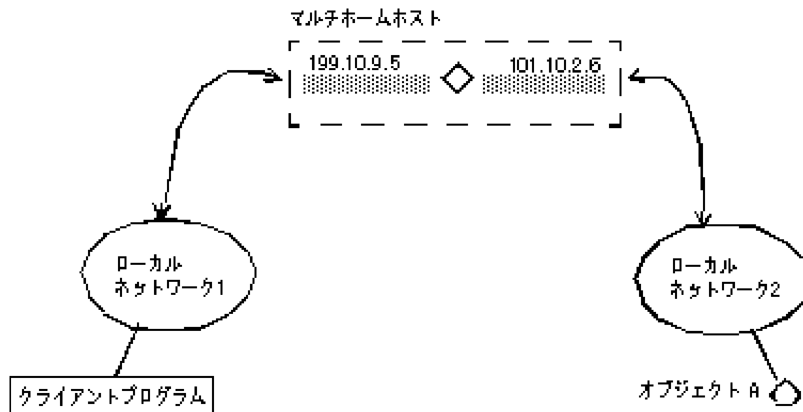
- エージェント 2 が起動すると，その存在を UDP ブロードキャストし，その他のスマートエージェントを検索するために要求メッセージを送信します。
- エージェント 1 は，エージェント 2 がネットワーク上で使用可能であることを認識し，要求メッセージに応答します。
- エージェント 2 は，ほかのエージェント（エージェント 1）がネットワーク上で使用可能であることを認識します。

〔Ctrl〕＋〔C〕などの終了操作でエージェント 2 を正常に終了すると，エージェント 2 が利用できなくなったことがエージェント 1 に通知されます。

マルチホームホストのしくみ

複数の IP アドレスを持つホスト（マルチホームホスト）でスマートエージェントを起動すると，異なるローカルネットワークに配置されたオブジェクトどうしをブリッジする強力なメカニズムを提供できます。このホストに接続されているすべてのローカルネットワークが単一のスマートエージェントと通信できるようになり，効率的にローカルネットワークがブリッジされます。

図 14.3　マルチホームホスト上のスマートエージェント



UNIX： マルチホームの UNIX ホストでは，スマートエージェントは自分自身を動的に設定して，ポイントツーポイント接続やブロードキャスト接続をサポートするすべてのホストのインターフェースで監視およびブロードキャストを行います。161 ページの「スマートエージェントのインターフェースの用法の指定」で説明している `localaddr` ファイルを使用すると，明示的にインターフェース設定を指定できます。

Windows： マルチホームの Windows ホストでは，スマートエージェントは正しいサブネットマスクとブロードキャストアドレス値を動的に判定できません。この制約に対処するには，`localaddr` ファイルで，スマートエージェントが使用するインターフェース設定を明示的に指定する必要があります。

`-v`（verbose）オプションでスマートエージェントを起動すると，生成されるメッセージの先頭に，スマートエージェントが使用するインターフェースが表示されます。次のサンプルは，マルチホームホストの詳細オプションで起動したスマートエージェントのサンプル出力を示しています。

```
Bound to the following interfaces:
Address: 199.10.9.5 Subnet: 255.255.255.0 Broadcast:199.10.9.255
Address: 101.10.2.6 Subnet: 255.255.255.0 Broadcast:101.10.2.255
...
```

上のように，出力には，マシンの各インターフェースに対応するアドレス，サブネットマスク，ブロードキャストアドレスがあります。

UNIX： 上の出力は，UNIX コマンド `ifconfig -a` の結果と一致する必要があります。

これらの設定を上書きする場合は，`localaddr` ファイルにインターフェース情報を設定します。詳細については，次の161 ページの「スマートエージェントのインターフェースの用法の指定」を参照してください。

スマートエージェントのインターフェースの用法の指定

メモ　シングルホームホストでは，インターフェース情報を指定する必要はありません。

マルチホームホストでスマートエージェントが使用する各インターフェースのインターフェース情報は，localaddr ファイルで指定できます。localaddr ファイルには，各インターフェースの行にホストの IP アドレス，サブネットマスク，およびブロードキャストアドレスを記述する必要があります。VisiBroker は，デフォルトで VBROKER_ADM ディレクトリの localaddr ファイルを検索します。この場所を上書きするには，OSAGENT_LOCAL_FILE 環境変数の宛先をこのファイルに設定します。このファイルで，先頭が「#」文字の行はコメントとして扱われ，無視されます。次のサンプルコードは，上のマルチホームホストの localaddr ファイルです。

```
#entries of format <address> <subnet_mask> <broadcast address>
199.10.9.5 255.255.255.0 199.10.9.255
101.10.2.6 255.255.255.0 101.10.2.255
```

UNIX：　UNIX 上のマルチホームホストでは，スマートエージェントが自動的に設定されますが，localaddr ファイルで，ホストが持つインターフェースを明示的に指定することもできます。UNIX ホストで利用できるすべてのインターフェースの値は，次のコマンドで表示できます。

```
prompt> ifconfig -a
```

このコマンドの出力は，次のように表示されます。

```
lo0: flags=849<UP,LOOPBACK,RUNNING,MULTICAST> mtu 8232
      inet 127.0.0.1 netmask ff000000
le0: flags=863<UP,BROADCAST,NOTRAILERS,RUNNING,MULTICAST> mtu 1500
      inet 199.10.9.5 netmask ffffff00 broadcast 199.10.9.255
le1: flags=863<UP,BROADCAST,NOTRAILERS,RUNNING,MULTICAST> mtu 1500
      inet 101.10.2.6 netmask ffffff00 broadcast 101.10.2.255
```

Windows：　Windows を実行しているマルチホームホストでは，スマートエージェントが自動的に設定されないので，localaddr ファイルを使用する必要があります。［Network Control Panel］から TCP/IP プロトコルのプロパティにアクセスすると，このファイルに記述する適切な値を取得できます。ホストで Windows が動作している場合は，ipconfig コマンドで必要な値を取得できます。このコマンドは，次のように実行します。

```
prompt> ipconfig
```

このコマンドの出力は，次のように表示されます。

```
Ethernet adapter El90x1:

      IP Address. . . . . . . . . : 172.20.30.56
      Subnet Mask . . . . . . . . : 255.255.255.0
      Default Gateway . . . . . . : 172.20.0.2
Ethernet adapter Elnk32:
      IP Address. . . . . . . . . : 101.10.2.6
      Subnet Mask . . . . . . . . : 255.255.255.0
      Default Gateway . . . . . . : 101.10.2.1
```

ポイントツーポイント通信の使い方

VisiBroker では，UDP ブロードキャストメッセージを使用しないでスマートエージェントのプロセスを検索する方法を 3 とおり用意しています。これらの方法の 1 つでスマートエージェントが検索されると，そのスマートエージェントがそれ以降のすべての対話で使用されます。これらの方法でスマートエージェントを検索できない場合，VisiBroker は，ブロードキャストメッセージ方式に戻ってスマートエージェントを検索します。

実行時パラメータとしてのホストの指定

次のサンプルコードは，スマートエージェントが実行する場所をクライアントプログラムやオブジェクトインプリメンテーションの実行時パラメータとして IP アドレスで指定する方法を示しています。IP アドレスを指定すると，ポイントツーポイント接続が確立されるので，ローカルネットワークの外部にあるホストの IP アドレスを指定することもできます。この方法は，ほかのホストの指定方法より優先されます。

```
prompt> Server -Dvbroker.agent.addr=<ip_address>
```

また, プロパティファイルを使って IP アドレスを指定することもできます。vbroker.agent.addr エントリを見つけてください。

```
vbroker.agent.addr=<ip_address>
```

デフォルトで，プロパティファイルの vbroker.agent.addr は NULL に設定されます。

エージェントが常駐するホスト名をリストして，そのファイルをプロパティファイルの vbroker.agent.addrFile オプションで指定することもできます。

環境変数による IP アドレスの指定

クライアントプログラムまたはオブジェクトインプリメンテーションを起動する前に，OSAGENT_ADDR 環境変数を設定して，スマートエージェントの IP アドレスを指定できます。ホストが実行時パラメータとして指定されていない場合は，この環境変数が優先します。

UNIX：

```
prompt> setenv OSAGENT_ADDR 199.10.9.5
prompt> client
```

Windows：Windows システムで OSAGENT_ADDR 環境変数を設定するには，[System control panel] で環境変数を編集します。

1 [System Variables] から，任意の現在の変数を選択します。

2 [Variable] 編集ボックスに「OSAGENT_ADDR」と入力します。

3 [Value] 編集ボックスに IP アドレスを入力します。例：199.10.9.5。

agentaddr ファイルによるホストの指定

クライアントプログラムまたはオブジェクトインプリメンテーションは，UDP ブロードキャストメッセージのかわりに agentaddr ファイルを使用して，スマートエージェントを見つけます。それには，スマートエージェントが動作している各ホストの IP アドレスまたは完全なホスト名を記述したファイルを作成します。次に，OSAGENT_ADDR_FILE 環境変数を設定してそのファイルへのパスを指示します。クライアントプログラムまたはオブジェクトインプリメンテーションにこの環境変数が設定されている場合，VisiBroker は，スマートエージェントが見つかるまでこのファイル内の各アドレスを試します。このメカニズムは，ホストを指定するすべてのメカニズムの中で最も優先順位が低いものです。このファイルを指定しなければ，VBROKER_ADM/agentaddr ファイルが適用されます。

オブジェクトの有効性の確認

オブジェクトのインスタンスを複数のホストで起動することにより，それらのオブジェクトにフォールトトレランスを提供することができます。1 つのインプリメンテーションが無効になると，VisiBroker ORB は，クライアントプログラムとオブジェクトインプリメンテーションの間の接続が失われたことを検出します。そして，自動的にスマートエージェントとコンタクトし，クライアントによって確立されている有効なリバインドポリシーに基づいて，そのオブジェクトインプリメンテーションの別のインスタンスと接続を確立します。クライアントポリシーの確立の詳細については，「クライアントの基礎」の第 12 章「クライアントの基礎」を参照してください。

メモ　スマートエージェントは，ORB ドメインベースではなくエージェントベースで単純なラウンドロビンアルゴリズムを使用して，負荷分散を実装します。ORB ドメインに複数のスマートエージェントがある場合, 複製サーバー間で負荷分散を実現するには, すべてのサーバーを同じスマートエージェントに登録します。

重要　VisiBroker がクライアントをインスタンスオブジェクトインプリメンテーションに再接続する場合，リバインドオプションを有効にする必要があります。この再接続がデフォルトの動作です。

状態を保持しないオブジェクトのメソッドの呼び出し

クライアントプログラムは，そのオブジェクトの新しいインスタンスが使用されるかどうかに関係なく，状態を保持しないオブジェクトインプリメンテーションのメソッドを呼び出すことができます。

状態を保持するオブジェクトのフォールトトレランスの実現

状態を保持するオブジェクトインプリメンテーションでもフォールトトレランスを実現できます。ただし，これはクライアントプログラムに透過的ではありません。この場合，クライアントプログラムは Quality of Service（QoS）ポリシーの VB_NOTIFY_REBIND を使用するか，VisiBroker または，ORB オブジェクトのインターセプタを登録する必要があります。QoS の使用方法については，第 12 章「クライアントの基礎」を参照してください。

オブジェクトインプリメンテーションへの接続が失敗し，VisiBroker がクライアントをオブジェクトインプリメンテーションの複製に再接続するとき，VisiBroker によってバインドインターセプタの bind メソッドが呼び出されます。この bind メソッドのインプリメンテーションを提供して，複製の状態を最新に保つのはクライアントの役目です。クライアントインターセプタについては，第 25 章「VisiBroker インターセプタの使い方」を参照してください。

OAD に登録されたオブジェクトの複製

いったんオブジェクトの起動が失敗しても，OAD によってそのオブジェクトが再起動されるので，オブジェクトの可用性が確実に高まります。ホストが利用できなくなるような状況でフォールトトレランスが必要な場合は，OAD を複数のホストで起動し，オブジェクトを各 OAD のインスタンスに登録します。

メモ　VisiBroker によって提供されるオブジェクトの複製には，マルチキャスト機能やミラーリング機能が提供されません。クライアントプログラムと特定のオブジェクトインプリメンテーションとの間には，常に 1 対 1 の対応があります。

ホスト間のオブジェクトの移行

オブジェクトの移行とは，あるホストでオブジェクトインプリメンテーションを終了し，別のホストでそれを起動する処理です。オブジェクトの移行を利用すると，負荷のかかりすぎたホストから，リソースや処理能力に余裕のあるホストにオブジェクトを移動して，負荷を分散できます（ただし，異なるスマートエージェントに登録されているサーバー間では負荷を分散できません）。ハードウェアまたはソフトウェアの保守のためにホストをシャットダウンする場合にも，オブジェクトを移行することにより，そのオブジェクトを継続して使用できます。

メモ 状態を保持しないオブジェクトの移行は，クライアントプログラムには透過的に行われます。移行されたオブジェクトインプリメンテーションにクライアントが接続すると，スマートエージェントが失われた接続を検出し，クライアントを新しいホストの新しいオブジェクトに透過的に再接続します。

状態を保持するオブジェクトの移行

状態を保持するオブジェクトの移行もできますが，移行処理が開始される前に接続していたクライアントプログラムには透過的には行われません。このような場合，クライアントプログラムは，そのオブジェクトのインターセプタを登録する必要があります。

元のオブジェクトとの接続が失われ，VisiBroker がクライアントをそのオブジェクトに再接続すると，VisiBroker によってインターセプタの `rebind_succeeded()` メソッドが呼び出されます。クライアントは，このメソッドを実行して，オブジェクトの状態を最新に保つことができます。

インターセプタの使用方法については，第 24 章「ポータブルインターセプタの使い方」を参照してください。

インスタンス化されたオブジェクトの移行

移行するオブジェクトが，インプリメンテーションのクラスをインスタンス化するサーバープロセスで作成済みの場合，必要なのは，新しいホストでそのオブジェクトを起動し，そのサーバープロセスを終了するだけです。元のインスタンスが終了すると，スマートエージェントへの登録が解除されます。新しいホストで新しいインスタンスが起動すると，スマートエージェントに登録されます。その時点から，クライアントによる呼び出しが新しいホストのオブジェクトインプリメンテーションに送信されます。

OAD に登録されたオブジェクトの移行

移行する VisiBroker オブジェクトが OAD に登録されている場合は，最初に元のホストの OAD の登録を解除する必要があります。次に，そのオブジェクトを新しいホストの OAD に登録します。

すでに OAD に登録されているオブジェクトを移行するには，次の手順を実行します。

1. 元のホストの OAD からオブジェクトインプリメンテーションの登録を解除します。
2. 新しいホストの OAD にオブジェクトインプリメンテーションを登録します。
3. 元のホストのオブジェクトインプリメンテーションを終了します。

オブジェクトインプリメンテーションの登録と登録解除の詳細については，第 20 章「オブジェクトアクティベーションデーモン（OAD）の使い方」を参照してください。

すべてのオブジェクトとサービスのレポート

Smart Finder（osfind）コマンドは，指定されたネットワークで現在使用可能な VisiBroker 関連のすべてのオブジェクトおよびサービスをレポートします。

osfind を使用すると，ネットワーク上で実行されているスマートエージェントのプロセス数，および稼動中の正確なホストを調べることができます。また，オブジェクトがスマートエージェントに登録されている場合には，osfind コマンドはネットワーク上で有効なすべての VisiBroker オブジェクトもレポートします。osfind を使用すれば，ネットワークの状態を監視したり，デバッグ中に孤立したオブジェクトを検索することができます。

osfind コマンドの構文は次のとおりです。

```
osfind [options]
```

osfind では，次のオプションを使用できます。オプションを何も指定しない場合，osfind は，ドメイン内のすべてのエージェント，OAD，およびインプリメンテーションをリストします。

オプション	説明
-a	ドメイン内のすべてのスマートエージェントをリストします。
-b	VisiBroker 2.0 下位互換 osfind メカニズムを使用します。
-d	ホスト名を IP アドレスで出力します。
-f <agent_address_file_name>	ファイルで指定したホスト上で実行されているスマートエージェントを照会します。このファイルには，1 行ごとに 1 つの IP アドレスまたは完全なホスト名が記述されています。すべてのスマートエージェントをレポートする場合にはこのファイルは適用されません。つまり，オブジェクトのインプリメンテーションおよびサービスをレポートする場合にだけこのファイルは適用されます。
-g	オブジェクトの存在を確認します。この検査は，ロード済みシステムに相当な遅れを引き起こします。存在を確認する対象は，BY_INSTANCE に登録されているオブジェクトだけです。OAD または BY_POA ポリシーに登録されているオブジェクトの存在は確認の対象外です。
-h, -help, -usage, -?	このオプションのヘルプ情報を出力します。
-o	ドメイン内のすべての OAD をリストします。
-p	同じホストでアクティブ化されたすべての POA インスタンスをリストします。このオプションを指定しない場合，一意の POA 名だけがリストされます。

Windows： osfind はコンソールアプリケーションです。［スタート］メニューから osfind を起動すると，そのまま最後まで実行され，結果を確認する前に処理が終了してしまいます。

オブジェクトへのバインド

クライアントアプリケーションでインターフェースのメソッドを呼び出すには，bind() メソッドでオブジェクトリファレンスを取得しておく必要があります。

クライアントアプリケーションが bind() メソッドを呼び出すと，VisiBroker はアプリケーションにかわっていくつかの処理を実行します。その内容は次のとおりです。

- VisiBroker は，osagent にアクセスし，要求されたインターフェースを提供するオブジェクトサーバーを探します。オブジェクト名とホスト名（または IP アドレス）を指定した場合は，その名前で，ディレクトリサービスの検索条件が限定されます。
- オブジェクトインプリメンテーションが見つかると，VisiBroker は，見つかったオブジェクトインプリメンテーションとクライアントアプリケーションを接続します。
- 正しく接続が確立されると，VisiBroker は，必要に応じてプロキシオブジェクトを作成し，そのオブジェクトへのリファレンスを返します。

メモ　VisiBroker は独立したプロセスではありません。VisiBroker は，クライアントとサーバーとの通信に利用されるクラスとリソースの集まりです。

第 15 章

ロケーションサービスの使い方

VisiBroker ロケーションサービスは，特定の属性に基づいてオブジェクトインスタンスを検索する高度なオブジェクト検索機能です。ロケーションサービスは，VisiBroker スマートエージェントとともに機能し，ネットワーク上にある現在アクセス可能なオブジェクト，およびそのオブジェクトの場所を通知します。ロケーションサービスは，CORBA 仕様を拡張した VisiBroker の機能であり，VisiBroker を使って実装されたオブジェクトの検索にだけ利用できます。スマートエージェント（osagent）の詳細については，第 14 章「スマートエージェントの使い方」を参照してください。

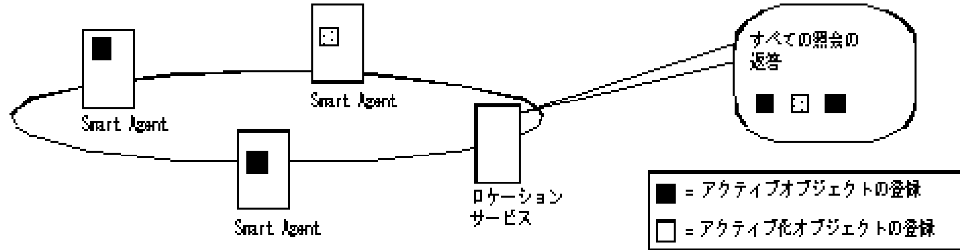
ロケーションサービスの概要

ロケーションサービスは，オブジェクトインスタンスを検索するために，CORBA 仕様が提供する汎用の機能を拡張したものです。ロケーションサービスは，スマートエージェントの 1 つと直接通信します。スマートエージェントは，それぞれ 1 つの「カタログ」を維持しており，そのカタログには，スマートエージェントが認識しているインスタンスのリストが保持されます。ロケーションサービスから照会があると，スマートエージェントは，ほかのすべてのスマートエージェントにその照会を転送し，応答を集計してその結果をロケーションサービスに返します。

ロケーションサービスは，BY_INSTANCE ポリシーを使って POA で登録されているすべてのオブジェクトインスタンス，および BOA で永続的として登録されているオブジェクトを認識します。これらのオブジェクトを持つサーバーは，手動で起動されるか，OAD によって自動的に起動されます。詳細については，第 9 章「POA の使い方」第 31 章「VisiBroker における BOA の使い方」，および第 20 章「オブジェクトアクティベーションデーモン（OAD）の使い方」を参照してください。

次に，このしくみについて図解します。

図 15.1　スマートエージェントを使ったオブジェクトインスタンスの検索



メモ　サーバーは，インスタンスを作成するとき，そのインスタンスのスコープを指定します。グローバルなスコープを持つインスタンスだけがスマートエージェントに登録されます。

ロケーションサービスは，スマートエージェントが各オブジェクトインスタンスに関して保持する情報を活用できます。ロケーションサービスは，オブジェクトインスタンスごとに，下に示す ObjLocation::Desc 構造体にその情報をカプセル化して維持します。

```
struct Desc {
  Object ref;
  ::IIOP::ProfileBodyValue iiop_locator;
  string repository_id;
  string instance_name;
  boolean activable;
  string agent_hostname;
};
typedef sequence<Desc>   DescSeq;
```

Desc 構造体の IDL には，次の情報が含まれます。

- ref はオブジェクトリファレンス，つまりオブジェクトを呼び出すためのハンドルです。
- iiop_locator インターフェースは，インスタンスのサーバーのホスト名とポートにアクセスするために使用されます。この情報は，オブジェクトが IIOP を使って接続されている場合にだけ意味があります。サポートされるのは IIOP だけです。ホスト名は，インスタンスの記述内の文字列として返されます。
- repository_ id はリポジトリ ID，つまりオブジェクトインスタンスのインターフェース名です。インターフェースリポジトリおよびインプリメンテーションリポジトリ内でこれが検索されます。1 つのインスタンスが複数のインターフェースを備えている場合は，各インターフェースごとに 1 つのインスタンスがあるかのように，各インターフェースのエントリがカタログに保持されます。
- instance_name は，サーバーによってオブジェクトに与えられる名前です。
- activable フラグは，OAD によってアクティブ化されるインスタンスと手動で起動されるインスタンスを区別します。
- agent_hostname は，インスタンスが登録されるスマートエージェントの名前です。

ロケーションサービスは，負荷分散や監視などの目的に使用できます。たとえば，あるオブジェクトの複製（コピー）が複数のホストにあるとします。複製を提供するホスト名のキャッシュ，および各ホストの最新の負荷平均を維持するバインドインターセプタを配布します。このインターセプタは，現在オブジェクトのインスタンスを提供しているホストをロケーションサービスに問い合わせことにより，そのキャッシュを更新します。また，それらのホストにその負荷平均を問い合わせて取得します。そして，インターセプタは，最も負荷が小さいホストにある複製へのオブジェクトリファレンスを返します。インターセプタの記述の詳細については，第 24 章「ポータブルインターセプタの使い方」と第 25 章「VisiBroker インターセプタの使い方」を参照してください。

ロケーションサービスのコンポーネント

ロケーションサービスには，Agent インターフェースを介してアクセスできます。Agent インターフェースのメソッドは，2 つのグループに分けることができます。それらは，インスタンスを記述するデータをスマートエージェントに照会するメソッド，および**トリガー**を登録および登録解除するメソッドです。トリガーは，インスタンスの有効性に変更があったときに，ロケーションサービスのクライアントが通知を受けるためのメカニズムを提供します。

ロケーションサービスエージェントの概要

ロケーションサービスエージェントは，スマートエージェントのネットワーク上にあるオブジェクトを検索するためのメソッドの集合です。インターフェースのリポジトリ ID，またはインターフェースのリポジトリ ID とインスタンス名の組み合わせに基づいて照会を行うことができます。照会の結果は，**オブジェクトリファレンス**またはさらに完全な**インスタンス記述**として戻すことができます。オブジェクトリファレンスは，スマートエージェントによって検索されたオブジェクトの特定のインスタンスのハンドルにすぎません。インスタンス記述は，オブジェクトリファレンスのほかに，インスタンスのインターフェース名，インスタンス名，ホスト名，ポート番号，および状態に関する情報（実行中か，アクティブ化可能かなど）を保持します。

メモ　ロケーションサービスはコア VisiBroker ORB の一部になったので，locserv 実行可能ファイルはなくなっています。

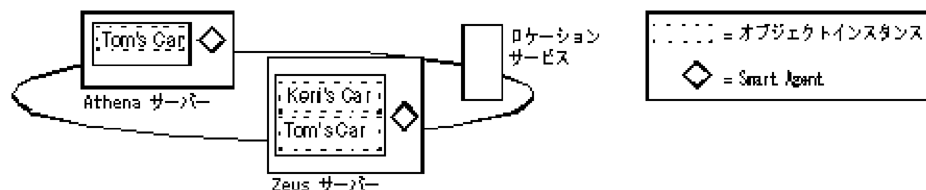
下の図は，次のサンプル IDL で指定されたインターフェースリポジトリ ID とインスタンス名の使用例を示します。

```
module Automobile {
  interface Car{...};
  interface Sedan:Car {...};
}
```

図 15.2　インターフェースリポジトリ ID とインスタンス名の使用

上のサンプルに基づいて，Car の複数のインスタンスへのリファレンスを持つネットワーク上のスマートエージェントを次の図に示します。この例には，Keri's Car のインスタンスが 1 つと Tom's Car の複製が 2 つ，合わせて 3 つのインスタンスがあります。

図 15.3　1 つのインターフェースの複数のインスタンスを持つネットワーク上のスマートエージェント



以下の節では，Agent クラスによって提供されるメソッドを使用して，VisiBroker スマートエージェントに情報を照会する方法について説明します。各照会メソッドは，Fail 例外を生成することがあります。この例外は，エラーの原因を示します。

スマートエージェントが動作するすべてのホストのアドレスを取得する

HostnameSeq メソッドを使用して，VisiBroker スマートエージェントをホストするサーバーを検索できます。次の図に示すサンプルで，このメソッドは，2 つのサーバー「Athena」と「Zeus」のアドレス（IP アドレス文字列など）を返します。

アクセス可能なすべてのインターフェースを検索する

ネットワーク上の VisiBroker スマートエージェントに照会して，アクセス可能なすべてのインターフェースを検索できます。検索するためには，RepositoryIDSeq メソッドを使用します。下の図に示すサンプルで，このメソッドは，2 つのインターフェース Car と Sedan のリポジトリ ID を返します。

メモ 以前のバージョンの VisiBroker ORB では，IDL インターフェース名を使ってインターフェースを識別しましたが，ロケーションサービスではかわりにリポジトリ ID を使用します。たとえば，次の名前のインターフェースがあるとします。

```
::module1::module2::interface
```

これに相当するリポジトリ ID は，次のようになります。

```
IDL:module1/module2/interface:1.0
```

上の図に示すサンプルで，Car のリポジトリ ID は，次のようになります。

```
IDL:Automobile/Car:1.0
```

Sedan のリポジトリ ID は，次のようになります。

```
IDL:Automobile/Sedan:1.0
```

1 つのインターフェースの複数のインスタンスへのリファレンスを取得する

ネットワーク上の VisiBroker スマートエージェントに照会して，特定のインスタンスで利用可能なすべてのインスタンスを検索できます。照会を実行する場合は，次のメソッドのどちらかを使用できます。

表 15.1　特定のインターフェースを実装するオブジェクトへのリファレンスの取得

メソッド	説明
CORBA::ObjectSeq* all_instances(const char* _repository_id)	このインターフェースのインスタンスへのオブジェクトリファレンスを返します。
DescSeq* all_instances_descs(const char* _repository_id)	このインターフェースのインスタンスのインスタンス記述を返します。

上の図に示されている例の場合は，どちらのメソッドを呼び出して IDL:Automobile/Car:1.0 を要求しても，Car インターフェースの 3 つのインスタンス，つまり Athena の Tom's Car，および Zeus の Tom's Car と Keri's Car が返されます。Tom's Car のインスタンスは，2 つの異なるスマートエージェントに見つかるので，2 度返されます。

1 つのインターフェースの名前が同じ複数のインスタンスへのリファレンスを取得する

次のメソッドのどちらかを使用すると，ネットワーク上の VisiBroker スマートエージェントに照会し，特定のインスタンス名を見つけてそれらをすべて取得することができます。

表 15.2　1 つのインターフェースの名前が同じインスタンスへのリファレンスの取得

メソッド	説明
CORBA::ObjectSeq* all_replica(const char* _repository_id, const char*_instance_name)	このインターフェースの名前が同じインスタンスへのオブジェクトリファレンスを返します。
DescSeq all_replica_descs(const char*_repository_id, const char* _instance_name)	このインターフェースの名前が同じインスタンスのインスタンス記述を返します。

上の図に示されている例の場合は，リポジトリ ID `IDL:Automobile/Sedan:1.0` とインスタンス名 Tom's Car を指定してどちらかのメソッドを呼び出すと，2 つの異なるスマートエージェントで見つかった 2 つのインスタンスが返されます。

トリガーの概要

トリガーは，本質的に，指定されたインスタンスの有効性に変更があったかどうかを判定するためのコールバックメカニズムです。これは，`Agent` をポーリングするための非同期の代替手段であり，通常，オブジェクトへの接続が失われた後の回復のために使用されます。照会は多くの目的で使用できますが，トリガーは特別な目的に使用します。

トリガーメソッド

次の表に，`Agent` クラスのトリガーメソッドを示します。

表 15.3　トリガーメソッド

メソッド	説明
`void reg_trigger(const TriggerDesc& _desc,TriggerHandler_ptr _handler)`	トリガーハンドラを登録します。
`void unreg_trigger(const TriggerDesc& _desc,TriggerHandler_ptr _handler)`	トリガーハンドラの登録を解除します。

`Agent` トリガーメソッドは，どちらも `Fail` 例外を生成することがあります。この例外は，エラーの原因を示します。

`TriggerHandler` インターフェースは次の表に説明するメソッドで構成されます。

表 15.4　TriggerHandler インターフェースのメソッド

メソッド	説明
`void impl_is_ready(const Desc& _desc)`	`desc` に一致するインスタンスがアクセス可能になると，ロケーションサービスによって呼び出されます。
`void impl_is_down(const Desc& _desc)`	インスタンスが利用できなくなると，ロケーションサービスによって呼び出されます。

トリガーの作成

`TriggerHandler` は，コールバックオブジェクトです。`TriggerHandlerPOA` クラス（または BOA では `TriggerHandlerImpl` クラス）から派生させて `TriggerHandler` を実装し，その `impl_is_ready()` と `impl_is_down()` メソッドを実装します。ロケーションサービスにトリガーを登録するには，`Agent` インターフェースの `reg_trigger()` メソッドを使用します。このメソッドには，監視するインスタンスの記述，およびインスタンスの有効性が変化したときに呼び出される `TriggerHandler` オブジェクトを提供する必要があります。インスタンスの記述（`TriggerDesc`）には，リポジトリ ID，インスタンス名，およびホスト名のインスタンス情報を組み合わせて挿入できます。インスタンス情報を多く提供するほど，よりインスタンスを特定できます。

```
struct TriggerDesc {
  string repository_id;
  string instance_name;
  string host_name;
};
```

メモ　空文字列（""）に設定された `TriggerDesc` のフィールドは無視されます。各フィールド値のデフォルトは，空文字列です。

たとえば，リポジトリ ID だけを含む `TriggerDesc` は，そのインターフェースのすべてのインスタンスと一致します。上のサンプルに戻ると，`IDL:Automobile/Car:1.0` の任意のインスタンスに対するトリガーは，Athena の Tom's Car，および Zeus の Tom's Car と Keri's Car のいずれか 1 つのインスタンスが使用可能になるか，使用できなくなった場合に発生します。`TriggerDesc` にインスタンス名「Tom's Car」を追加すると，指定範囲が狭

まり，2 つの「Tom's Car」インスタンスのどちらかの有効性が変化した場合にだけトリガーが発生するようになります。最後に，ホスト名 Athena を追加してさらにトリガーを特定すると，Athena サーバーの Tom's Car インスタンスが使用可能になるか，使用できなくなった場合にだけトリガーが発生するようになります。

トリガーによって検出される最初のインスタンス

トリガーは多少冗長です。トリガー記述を満たすオブジェクトがアクセス可能になるたびに，TriggerHandler が呼び出されます。最初のインスタンスがアクセス可能になるときを調べるだけの場合は，最初のインスタンスが見つかった後で，Agent の unreg_trigger() メソッドを呼び出してトリガーの登録を解除します。

エージェントの照会

この節では，ロケーションサービスを使ってインターフェースのインスタンスを検索する 2 つの例を示します。最初の例では，次の IDL の抜粋に示された Account インターフェースを使用します。

```
// Bank.idl
module Bank {
  interface Account {
    float balance();
  };
  interface AccountManager {
    Account open (in string name);
  };
};
```

1 つのインターフェースのすべてのインスタンスを検索する

次のサンプルコードは，all_instances() メソッドを使用して，Account インターフェースのすべてのインスタンスを検索します。スマートエージェントに照会するために，ORB::resolve_initial_references() メソッドに「LocationService」を渡し，そのメソッドから返されたオブジェクトを ObjLocation::Agent にナローイングしています。また，Account のリポジトリ ID の IDL:Bank/Account:1.0 の形式にも注目してください。

AccountManager インターフェースのすべてのインスタンスの検索

```
#include "corba.h"
#include "locate_c.hh"

// USE_STD_NS は，std 名前空間 USE_STD_NS を使用するために VisiBroker によって設定され
る定義です
int main(int argc, char** argv) {
  try {
    // ORB を初期化します。
    CORBA::ORB_var the_orb = CORBA::ORB_init(argc, argv);

    // ロケーションサービスへのリファレンスを取得します。
    CORBA::Object_var obj = the_orb->
      resolve_initial_references("LocationService");
    if ( CORBA::is_nil(obj) ) {
      cout << "Unable to locate initial LocationService" << endl;
      return 0;
    }
    ObjLocation::Agent_var the_agent = ObjLocation::Agent::_narrow(obj);

    // Account インターフェースのすべてのインプリメンテーションに対して，ロケーション
    // サービスを照会します。
```

```
    ObjLocation::ObjSeq_var accountRefs =
      the_agent->all_instances("IDL:Bank/AccountManager:1.0");
    cout << "Obtained " << accountRefs->length()
      << " Account objects" << endl;
    for (CORBA::ULong i=0; i < accountRefs->length(); i++) {
      cout << "Stringified IOR for account #" << i <<
        ":" << endl;
      CORBA::String_var stringified_ior(the_orb
        ->object_to_string(accountRefs[i]));
      cout << stringified_ior << endl;
      cout << endl;
    }
  }
  catch (const CORBA::Exception& e) {
    cout << "Caught exception: " << e << endl;
    return 0;
  }
  return 1;
}
```

スマートエージェントに既知のインターフェースとインスタンスを検索する

下のサンプルコードは，スマートエージェントが認識しているすべてのインターフェースを検索する方法です。まず，all_repository_ids() メソッドを呼び出して，すべての既知のインターフェースを取得します。次に，各インターフェースに対して all_instances_descs() メソッドを呼び出し，そのインスタンス記述を取得します。

スマートエージェントに既知のすべてのインターフェースの検索

```
#include "corba.h"
#include "locate_c.hh"

// USE_STD_NS は，std 名前空間が存在する場合に，それを使用するために
// VisiBroker によって設定される定義です

int DisplaybyRepID(CORBA::ORB_ptr the_orb,
  ObjLocation::Agent_var the_agent,
  char * myRepId) {

  ObjLocation::ObjSeq_var accountRefs;
  accountRefs = the_agent->all_instances(myRepId);
  cout << "Obtained " << accountRefs->length()
    << " Account objects" << endl;
  for (CORBA::ULong i=0; i < accountRefs->length(); i++) {
    cout << "Stringified IOR for account #" << i << ":"
      << endl;
    CORBA::String_var stringified_ior(
      the_orb->object_to_string(accountRefs[i]));
    cout << stringified_ior << endl;
    cout << endl;
  }
  return(1);
}
void PrintUsage(char * name) {
  cout << "¥nUsage: ¥n" << endl;
  cout << "¥t" << name << " [Rep ID]" << endl;
  cout << "¥n¥tWith no argument, finds and prints all objects" << endl;
  cout << "¥tOptional rep ID searches for specific rep ID¥n" << endl;
```

```
}
int main(int argc, char** argv) {
  char myRepId[255] = "";
  if (argc == 2) {
    if (!strcmp(argv[1], "-h") || !strcmp(argv[1], "/?") ||
      !strcmp(argv[1], "-?") ) {
      PrintUsage(argv[0]);
      exit(0);
    } else {
      strcpy(myRepId, argv[1]);
    }
  }
  else if (argc > 2) {
    PrintUsage(argv[0]);
    exit(0);
  }
  try {
    CORBA::ORB_ptr the_orb = CORBA::ORB_init(argc, argv);
    CORBA::Object_ptr obj = the_orb->
    resolve_initial_references("LocationService");
    if ( CORBA::is_nil(obj) ) {
      cout << "Unable to locate initial LocationService" << endl;
      return 0;
  }
  ObjLocation::Agent_var the_agent = ObjLocation::Agent::_narrow(obj);
  ObjLocation::DescSeq_var descriptors;
  // 要求された RepID の文字列化した IOR を表示して終了します。
  if (argc == 2) {
    DisplaybyRepID(the_orb, the_agent, myRepId);
    exit(0);
  }
  //osagent を実行しているすべてのホストをレポートします。
  ObjLocation::HostnameSeq_var HostsRunningAgents =
    the_agent->all_agent_locations();
  cout << "Located " << HostsRunningAgents->length()
    << " Hosts running Agents" << endl;
  for (CORBA::ULong k=0; k<HostsRunningAgents->length(); k++) {
    cout << "¥tHost #" << (k+1) << ": "
      << (const char*) HostsRunningAgents[k] << endl;
  }
  cout << endl;
  // すべてのリポジトリ ID を見つけて表示します。
  ObjLocation::RepositoryIdSeq_var repIds = the_agent->all_repository_ids();
  cout << "Located  " << repIds->length() <<
    " Repository Ids" << endl;
  for (CORBA::ULong j=0; j<repIds->length(); j++) {
    cout << "¥tRepository ID #" << (j+1) << ": "
      << repIds[j] << endl;
  }
  // 各リポジトリ ID について，すべてのオブジェクトデスクリプタを検索します。
  for (CORBA::ULong i=0; i < repIds->length(); i++) {
    descriptors = the_agent->all_instances_descs(repIds[i]);
    cout << endl;
    cout << "Located " << descriptors->length()
      << " objects for " << (const char*) (repIds[i])
      << " (Repository Id #" << (i+1) << "):"
        << endl;
    for (CORBA::ULong j=0; j < descriptors->length(); j++) {
      cout << endl;
      cout << (const char*) repIds[i] << " #" << (j+1)
```

```
        << ":" << endl;
      cout << "¥tInstance Name ¥t= " << descriptors[j].instance_name <<endl;
      cout << "¥tHost       ¥t= " << descriptors[j].iiop_locator.host
   <<endl;
      cout << "¥tPort       ¥t= " << descriptors[j].iiop_locator.port
   <<endl;
      cout << "¥tAgent Host ¥t= " << descriptors[j].agent_hostname <<endl;
      cout << "¥tActivable ¥t= " << (descriptors[j].activable?"YES":"NO") << endl;
    }
  }
  } catch (const CORBA::Exception& e) {
      cout << "CORBA Exception during execution of find_all: " << e << endl;
      return 0;
  }
  return 1;
}
```

トリガーハンドラの書き込みと登録

次のサンプルコードは，TriggerHandler を実装して登録します。TriggerHandlerImpl の impl_is_ready() メソッドと impl_is_down() メソッドは，トリガーを呼び出す原因となったインスタンスの詳細を表示します。また，選択にしたがってトリガーハンドラ自身の登録を解除します。

登録を解除した場合，それらのメソッドは CORBA::ORB::shutdown() メソッドを呼び出します。このメソッドは，メインプログラムの impl_is_ready() メソッドを終了するように BOA に指示し，プログラムは終了します。

TriggerHandlerImpl クラスは，作成時に使用された desc パラメータと Agent パラメータのコピーを保存していることに注目してください。unreg_trigger() メソッドには，desc パラメータが必要です。Agent パラメータは，メインプログラムからリファレンスが解放された場合にコピーされます。

トリガーハンドラの実装：

```
// AccountTrigger.c
#include "locate_s.hh"

// USE_STD_NS は，std 名前空間を使用するために VisiBroker によって設定される定義です
   USE_STD_NS
// このクラスのインスタンスは，登録されているイベントが発生したときに
// エージェントからコールバックされます。

class TriggerHandlerImpl : public _sk_ObjLocation::_sk_TriggerHandler
{
   public:
      TriggerHandlerImpl(
                  ObjLocation::Agent_var agent,
                  const ObjLocation::TriggerDesc& initial_desc)
                  : _agent(ObjLocation::Agent::_duplicate(agent)),
                  _initial_desc(initial_desc) {}

      void impl_is_ready(const ObjLocation::Desc& desc) {
         notification(desc, 1);
      }
      void impl_is_down(const ObjLocation::Desc& desc) {
         notification(desc, 0);
      }

   private:
```

```
      void notification(const ObjLocation::Desc& desc, CORBA::Boolean isReady) {
         if (isReady) {
            cout << "Implementation is ready:" << endl;
         } else {
            cout << "Implementation is down:" << endl;
         }
         cout << "¥tRepository Id = " << desc.repository_id << endl;
         cout << "¥tInstance Name = " << desc.instance_name << endl;
         cout << "¥tHost Name     = " << desc.iiop_locator.host << endl;
         cout << "¥tPort      = " << desc.iiop_locator.port << endl;
         cout << "¥tAgent Host = " << desc.agent_hostname << endl;
         cout << "¥tActivable     = " << (desc.activable?"YES" : "NO") << endl;
         cout << endl;
         cout << "Unregister this handler and exit (yes/no)? " << endl;
         char prompt[256];
         cin >> prompt;
         if ((prompt[0] == 'y') || (prompt[0] == 'Y')) {
            try {
               _agent->unreg_trigger(_initial_desc, this);
            }
            catch (const ObjLocation::Fail& e) {
               cout << "Failed to unregister trigger with reason=["
                              << (int) e.reason << "]" << endl;
            }
            cout << "exiting..." << endl;
            CORBA::ORB::shutdown();
         }
      }

   private:
      ObjLocation::Agent_var _agent;
      ObjLocation::TriggerDesc _initial_desc;
};

int main(int argc, char* const * argv)
{
   try {
      CORBA::ORB_var the_orb = CORBA::ORB_init(argc, argv);
      CORBA::BOA_var boa = the_orb->BOA_init(argc, argv);
      CORBA::Object_var obj = the_orb->
      resolve_initial_references("LocationService");
      if ( CORBA::is_nil(obj) ) {
         cout << "Unable to locate initial LocationService" << endl;
         return 0;
      }
      ObjLocation::Agent_var the_agent = ObjLocation::Agent::_narrow(obj);

      // 該当する Account オブジェクトの OSAgent の変化を通知します。
      // トリガーデスクリプタを作成します。
      ObjLocation::TriggerDesc desc;
      desc.repository_id = (const char*) "IDL:Bank/AccountManager:1.0";
      desc.instance_name = (const char*) "";
      desc.host_name = (const char*) "";

      ObjLocation::TriggerHandler_var trig = new TriggerHandlerImpl(the_agent,
         desc);
      boa->obj_is_ready(trig);
      the_agent->reg_trigger(desc,trig);
      boa->impl_is_ready();
   }
```

```
   catch (const CORBA::Exception& e) {
      cout << "account_trigger caught Exception: " << e << endl;
      return 0;
   }
   return 1;
}
```

第 16 章

VisiNaming サービスの使い方

ここでは，CORBA ネーミングサービス仕様バージョン 1.2（正式版 02-09-02）の完全なインプリメンテーションである VisiBroker VisiNaming サービスの使用方法について説明します。

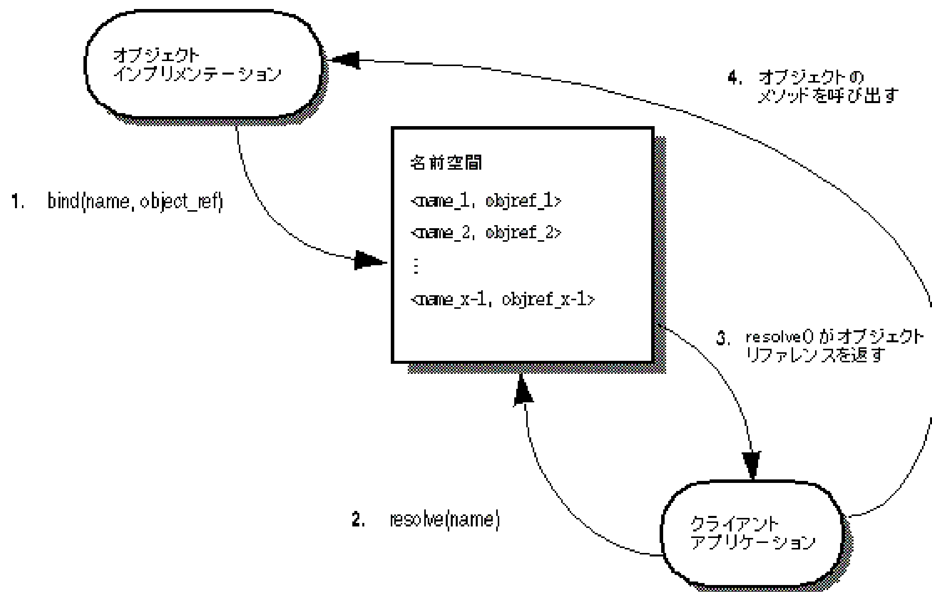
概要

VisiNaming サービスでは，1 つのオブジェクトリファレンスに 1 つ以上の*論理名*を関連付けることができます。また，これらの名前は*名前空間*に保管できます。VisiNaming サービスにより，クライアントアプリケーションは，オブジェクトに割り当てられた論理名を使用してオブジェクトリファレンスを取得できます。

この後の図では，VisiNaming サービスの次の機能について概要を示します。

1. オブジェクトインプリメンテーションは，名前を名前空間内のオブジェクトの 1 つに**バインド**できます。
2. クライアントアプリケーションは，同じ名前空間を使用して名前を*解決*し，ネーミングコンテキストやオブジェクトへのオブジェクトリファレンスを取得します。

図 16.1　名前空間内のネーミングコンテキストにあるオブジェクト名のバインド，解決，および使用



VisiNaming サービスとスマートエージェントとでは，オブジェクトインプリメンテーションの検索に重要な違いがいくつかあります。

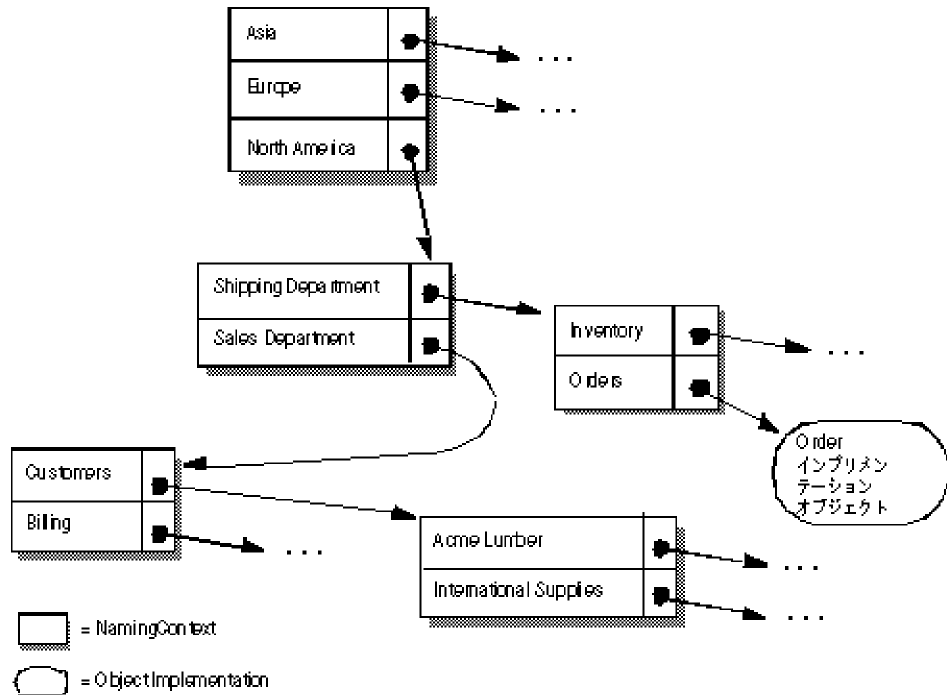
- スマートエージェントはフラットな名前空間を使用しますが，VisiNaming サービスでは階層構造を持つ名前空間を使用します。
- スマートエージェントを使用する場合，オブジェクトのインターフェース名は，クライアントとサーバーアプリケーションをコンパイルするときに定義します。したがって，インターフェース名を変更すると，アプリケーションも再コンパイルする必要があります。これに対して，VisiNaming サービスでは，実行時にオブジェクトインプリメンテーションでそのオブジェクトに論理名をバインドできます。
- スマートエージェントを使用する場合，オブジェクトは 1 つのインターフェース名しか実装できません。VisiNaming サービスでは，1 つのオブジェクトに複数の論理名をバインドできます。

スマートエージェント（osagent）の詳細は，第 14 章「スマートエージェントの使い方」を参照してください。

名前空間の理解

次の図は，受注管理システムを構成しているオブジェクトに，VisiNaming サービスで名前を付ける方法を示します。この架空の受注管理システムでは，地域，部署などによってその名前空間を整理します。VisiNaming サービスでは，NamingContext オブジェクトの階層構造で名前空間を整理し，そのオブジェクトをたどって特定の名前を探すことができます。たとえば，論理名 NorthAmerica/ShippingDepartment/Orders で Order オブジェクトを検索できます。

図 16.2　受注管理システムの名前の付け方



ネーミングコンテキスト

上に示す名前空間を VisiNaming サービスで実装するには，影付きのボックスをそれぞれ NamingContext オブジェクトで実装します。NamingContext オブジェクトには，オブジェクトインプリメンテーションや，他の NamingContext オブジェクトにバインドされた Name 構造体のリストが収められています。NamingContext に論理名をバインドできますが，NamingContext にはデフォルトで論理名が関連付けられていないか，そのような名前が不要なので注意してください。

オブジェクトインプリメンテーションは，NamingContext オブジェクトを使用して，提供するオブジェクトに名前をバインドします。クライアントアプリケーションは，NamingContext を使用して，バインドされた名前をオブジェクトリファレンスに解決します。

NamingContextExt インターフェースも利用でき，このインターフェースでは，文字列化したオブジェクトの使用時に必要なメソッドを提供します。

ネーミングコンテキストファクトリ

ネーミングコンテキストファクトリでは，VisiNaming サービスをブートストラップするインターフェースを用意しています。このインターフェースには，VisiNaming サービスをシャットダウンしたり，コンテキストがない場合に新しいコンテキストを作成するためのオペレーションがあります。また，ファクトリにはルートコンテキストを返す追加の API

もあります。ルートコンテキストは，リファレンスポイントとして重要な役割を果たします。これは，パブリックで使用される予定のデータを保存する共通の開始位置になります。

名前空間を作成するために VisiNaming サービスによって提供されるクラスには，デフォルトネーミングコンテキストファクトリと拡張ネーミングコンテキストファクトリの 2 つがあります。デフォルトのネーミングコンテキストファクトリでは，ルート NamingContext なしの空の名前空間を作成します。拡張ネーミングコンテキストファクトリでは，ルート NamingContext 付きの名前空間が作成されるので，この方が便利です。

オブジェクトインプリメンテーションによるオブジェクトと名前のバインドや，クライアントアプリケーションによる名前からオブジェクトリファレンスへの解決では，これらの NamingContext オブジェクトのうち最低 1 つを，事前に取得してください。

上の図に示す NamingContext オブジェクトは，1 つのネーミングサービスプロセス内に実装できます。または，最大 5 つの独立した名前サーバープロセス内に実装できます。

Name と NameComponent

CosNaming::Name は，オブジェクトインプリメンテーションまたは CosNaming::NamingContext にバインドできる識別子を表します。Name は，単純な英字の文字列ではなく，1 つ以上の NameComponent 構造体のシーケンスを表します。

各 NameComponent には，id と kind という 2 つの属性文字列が含まれています。ネーミングサービスは，各 id と kind が特定の NamingContext 内で一意であることを確認する以外に，これらの文字列の解釈や管理は行いません。

id 属性と kind 属性は，名前をバインドしたオブジェクトを一意で識別する文字列です。kind メンバーは，名前に追加する特性を記述します。たとえば，「Inventory.RDBMS」という名前は，「Inventory」の id メンバーと「RDBMS」の kind メンバーで構成されています。

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

NameComponent の id 属性と kind 属性に使用できるのは，ISO 8859-1（Latin-1）文字セットだけです。null 文字（0x00）と，その他の印刷不可文字は使用できません。NameComponent の文字列は，どちらも 255 文字までです。また，VisiNaming サービスは，ワイド文字列を使用する NameComponent をサポートしていません。

メモ　名前の id 属性を空の文字列にすることはできませんが，kind 属性では可能です。

名前の解決

クライアントアプリケーションは，NamingContext の resolve メソッドを使用して，特定の論理名（Name）のオブジェクトリファレンスを取得します。1 つの Name は，1 つまたは複数の NameComponent オブジェクトから構成されるため，解決のプロセスでは Name を構成しているすべての NameComponent 構造体を調べる必要があります。

文字列化された名前

CosNaming::Name 表現は読み取りや変換に適した形式ではないため，この問題を解決するために文字列化した名前が定義されています。文字列化された名前は，文字列と CosNaming::Name とを 1 対 1 でマップします。2 つの CosNaming::Name オブジェクトが等しい場合は，それらの文字列化表現も等しくなり，その逆も同様です。文字列化された名前では，名前の区切りにはスラッシュ（/），id と kind 属性の区切りにはピリオド（.），エス

ケープ文字にはバックスラッシュ（\）を使用します。規則により，Order など，空の kind 属性を持つ NameComponent ではピリオドを使用しません。

```
"Borland.Company/Engineering.Department/Printer.Resource"
```

メモ　次のサンプルでは，NameComponent 構造体が文字列化表現で示されています。

単純な名前と複雑な名前

Billing などの**単純な名前**は，常に 1 つの NameComponent で構成され，その解決は目的のネーミングコンテキストを基準に行われます。単純な名前は，1 つのオブジェクトインプリメンテーションや 1 つの NamingContext にしかバインドできません。

NorthAmerica/ShippingDepartment/Inventory などの**複雑な名前**は，複数の NameComponent 構造体のシーケンスで構成します。**n** 個の NameComponent オブジェクトからなる複雑な名前が 1 つのオブジェクトインプリメンテーションにバインドされている場合，シーケンス内の最初の（n-1）の NameComponent オブジェクトは，それぞれ必ず NamingContext に解決され，最後の NameComponent オブジェクトはオブジェクトインプリメンテーションに解決されます。

Name が 1 つの NamingContext にバインドされる場合，シーケンス内の各 NameComponent 構造は，NamingContext を参照する必要があります。

次のサンプルコードは，3 つのコンポーネントからなる複雑な名前であり，CORBA オブジェクトにバインドされています。この名前は文字列化された名前 NorthAmerica/SalesDepartment/Order に対応しています。最上位のネーミングコンテキストの段階で解決すると，この複雑な名前の最初の 2 つの要素は NamingContext オブジェクトに解決し，最後の要素は論理名 Order を持つオブジェクトインプリメンテーションに解決します。

```
. . .
// Name は文字列 "NorthAmerica/SalesDepartment/Order" を示します。
CosNaming::Name_var continentName =
 rootNamingContext->to_name("NorthAmerica");
CosNaming::NamingContext_var continentContext =
      rootNamingContext->bind_new_context(continentName);
CosNaming::Name_var departmentName = continentContext->to_name("SalesDepartment");
CosNaming::NamingContext_var departmentContext =
      rootNamingContext->bind_new_context(departmentName);
CosNaming::Name_var objectName =
 departmentContext->to_name("Order");
      departmentContext->rebind(objectName, myPOA-
>servant_to_reference(managerServant));
. . .
```

VisiNaming サービスの実行

VisiNaming サービスは，次のコマンドで起動できます。起動したネーミングサービスのコンテンツは，VisiBroker コンソールで閲覧できます。

VisiNaming サービスのインストール

VisiNaming サービスは，VisiBroker 5.0 をインストールすると自動的にインストールされます。ネーミングサービスは，ファイル nameserv と Java クラスファイルで構成されています。nameserv は Windows NT ではバイナリ実行可能ファイル，UNIX ではスクリプトです。また，Java クラスファイルは vbjorb.jar ファイルに格納されています。

VisiNaming サービスの設定

以前のバージョンの VisiBroker では，フラットファイルに対するすべての変更操作を VisiNaming サービスで記録して，永続性を維持していました。バージョン 4.0 以降の

VisiNaming サービスは，バックストアアダプタとともに機能します。ただし，すべてのバックストアアダプタが永続性をサポートしているわけではないので注意してください。デフォルトの InMemory アダプタは永続的ではありませんが，その他のアダプタは永続的です。アダプタの詳細は，193 ページの「取り替え可能なバックストア」を参照してください。

メモ　ネーミングサービスは，起動時に自分自身をスマートエージェントに登録するように設計されています。ほとんどの場合，VisiNaming サービスをブートストラップするには，スマートエージェントを実行する必要があります。これにより，クライアントは，resolve_initial_references メソッドを呼び出して初期のルートコンテキストを取得できます。必要なリファレンスを取得するために，スマートエージェントで解決機能が動作します。同様に，このしくみに加わるネーミングサーバーも，同じメカニズムを使用して必要なリファレンスを取得します。

スマートエージェントの詳細は，第 14 章「スマートエージェントの使い方」を参照してください。

VisiNaming サービスの起動

VisiNaming サービスを起動するには，/bin ディレクトリの nameserv 起動プログラムを使用します。nameserv 起動プログラムは，デフォルトで com.inprise.vbroker.naming.ExtFactory ファクトリクラスを使用します。

UNIX
```
nameserv [driver_options] [nameserv_options] <ns_name> &
```

Windows
```
start nameserv [driver_options] [nameserv_options] <ns_name>
```

すべての VisiBroker プログラマツールで使用できるドライバオプションについては，26 ページの「共通オプション」を参照してください。

表 16.1　nameserv_option のオプションと説明

nameserv_option	説明
-?, -h, -help, -usage	使い方を表示します。
-config <properties_file>	VisiNaming サービスの起動では，設定ファイルとして <properties_file> を使用します。
<ns_name>	この VisiNaming サービスの名前を指定します。これはオプションです。デフォルトの名前は NameService です。

VisiNaming サービスを強制的に特定のポートで起動するには，次のコマンドラインオプションを使用して VisiNaming サービスを起動する必要があります。

```
prompt> nameserv -J-Dvbroker.se.iiop_tp.scm.iiop_tp.listener.port=<port number>
```

VisiNaming のデフォルト名は「NameService」ですが，これ以外の名前を指定する場合は，次のようにして VisiNaming を起動します。

```
prompt> nameserv -J-Dvbroker.se.iiop_tp.scm.iiop_tp.listener.port=<port number>
<ns_name>
```

コマンドラインからの VisiNaming サービスの起動

VisiNaming サービスユーティリティ（nsutil）を使用して，コマンドラインからバインディングを格納および取得できます。

nsutil の設定

nsutil を使用するには，最初に次のコマンドを使用してネーミングサービスのインスタンスを設定します。

```
prompt>nameserv <ns_name>
```

```
prompt>nsutil -VBJprop <option> <cmd> [args]
```

オプション	説明
ns_name	接続するネーミングサービスを設定します。
SVCnameroot=<ns_name>	メモ：SVCnameroot を使用する前に，OSAgent を実行しておいてください。
ORBInitRef=NameService=<url>	ファイル名または URL を指定します。その種類（corbaloc:，corbaname:，file:，ftp:，http:，ior: など）を前に付けます。たとえば，ローカルディレクトリ内のファイルを割り当てる場合，ns_config 文字列は -VBJprop ORBInitRef=NameService=<file:ns.ior> のようになります。
cmd	任意の CosNaming 操作。ほかに，ping または shutdown もあります。

nsutil の実行

VisiNaming サービスユーティリティでは, すべての CosNaming 操作と 3 つの追加コマンドを使用できます。サポートされている CosNaming 操作は次のとおりです。

cmd	パラメータ
bind	name objRef
bind_context	name ctxRef
bind_new_context	name
destroy	name
list	[name1 name2 name3...]
new_context	パラメータなし
rebind	name objRef
rebind_context	name ctxRef
resolve	name
unbind	name

メモ　destroy および list オペレーションでは，name パラメータが既存のネーミングコンテキストを参照する必要があります。list オペレーションの場合にのみ，0 個以上のネーミングコンテキストがあればよく，その内容が表示されます。ネーミングコンテキストが指定されていない場合は，ルートネーミングコンテキストの内容が表示されます。

ほかにも，次の nsutil コマンドを使用できます。

cmd	パラメータ	説明
ping	name	文字列化された name を解決し，そのオブジェクトにコンタクトしてまだ存続しているかどうかを確認します。
shutdown	< ネーミングコンテキストファクトリ名または文字列化された IOR>	VisiNaming サービスをコマンドラインから正常にシャットダウンします。この操作の必須パラメータには，osagent に登録されたネーミングファクトリの名前またはファクトリの文字列化された IOR を指定します。
unbind_from_cluster	name objRef	暗黙的なクラスタ内の特定のオブジェクトをバインド解除します。name はオブジェクトの論理名，objRef はバインド解除される文字列化されたオブジェクトリファレンスです。

nsutil コマンドからオペレーションを実行するには，オペレーション名とパラメータを <cmd> パラメータの位置に指定します。たとえば，次のようになります。

```
prompt>nsutil -VBJprop ORBInitRef=NameService=file://ns.ior resolve myName
```

nsutil を使用して VisiNaming サービスを閉じる

nsutil を使用して VisiNaming サービスを閉じるには，shutdown コマンドを実行します。

```
prompt>nsutil -VBJprop ORBInitRef=NameService=file://ns.ior shutdown <ns_name>
```

VisiNaming サービスのブートストラップ

クライアントアプリケーションを起動して，指定した VisiNaming サービスへの初期のオブジェクトリファレンスを取得する方法は 3 つあります。VisiNaming サービスの起動時には，次のコマンドラインオプションを使用できます。

- ORBInitRef
- ORBDefaultInitRef
- SVCnameroot

次の例に，オプションの使い方を示します。

ここでは，ホスト TestHost で次の 3 つの VisiNaming サービスが動作しているとします。

```
ns1, ns2, ns3
```

それぞれポート 20001，20002，および 20003 で実行されます。

さらに，次の 3 つのサーバーアプリケーションがあります。

```
sr1, sr2, sr3.
```

サーバー sr1 は自分自身を ns1 でバインドします。

サーバー sr2 は自分自身を ns2 でバインドします。

サーバー sr3 は自分自身を ns3 でバインドします。

resolve_initial_references の呼び出し

VisiNaming サービスメカニズムで resolve_initial_references メソッドを設定し，簡単に共通ネーミングコンテキストを取得できます。クライアントプログラムが接続するネー

ミングサーバーのルートコンテキストを取得するには，resolve_initial_references メソッドを使用します。

```
. . .
   CORBA::ORB_ptr orb = CORBA::ORB_init(argv, argc, NULL);
   CORBA::Object_var rootObj = orb->resolve_initial_references("NameService");
. . .
```

-DSVCnameroot の使用

-DSVCnameroot オプションでは，ブートストラップする VisiNaming サービスのインスタンスを指定します。これは，互いに無関係な複数のネーミングサービスのインスタンスが実行されている場合，特に重要です。

たとえば ns1 にブートストラップする場合は，次のようにクライアントプログラムを起動します。

```
<client_application> -DSVCnameroot=ns1
```

これで次の図にも示したように，クライアントアプリケーションから ORB リファレンスの resolve_initial_references メソッドを呼び出して ns1 のルートコンテキストを取得できます。このオプションを使用する前に，OSAgent を起動しておいてください。

-DSVCnameroot ブートストラップメカニズムは，VisiBroker OSAgent 固有の機能に基づいており，他の CORBA インプリメンテーションとは相互運用できないことに注意してください。

-ORBInitRef の使用

ブートストラップする VisiNaming サービスは，corbaloc と corbaname のどちらかの URL 命名方式で指定できます。このメソッドはスマートエージェントに依存しません。

corbaloc URL の使用

VisiNaming サービス ns2 でブートストラップする場合は，次のようにクライアントアプリケーションを起動します。

```
<client_application> -ORBInitRef=NameService=corbaloc://TestHost:20002/NameService
```

これで，上の例で示したように，クライアントアプリケーションから VisiBroker ORB リファレンスの resolve_initial_references メソッドを呼び出して，ns2 のルートコンテキストを取得できます。

メモ　非推奨の iioploc と iiopname の URL 方式はそれぞれ，corbaloc と corbaname によって実装されています。下位互換性を保つために，古いネーミング方式もサポートされています。

corbaname URL の使用

corbaname を使用して ns3 にブートストラップする場合は, 次のようにクライアントプログラムを起動します。

```
<client_application> -ORBInitRef NameService=corbaname://TestHost:20003/
```

これで，上に示したように，クライアントアプリケーションから VisiBroker ORB リファレンスの resolve_initial_references メソッドを呼び出して，ns3 のルートコンテキストを取得できます。

-ORBDefaultInitRef

ブートストラップする VisiNaming サービスは，corbaloc と corbaname のどちらかの URL で指定できます。このメソッドはスマートエージェントに依存しません。

-ORBDefaultInitRef での corbaloc URL の使用

ns2 にブートストラップする場合は，次のようにクライアントプログラムを起動します。

```
<client_application> -ORBDefaultInitRef corbaloc://TestHost:20002
```

これで，上の例に示したように，クライアントアプリケーションから VisiBroker ORB リファレンスの resolve_initial_references メソッドを呼び出して，ns2 のルートコンテキストを取得できます。

-ORBDefaultInitRef での corbaname の使用

-ORBDefaultInitRef または -DORBDefaultInitRef と corbaname を組み合わせると, 予想とは異なる動作を行います。-ORBDefaultInitRef または -DORBDefaultInitRef を指定すると，スラッシュと文字列化オブジェクト key が常に corbaname に追加されます。

たとえば，URL が corbaname::TestHost:20002 の場合は，-ORBDefaultInitRef を指定することで, C++ の resolve_initial_references の結果は新しい URL corbaname::TestHost:20003/NameService になります。

NamingContext

このオブジェクトは，VisiBroker ORB オブジェクトまたは他の NamingContext オブジェクトにバインドされている名前のリストを保持および操作するために使用されます。クライアントアプリケーションはこのインターフェースを使用して, そのコンテキスト内のすべての名前を解決，または一覧表示します。オブジェクトインプリメンテーションはこのオブジェクトを使用して，オブジェクトインプリメンテーションや NamingContext オブジェクトに名前をバインドします。次のサンプルに，NamingContext の IDL 仕様を示します。

```
Module CosNaming {
   interface NamingContext {
      void bind(in Name n, in Object obj)
         raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
      void rebind(in Name n, in Object obj)
         raises(NotFound, CannotProceed, InvalidName);
      void bind_context(in Name n, in NamingContext nc)
         raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
      void rebind_context(in Name n, in NamingContext NC)
         raises(NotFound, CannotProceed, InvalidName);
      Object resolve(in Name n)
         raises(NotFound, CannotProceed, InvalidName);
      void unbind(in Name n)
         raises(NotFound, CannotProceed, InvalidName);
      NamingContext new_context();
      NamingContext bind_new_context(in Name n)
         raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
      void destroy()
         raises(NotEmpty);
      void list(in unsigned long how_many,
            out BindingList bl,
            out BindingIterator bi);
   };
};
```

NamingContextExt

NamingContext を拡張した NamingContextExt インターフェースは，文字列化された名前と URL の使用に必要な操作を提供します。

```
Module CosNaming {
   interface NamingContextExt : NamingContext {
      typedef string StringName;
      typedef string Address;
      typedef string URLString;
      StringName to_string(in Name n)
         raises(InvalidName);
      Name to_name(in StringName sn)
         raises(InvalidName);
      exception InvalidAddress {};
      URLString to_url(in Address addr, in StringName sn)
         raises(InvalidAddress, InvalidName);
      Object resolve_str(in StringName n)
         raises(NotFound, CannotProceed, InvalidName);
   };
};
```

デフォルトネーミングコンテキスト

クライアントアプリケーションでは，デフォルトのネーミングコンテキストを指定できます。アプリケーションでは，これをルートコンテキストとみなします。デフォルトのネーミングコンテキストはそのクライアントアプリケーションだけに対するルートで，実際は別のコンテキストに属する場合もあります。

デフォルトコンテキストの取得

クライアントアプリケーションでは，VisiBroker ORB メソッド resolve_initial_references でデフォルトのネーミングコンテキストを取得できます。デフォルトのネーミングコンテキストは，クライアントアプリケーションの起動時に SVCnameroot または ORBInitRef コマンドライン引数で指定しておきます。次のサンプルは，C++ クライアントアプリケーションでこのメソッドを呼び出す方法を示しています。

```
#include "CosNaming_c.hh"
. . .
int main(int argc, char* const* argv) {
   try {
      CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
      . . .
      CORBA::Object_var ref = orb->resolve_initial_references("NameService");
      CosNaming::NamingContext_var rootContext =
            CosNaming::NamingContext::_narrow(ref);
      . . .
   } catch(const CORBA::Exception& e) {
      cout << "Failure: " << e << endl;
      exit(1);
   }
   exit(0);
}
```

ネーミングコンテキストファクトリの取得

ネットワーク上で実行されている osagent がない場合，ネーミングサービスクライアントは，次のようにファクトリの初期リファレンスを解決することで，ネーミングコンテキストファクトリへのリファレンスを取得できます。

```
...
CORBA::Object_var factRef = orb->resolve_initial_references("VisiNamingContextFactory");
CosNamingExt::NamingContextFactory_var factory =
        CosNamingExt::NamingContextFactory::_narrow(factRef);
...
```

次の例に示すように，このクライアントを起動します。

```
Client -ORBInitRef = VisiNamingContextFactory =
corbaloc::<host>:<port>/VisiNamingContextFactory
```

VisiNaming サービスのプロパティ

次の表は，VisiNaming サービスのプロパティを一覧です。

表 16.2　VisiNaming サービスのコアプロパティ

プロパティ	デフォルト値	説明
vbroker.naming.adminPwd	inprise	Visibroker ネーミングサービスの管理操作に必要なパスワード。
vbroker.naming.enableSlave	0	1 に設定した場合は，マスター／スレーブのネーミングサービス設定が有効になります。マスター／スレーブのネーミングサービスの設定方法については，203 ページの「VisiNaming サービスクラスタによるフェイルオーバーと負荷分散」を参照してください。
vbroker.naming.iorFile	ns.ior	ネーミングサービス IOR を格納するためのフルパス名を指定します。このプロパティを設定しないと，ネーミングサービスは IOR を現在のディレクトリにある ns.ior という名前のファイルに出力します。ネーミングサービスは，IOR の出力時にファイルアクセス許可の例外を暗黙的に無視します。
vbroker.naming.logLevel	emerg	ネーミングサービスから出力されるログメッセージのレベルを指定します。次の値を指定できます。 • vbroker.log.enable=true • vbroker.log.filter.default.enable=false • vbroker.log.filter.default.register=naming • vbroker.log.filter.default.naming.enable=true • vbroker.log.filter.default.naming.logLevel=debug

表 16.2 VisiNaming サービスのコアプロパティ（続き）

プロパティ	デフォルト値	説明
vbroker.naming.logUpdate	false	このプロパティにより，CosNaming::NamingContext，CosNamingExt::Cluster，および CosNamingExt::ClusterManager インターフェースのすべての更新操作をログに記録できます。 このプロパティが有効な CosNaming::NamingContext インターフェースのオペレーションは，次のとおりです。 bind，bind_context，bind_new_context，destroy，rebind，rebind_context，unbind。 このプロパティが有効な CosNamingExt::Cluster インターフェースのオペレーションは，次のとおりです。 bind，rebind，unbind，destroy。 このプロパティが有効な CosNamingExt::ClusterManager インターフェースのオペレーションは，次のとおりです。 create_cluster このプロパティの値が true に設定されている場合に上のいずれかのメソッドが呼び出されると，次のようなログメッセージが出力されます（この出力は実行中のバインド操作を示します）。 00000007,5/26/04 10:11 AM,127.0.0.1,00000000, VBJ-Application,VBJ ThreadPool Worker,INFO, OPERATION NAME : bind CLIENT END POINT : Connection[socket=Socket [addr=/127.0.0.1, port=2026, localport=1993]] PARAMETER 0 : [(Tom.LoanAccount)] PARAMETER 1 : Stub[repository_id=IDL:Bank/LoanAccount:1.0, key=TransientId[poaName=/, id={4 bytes: (0)(0)(0)(0)},sec=505,usec=990917734, key_string=%00VB%01%00%00%00%02/%00%20%20%00%00%00%04%00%00%00%00%00%00%01%f9;%104f],codebase=null]

詳細は，199 ページの「クラスタ」を参照してください。

表 16.3 オブジェクトクラスタ関連のプロパティ

プロパティ	デフォルト値	説明
vbroker.naming.enableClusterFailover	true	true に設定した場合は，VisiNaming サービスから取得されたオブジェクトのフェイルオーバーを処理するインターセプタがインストールされます。オブジェクトに障害が発生した場合は，元のオブジェクトと同じクラスタにある別のオブジェクトに対して透過的な再接続を試みます。
vbroker.naming.propBindOn	0	1 の場合，暗黙的クラスタリング機能が有効になります。

表 16.3　オブジェクトクラスタ関連のプロパティ（続き）

プロパティ	デフォルト値	説明
vbroker.naming.smrr.pruneStaleRef	1	このプロパティが関係するのは，ネーミングサービスクラスタが SmartRoundRobin 基準を使用する場合です。このプロパティを 1 に設定すると，ネーミングサービスが SmartRoundRobin 基準を使用してクラスタへ以前にバインドした無効なオブジェクトリファレンスを発見すると，バインディングから除去します。このプロパティを 0 に設定すると，クラスタにおける無効なオブジェクトリファレンスバインディングは削除されません。ただし，SmartRoundRobin 基準を持つクラスタは，resolve() 呼び出しまたは select() 呼び出しでアクティブオブジェクトリファレンスを常に返します。これは，そのようなオブジェクトバインディングが存在するのであれば，vbroker.naming.smrr.pruneStaleRef プロパティの値に関係ありません。デフォルトでは，4.5 ネーミングサービスでの暗黙的なクラスタリングは，このプロパティ値が 1 に設定された SmartRoundRobin 基準を使用します。このプロパティを 2 に設定すると，無効なリファレンスが削除されなくなり，バインディングのクリーンアップは VisiNaming ではなくアプリケーションが行うことになります。

詳細は，203 ページの「VisiNaming サービスクラスタによるフェイルオーバーと負荷分散」を参照してください。

表 16.4　VisiNaming サービスクラスタ関連のプロパティ

プロパティ	デフォルト値	説明
vbroker.naming.enableSlave	0	「VisiNaming サービスのコアプロパティ」を参照してください。
vbroker.naming.slaveMode	デフォルトなし。cluster または slave に設定できます。	このプロパティを使用して，クラスタモードまたはマスター／スレーブモードの VisiNaming サービスインスタンスを設定します。このプロパティを有効にするには，vbroker.naming.enableSlave プロパティを 1 に設定する必要があります。 クラスタモードの VisiNaming サービスインスタンスを設定するには，このプロパティを cluster に設定します。これで，クラスタを構成する VisiNaming サービスインスタンス間で VisiNaming サービスクライアントが負荷分散されます。これらのインスタンス間でのクライアントのフェイルオーバーが有効になります。 マスター／スレーブモードの VisiNaming サービスインスタンスを設定するには，このプロパティを slave に設定します。VisiNaming サービスクライアントは，マスターの実行中は常にマスターサーバーにバインドされますが，マスターサーバーがダウンした場合はスレーブサーバーにフェイルオーバーします。
vbroker.naming.serverClusterName	null	このプロパティは，VisiNaming サービスクラスタの名前を指定します。このプロパティを使用して，複数の VisiNaming サービスインスタンスにクラスタ名（たとえば，clusterXYZ）を設定した場合，それらのインスタンスは特定のクラスタに属します。
vbroker.naming.serverNames	null	このプロパティは，クラスタに属している VisiNaming サービスインスタンスのファクトリ名を指定します。クラスタ内の各 VisiNaming サービスインスタンスは，このプロパティを使用して，クラスタを構成するすべてのインスタンスを認識するように設定されます。リスト内の名前は一意である必要があります。このプロパティは次の形式をサポートします。 `vbroker.naming.serverNames=Server1:Server2:Server3` 関連のプロパティ vbroker.naming.serverAddresses を参照してください。

表 16.4　VisiNaming サービスクラスタ関連のプロパティ（続き）

プロパティ	デフォルト値	説明
vbroker.naming.serverAddresses	null	このプロパティは，ホストおよび VisiNaming サービスクラスタを構成する VisiNaming サービスインスタンスの監視ポートを指定します。このリスト内の VisiNaming サービスインスタンスの順番は，関連プロパティ vbroker.naming.serverNames（VisiNaming サービスクラスタを構成する VisiNaming サービスインスタンスの名前を指定する）の順番と同じである必要があります。このプロパティは次の形式をサポートします。 `vbroker.naming.serverAddresses=host1:port1;host2:port2;host3:port3`
vbroker.naming.anyServiceOrder (To be set on VisiNaming Service clients)	false	VisiNaming サービスインスタンスが VisiNaming サービスクラスタモードで設定されている場合に負荷分散機能とフェイルオーバー機能を使用するには，VisiNaming サービスクライアントでこのプロパティを true に設定する必要があります。このプロパティの使い方の例を次に示します。 `client -DVbroker.naming.anyServiceOrder=true`

取り替え可能なバックストア

VisiNaming サービスでは，取り替え可能なバックストアを使用してその名前空間を維持します。名前空間が永続的であるかどうかは，バックストアの設定，つまり JDBC アダプタ，JNDI，またはインメモリアダプタ（デフォルト）のどれを使用するかよって異なります。JNDI は Java Naming and Directory Interface の略で，LDAP で動作保証されています。

バックストアの種類

サポートされているバックストアアダプタの種類は次のとおりです。

- インメモリアダプタ
- リレーショナルデータベース用の JDBC アダプタ
- DataExpress アダプタ
- JNDI（LDAP のみ）

メモ　取り替え可能なアダプタの使い方については，次のディレクトリ内のサンプルコードを参照してください。

```
<install dir>/vbe/examples/ins/pluggable_adaptors
```

インメモリアダプタ

インメモリアダプタは名前空間情報をメモリに保持し，永続的ではありません。これは，VisiNaming サービスがデフォルトで使用するアダプタです。

JDBC アダプタ

リレーショナルデータベースは JDBC を介してサポートされます。VisiNaming サービス JDBC アダプタに対しては，次のデータベースが動作保証されています。

- JDataStore 7
- Oracle 10G，リリース 1
- Sybase 11.5

- Microsoft SQLServer 2000
- DB2 8.1
- InterBase 7

次のどちらかに該当する場合は，複数の VisiNaming サービスインスタンスが同じバックエンドリレーショナルデータベースを使用できます。

- VisiNaming サービスインスタンスが互いに独立しており，異なるファクトリ名を使用している。
- VisiNaming サービスインスタンスがすべて同じ VisiNaming サービスクラスタに属している。

DataExpress アダプタ

JDBC アダプタのほかには DataExpress アダプタがあります。これは，JDataStore データベースにネイティブにアクセスできます。JDBC を介して JDataStore にアクセスするよりもかなり高速になりますが，DataExpress アダプタにはいくつかの制限があります。DataExpress アダプタは，ネーミングサーバーと同じマシンで動作しているローカルデータベースしかサポートしません。リモートの JDataStore データベースにアクセスするには，JDBC アダプタを使用する必要があります。

JNDI アダプタ

JNDI アダプタもサポートしています。Sun の JNDI（Java naming and directory interface）は，企業全体の複数のネーミングサービスとディレクトリサービスに標準のインターフェースを提供します。JNDI は SPI（Service Provider Interface）を持ち，さまざまなネーミングおよびサービスベンダーがこの SPI にしたがっています。Netscape LDAP Server，Novell NDS，WebLogic Tengah などでは，さまざまな SPI モジュールを使用できます。VisiNaming サービスは JNDI をサポートするので，これらのネーミングサービス，ディレクトリサービス，およびその他の将来の SPI プロバイダへのアクセスの可搬性が確保されます。

VisiNaming JNDI アダプタは，次の LDAP インプリメンテーションで動作が保証されます。

- iPlanet Directory Server 5.0
- OpenLdap 2.2.26

LDAP を使用するには，Sun および Netscape JNDI Driver バージョン 1.2 を使用する必要があります。

設定と使用

バックストアアダプタは取り替え可能なので，VisiNaming サービスの起動時に設定（プロパティ）ファイルに保存されているユーザー定義情報に基づいて，使用するアダプタの種類を指定できます。インメモリアダプタ以外のアダプタは，永続性を提供します。インメモリアダプタは，名前空間をすべてメモリに保持する軽量な VisiNaming サービスが必要な場合に使用してください。

メモ　VisiNaming サービスの現在のバージョンでは，VisiNaming サービスの実行中に設定を変更することはできません。設定を変更するには，サービスをいったん停止して設定ファイルを変更してから，VisiNaming サービスを再起動する必要があります。

プロパティファイル

VisiNaming サービスの場合，基本的には，使用するアダプタの選択やアダプタの具体的な設定は，VisiNaming サービスのプロパティファイルで処理します。すべてのアダプタに共通するデフォルトのプロパティは次のとおりです。

表 16.5　すべてのアダプタに共通するデフォルトのプロパティ

プロパティ	デフォルト値	説明
vbroker.naming.backingStoreType	InMemory	使用するネーミングサービスアダプタのタイプを指定します。このプロパティは，VisiNaming サービスで使用するバックストアのタイプを指定します。有効なオプションは，InMemory，JDBC，Dx，JNDI です。デフォルトは InMemory です。
vbroker.naming.cacheOn	0	ネーミングサービスキャッシュを使用するかどうかを指定します。値を 1 にすると，キャッシュが有効になります。
vbroker.naming.cache.connectString		ネーミングサービスキャッシュが有効で (vbroker.naming.cacheOn=1)，ネーミングサービスインスタンスがクラスタモードまたはマスター／スレーブモードで設定されている場合は，このプロパティが必要です。これにより，イベントサービス／VisiNotify のインスタンスを <hostname>:<port> の形式で検索できます。たとえば，次のようになります。 vbroker.naming.cache.connectString=127.0.0.1:14500 キャッシング機能を有効にし，適切なプロパティを設定する方法の詳細は，197 ページの「キャッシング機能」を参照してください。
vbroker.naming.cache.size	2000	このプロパティは，ネーミングサービスキャッシュのサイズを指定します。値を大きくすると，より多くのデータをキャッシュできますが，メモリの消費量が増大します。
vbroker.naming.cache.timeout	0（制限なし）	このプロパティには，前回アクセスされてからデータを保持する時間（秒）を指定します。この時間が経過すると，キャッシュのデータはメモリを解放するために破棄されます。キャッシュに保持されるエントリは，LRU（使用頻度が低い）順に削除されます。

JDBC アダプタのプロパティ

次に，JDBC アダプタのプロパティについて説明します。

```
vbroker.naming.backingStoreType
```

このプロパティは，JDBC に設定します。JDBC アダプタの場合は，poolSize，jdbcDriver，url，loginName，および loginPwd プロパティも設定する必要があります。

```
vbroker.naming.jdbcDriver
```

このプロパティでは，バックストアとして使用するデータベースへアクセスするために必要な JDBC ドライバを指定します。VisiNaming サービスは，指定された適切な JDBC ドライバをロードします。デフォルトは，Java DataStore JDBC ドライバです。

JDBC ドライバクラス名	説明
com.borland.datastore.jdbc.DataStoreDriver	JDataStore JDBC ドライバ 7.0
com.sybase.jdbc2.jdbc.SybDriver	Sybase ドライバ (jConnect バージョン 5.0)
oracle.jdbc.driver.OracleDriver	Oracle ドライバ（classes12.zip バージョン 8.1.7.0.0）
interbase.interclient.Driver	Interbase ドライバ（InterClient.jar バージョン 3.0.12）

JDBC ドライバクラス名	説明
weblogic.jdbc.mssqlserver4.Driver	WebLogic MS SQLServer JDBC ドライバ（バージョン 5.1）
com.ibm.db2.jcc.DB2Driver	IBM DB2 ドライバ（db2jcc.jar バージョン 1.2.117）

vbroker.naming.minReconInterval

このプロパティは，ネーミングサービスのデータベース再接続再試行時間を秒単位で設定します。デフォルト値は，30 です。この要求と最後の接続時刻の間の時間間隔がこのプロパティで設定された値未満の場合，ネーミングサービスは要求を無視し，CannotProceed 例外を生成します。このプロパティの有効値は 0 以上の整数です。このプロパティ値が 0 の場合に接続が解除されると，VisiNaming サービスは，要求があるたびにデータベースに再接続しようとします。

vbroker.naming.loginName

このプロパティは，データベースに関連付けられているログイン名です。デフォルトは VisiNaming です。

vbroker.naming.loginPwd

このプロパティは，データベースに関連付けられていログイン用のパスワードです。デフォルト値は，VisiNaming です。

vbroker.naming.poolSize

バックストアとして JDBC アダプタを使用する場合は，このプロパティで接続プールのデータベース接続数を指定します。デフォルト値は 5 ですが，データベースで処理できる最大値まで増やすことができます。VisiNaming サービスに多くの要求があると予想される場合は，この値を増やしてください。

vbroker.naming.url

このプロパティでは，アクセスするデータベースの場所を指定します。設定内容は，使用するデータベースによって異なります。デフォルトは JDataStore で，データベースの場所は現在のディレクトリであり，名前は rootDB.jds です。rootDB.jds 以外の名前を使用することもできますが，名前の変更に合わせて設定ファイルも更新する必要があります。

URL 値	説明
jdbc:borland:dslocal:<db_name>	JDataStore URL
jdbc:sybase:Tds:<host>:<port>/<db_name>	Sybase URL
jdbc:oracle:thin:@<host>:<port>:<sid>	Oracle URL
jdbc:interbase://<server>/<full_db_path>	Interbase URL
jdbc:weblogic:mssqlserver4:<db_name>@<host>:<port>	WebLogic MS SQLServer URL
jdbc:db2://<host_name>:<port-number>/<db_name>	IBM DB2 URL
<full_path_JDataStore_db>	DataExpress3 URL（ネイティブドライバの場合）

[1]JDBC を介して InterBase にアクセスするには，その前に InterServer を起動する必要があります。InterBase サーバーがローカルホストにある場合は，<server> を localhost と指定します。これ以外の場合は，ホスト名を指定します。InterBase データベースが Windows NT 上にある場合は，<full_db_path> を driver:\\dir1\dir2\\db.gdb と指定します。ここで，最初の \ は 2 番めのバックスラッシュ \ をエスケープします。InterBase データベースが UNIX 上にある場合は，<full_db_path> を \dir1\dir2\db.gdb と指定します。詳細は，http://www.borland.co.jp/interbase/ を参照してください。

[2] JDBC を介して DB2 にアクセスするには，その前に Client Configuration Assistant で，エリアス <db_name> をデータベースに登録する必要があります。データベースの登録後は，vbroker.naming.url プロパティに対して <host> と <port> を指定する必要はありません。

[3] JDataStore データベースが Windows 上にある場合，<full path of the JDataStore database> は Driver:\\dir1\\dir2\\db.jds になります。ここで，最初の \ は 2 番めのバッ

クスラッシュ \ をエスケープします。JDataStore データベースが UNIX 上にある場合，<full path of the JDataStore database> は /dir1/dir2/db.jds になります。

DataExpress アダプタのプロパティ

次に，DataExpress アダプタのプロパティについて説明します。

プロパティ	説明
vbroker.naming.backingStoreType	このプロパティは，Dx に設定します。
vbroker.naming.loginName	このプロパティは，データベースに関連付けられているログイン名です。デフォルトは VisiNaming です。
vbroker.naming.loginPwd	このプロパティは，データベースに関連付けられていログイン用のパスワードです。デフォルト値は，VisiNaming です。
vbroker.naming.url	このプロパティは，データベースの場所を指定します。

JNDI アダプタのプロパティ

次に示すのは，JNDI アダプタの設定ファイルで指定できる設定のサンプルです。

設定値	説明
vbroker.naming.backingStoreType=JNDI	バックストアのタイプを指定します。JNDI アダプタの場合は JNDI です。
vbroker.naming.loginName=<user_name>	JNDI バックサーバーでのユーザーログイン名です。
vbroker.naming.loginPwd=<password>	JNDI バックサーバーユーザーのパスワードです。
vbroker.naming.jndiInitialFactory=com.sun.jndi.ldap.LdapCtxFactory	JNDI 初期ファクトリを指定します。
vbroker.naming.jndiProviderURL=ldap://<hostname>:389/<initial root context>	JNDI プロバイダの URL を指定します。
vbroker.naming.jndiAuthentication=simple	JNDI バックサーバーがサポートしている JNDI 認証のタイプを指定します。

OpenLDAP の設定

OpenLDAP は，サポートされている VisiNaming バックエンドストアの 1 つです。OpenLDAP を使用する場合は，OpenLDAP サーバーで追加の設定が必要です。次の作業を行う必要があります。

1. OpenLDAP サーバーの設定ファイルに corba.schema を追加します（デフォルトは slapd.conf）。corba.schema は OpenLDAP サーバーのインストールに付属しています。
2. OpenLDAP の設定ファイルに openldap_ns.schema を追加します。openldap_ns.schema は VisiBroker に付属しており，次の場所にあります。

   ```
   <install-dir>/etc/ns_schema/
   ```

メモ　ディレクトリサーバーにスキーマ／属性を追加するには，対応する権限が必要です。

キャッシング機能

キャッシング機能を有効にすると，バックストア使用時のネーミングサービスのパフォーマンスを改善できます。たとえば JDBC アダプタの場合，リゾルブやバインド操作があるたびにデータベースに直接アクセスすると，相対的に速度は低下してしまいます。しかし，操作結果をキャッシングすることで，データベースへのアクセス回数を減らすことができます。また，バックストアのパフォーマンスに向上が見られるのは，同じデータに何度もアクセスする場合だけです。

メモ　ネーミングサービスクラスタモードまたはマスター／スレーブモードでは，複数のネーミングサービスインスタンスが同じバックストアにアクセスできます。この 2 つのモードでキャッシング機能を使用するには，各ネーミングサービスインスタンスを vbroker.naming.cache.connectString プロパティで特別に設定する必要があります。VisiBroker イベントサービスまたは VisiNotify を使用して，さまざまなネーミングサービスインスタンス間のキャッシング機能が調整されます。

キャッシング機能を有効にするには，設定ファイルで次のプロパティを設定します。

```
vbroker.naming.cacheOn=1
```

複数のクラスタモードまたはマスター／スレーブモードのネーミングサービスインスタンスがキャッシュにアクセスする場合は，vbroker.naming.cache.connectString プロパティを設定して，ネーミングサービスがイベントサービス（または VisiNotify）を見つけることができるようにします。

vbroker.naming.cache.connectString の形式は次のとおりです。

```
vbroker.naming.cache.connectString=<host>:<port>
```

ここで，<host> は VisiBroker イベントサービスが実行されているコンピュータのホスト名または IP アドレスです。また，<port> は VisiBroker イベントサービス／ VisiNotify が使用するポート（デフォルトでは，イベントサービスの場合は 14500，VisiNotify の場合は 14100）です。

たとえば，次のようになります。

```
vbroker.naming.cache.connectString=127.0.0.1:14500
```

または

```
vbroker.naming.cache.connectString=myhost:14100
```

ホストのアドレスが IPv6 形式の場合は，アドレスをブラケットで囲んでください。

メモ　VisiBroker イベントサービス（バージョン 6.5 以降）は，ネーミングサービスインスタンスを起動する前に起動しておく必要があります。かわりに VisiNotify を使用する場合は，VisiNotify を起動しておく必要があります。ネーミングサービスインスタンスを起動する前に，（デフォルト名が使用されるように）チャネル名を指定せずにイベントサービス／ VisiNotify を起動してください。

キャッシュの調整が必要な場合は，次のプロパティを設定します。

```
vbroker.naming.cache.size
vbroker.naming.cache.timeout
```

キャッシング機能のプロパティについては，195 ページの「プロパティファイル」を参照してください。

キャッシング機能に関する重要事項

一貫性のある設定は重要です。クラスタ内のすべてのネーミングサービスインスタンスが一貫した方法でキャッシング機能を使用するように設定することが，非常に重要です。クラスタを構成する**すべて**のネーミングサービスインスタンスがキャッシング機能を使用するか，または**まったく使用しない**かのどちらかにする必要があります。他のネーミングサービスインスタンスがキャッシング機能を使用しない場合に，一部のインスタンスがキャッシング機能を使用すると，クラスタの動作に矛盾が生じます。これは，マスター／スレーブモードに設定されているネーミングサービスの場合も同じです。マスターがキャッシング機能を使用するように設定されている場合は，スレーブもキャッシング機能を使用するように設定する必要があり，その逆も同様です。

分散キャッシュはイベントサービス／ VisiNotify に依存します。ネーミングサービスのクラスタモード（またはマスター／スレーブモード）でキャッシング機能を使用する場合，分散キャッシュは複数のネーミングサービスインスタンス間で同期をとる必要があります。それには，イベントサービス（または VisiNotify）を使用します。このような設定では，キャッシュされたデータが無効な場合があることに注意してください。データの品質は，イベントサービス／ VisiNotify の状態によって異なります。許容できない品質の場合，アプリケーションでキャッシング機能を使用しないでください。テストを実行して，分

散キャッシング機能がアプリケーションに適しているかどうかを個別に判断することをお勧めします。

クラスタ

VisiBroker では，複数のオブジェクトバインディングを 1 つの名前に関連付けるためのクラスタリング機能をサポートしています。この機能を使用して，VisiNaming サービスはクラスタにある複数のバインディング間で負荷分散を実行できます。負荷分散の基準は，クラスタを作成するときに指定します。負荷分散の基準を指定した後，クライアントがクラスタに対して名前とオブジェクトとのバインディングを解決すると，クラスタのサーバーメンバー間で負荷が分散されます。これらのオブジェクトバインディングクラスタを 203 ページの「VisiNaming サービスクラスタによるフェイルオーバーと負荷分散」と混同しないでください。

クラスタは，1 つの Name とオブジェクトリファレンスのグループを関連付けるマルチバインドメカニズムです。クラスタは，ClusterManager リファレンスを介して作成します。その際，ClusterManager の create_cluster メソッドは，使用する基準を指定するための文字列パラメータを受け取ります。このメソッドは，クラスタへのリファレンスを返します。クラスタメンバーの追加，除去，および巡回はこのリファレンスで行います。クラスタの構造を決定したら，名前を指定してリファレンスを VisiNaming サービスの任意のコンテキストにバインドできます。その場合，Name に対する後続の解決動作ではクラスタ内の特定のオブジェクトリファレンスが返ります。

クラスタリングの基準

VisiNaming サービスは，デフォルトのクラスタで SmartRoundRobin 基準を使用します。いったんクラスタを作成すると，その基準を変更することはできません。ユーザー定義の基準は現在サポートされていませんが，サポートする基準は将来増やしていく予定です。SmartRoundRobin はいくつかの検証を実行します。これにより，CORBA オブジェクトリファレンスがアクティブであること，つまりそのオブジェクトリファレンスが準備完了状態にある CORBA サーバーを参照していることが保証されます。

Cluster と ClusterManager インターフェース

クラスタはネーミングコンテキストに似ていますが，コンテキストにはクラスタに関係のないメソッドがあります。たとえば，ネーミングコンテキストをクラスタにバインドしても意味がありません。クラスタは，ネーミングコンテキストではなく，オブジェクトリファレンスのセットを保持しているからです。ただし，クラスタインターフェースは，NamingContext インターフェースと多くの同じメソッド（bind，rebind，resolve，unbind，list など）を共有します。この共通のオペレーションは，主にグループの操作に関係するものです。クラスタ固有のオペレーションは pick だけです。両者の重要な相違点のもう 1 つは，クラスタが複合名をサポートしないことです。クラスタは階層ディレクトリ構造を持たず，オブジェクトリファレンスをフラットな構造で保存するので，単一要素の名前だけを使用します。

Cluster インターフェースの IDL 仕様

```
CosNamingExt module {
   typedef sequence<Cluster> ClusterList;
   enum ClusterNotFoundReason {
      missing_node,
      not_context,
      not_cluster_context
   };
   exception ClusterNotFound {
```

```
      ClusterNotFoundReason why;
      CosNaming::Name rest_of_name;
   };
   exception Empty {};
   interface Cluster {
      Object select() raises(Empty);
      void bind(in CosNaming::NameComponent n, in Object obj)
         raises(CosNaming::NamingContext::CannotProceed,
            CosNaming::NamingContext::InvalidName,
            CosNaming::NamingContext::AlreadyBound);
      void rebind(in CosNaming::NameComponent n, in Object obj)
         raises(CosNaming::NamingContext::CannotProceed,
            CosNaming::NamingContext::InvalidName);
      Object resolve(in CosNaming::NameComponent n)
         raises(CosNaming::NamingContext::NotFound,
            CosNaming::NamingContext::CannotProceed,
            CosNaming::NamingContext::InvalidName);
      void unbind(in CosNaming::NameComponent n)
         raises(CosNaming::NamingContext::NotFound,
            CosNaming::NamingContext::CannotProceed,
            CosNaming::NamingContext::InvalidName);
      void destroy()
         raises(CosNaming::NamingContext::NotEmpty);
      void list(in unsigned long how_many,
         out CosNaming::BindingList bl,
         out CosNaming::BindingIterator BI);
};
```

ClusterMangager インターフェースの IDL 仕様

```
CosNamingExt module {
   interface ClusterManager
      Cluster create_cluster(in string algo);
      Cluster find_cluster(in CosNaming::NamingContext ctx, in CosNaming::Name n)
         raises(ClusterNotFound, CosNaming::NamingContext::CannotProceed,
      CosNaming::NamingContext::InvalidName);
      Cluster find_cluster_str(in CosNaming::NamingContext ctx, in string n)
         raises(ClusterNotFound, CosNaming::NamingContext::CannotProceed,
      CosNaming::NamingContext::InvalidName);
      ClusterList clusters();
   };
};
```

NamingContextExtExtended インターフェースの IDL 仕様

NamingContextExt を拡張した NamingContextExtExtended は，暗黙的なクラスタからオブジェクトリファレンスを削除するために必要ないくつかのオペレーションを提供します。これらのオペレーションを使用するには，NamingContext を NamingContextExtExtended にナローイングする必要があります。ただし，これらのオペレーションは VisiBroker 専用です。

```
module CosNamingExt {
  interface NamingContextExtExtended : NamingContextExt {
    void unbind_from_cluster(in Name n, in Object obj)
     raises(NotFound, CannotProceed, InvalidName);
    boolean is_ncluster_type(in Name n, out Object cluster)
     raises(NotFound, CannotProceed, InvalidName);
  };
}
```

```
unbind_from_cluster()
```

unbind_from_cluster() メソッドを使用して，クラスタ内の特定のオブジェクトをバインド解除できます。オブジェクトの論理名（「London.Branch/Jack.SavingAccount」など）およびバインド解除するオブジェクトリファレンスをこのメソッドに渡す必要があります。クラスタ内のオブジェクトの数が 0 になった場合は，クラスタも削除されます。

クラスタ内の無効なオブジェクトリファレンスの自動削除が必要でない場合は，このメソッドが便利です。アプリケーション固有の規則に基づいてクラスタ内のオブジェクトをバインド解除するには，このメソッドを呼び出します。

メモ unbind_from_cluster() メソッドは，VisiNaming Service が暗黙的クラスタリングモードで実行されており，無効なオブジェクトリファレンスの自動削除が無効になっている場合にのみ使用できます。つまり，VisiNaming Service 側で次の 2 つのプロパティが設定されている必要があります。

```
vbroker.naming.smrr.pruneStaleRef=0
vbroker.naming.propBindOn=1
```

is_ncluster_type()

is_ncluster_type() メソッドを使用して，コンテキストがクラスタタイプかどうかをチェックできます。オブジェクトの論理名をこのメソッドに渡す必要があります。コンテキストがクラスタタイプである場合は，true を返し，第 2 引数の値にクラスタオブジェクトを設定します。コンテキストがクラスタタイプでない場合は，false を返し，第 2 引数の値に null に設定します。

クラスタの作成

クラスタを作成するには，ClusterManager インターフェースを使用します。ネーミングサーバーを起動すると，1 つの ClusterManager オブジェクトが自動的に作成されます。ClusterManager は各ネーミングサービスに対して 1 つだけです。ClusterManager の役割は，ネーミングサーバー内のクラスタの作成，取得，追跡です。一般的なクラスタ作成手順を次に示します。

1. クラスタオブジェクトの作成に使用するネーミングサーバーにバインドします。
2. get_cluster_manager メソッドをファクトリリファレンスで呼び出して Cluster Manager までのリファレンスを取得します。
3. 指定されたクラスタ基準でクラスタを作成します。
4. そのクラスタで，オブジェクトを Name にバインドします。
5. Cluster オブジェクトそのものを Name にバインドします。
6. 指定されたクラスタ基準のクラスタリファレンスを介して解決します。

```
. . .
ExtendedNamingContextFactory_var myFactory =
      ExtendedNamingContextFactory::_bind(orb, "NamingService");
ClusterManager_var clusterMgr = myFactory->get_cluster_manager();
Cluster_var clusterObj = clusterMgr->create_cluster("RoundRobin");
clusterObj->bind(new NameComponent("member1", "aCluster"), obj1);
clusterObj->bind(new NameComponent("member2", "aCluster"), obj2);
clusterObj->bind(new NameComponent("member3", "aCluster"), obj3);
NameComponent_var myClusterName = new NameComponent("ClusterName", "");
root->bind(myClusterName, clusterObj);
root->resolve(myClusterName); // クラスタの 1 つのメンバーが返されます。
root->resolve(myClusterName); // クラスタの次のメンバーが返されます。
root->resolve(myClusterName); // クラスタの最後のメンバーが返されます。
. . .
```

明示的クラスタと暗黙的クラスタ

VisiNaming サービスのクラスタリング機能を自動的に有効にすることができます。ただし，この機能を有効にすると，オブジェクトをバインドするためにクラスタが透過的に作

成されるので注意してください。使用される基準は，ラウンドロビン（総当たり）方式だけです。つまり，複数のオブジェクトがネーミングサーバー内の同じ名前にバインドされることがあります。逆に，その名前を解決すると，そのオブジェクトの 1 つが返されます。また，unbind オペレーションは，その名前に結び付けられているクラスタを破棄します。このような VisiNaming サービスは，CORBA 仕様に準拠していません。**Interoperable Naming Specification** では，複数のオブジェクトを同じ名前にバインドする機能の禁止を明示しています。これに準拠した VisiNaming サービスでは，クライアントが同じ名前で異なるオブジェクトにバインドしようとすると，AlreadyBound 例外が生成されます。ユーザーは，サーバーでこの機能を使用するかどうかを最初に決定し，その後もその決定内容に準拠する必要があります。

メモ　暗黙的なクラスタモードから明示的なクラスタモードへは切り替えないでください。バックストアが破棄されることがあります。

いったん暗黙的なクラスタリング機能とネーミングサーバーでクラスタリング機能を有効にした場合は，そのまま「有効」にしておく必要があります。この機能を有効にするには，設定ファイルで次のプロパティ値を定義します。

```
vbroker.naming.propBindOn=1
```

メモ　明示的クラスタリングと暗黙的クラスタリングのサンプルについては，次のディレクトリにあるサンプルコードを参照してください。

```
<install_dir>/examples/vbe/ins/implicit_clustering
<install_dir>/examples/vbe/ins/explicit_clustering
```

負荷分散

クラスタマネージャとスマートエージェントは，どちらにもラウンドロビン（総当たり）方式の負荷分散機能がありますが，その特性は大きく異なります。負荷分散は，スマートエージェントから透過的に取得できます。サーバーは，起動時に自動的に自分自身をスマートエージェントに登録します。これによって VisiBroker Edition 独自の方法で，クライアントが簡単にサーバーへのリファレンスを取得できます。ただし，グループやグループのメンバーの構成を決定することはできません。スマートエージェントがこの構成をすべて決定してしまいます。そこで，クラスタがその代替手段となります。所定の方法でクラスタのプロパティを定義し，クラスタを作成できます。これはクラスタが使用する基準を定義できるので，クラスタのメンバーをフレキシブルに選択できます。基準は作成時に確定しますが，クライアントはクラスタが存在する限り，クラスタのメンバーの追加や除去が可能です。

オブジェクトのフェイルオーバー

オブジェクトクラスタリングを使用する利点の 1 つは，VisiNaming サービス内でクラスタリングされたオブジェクトの間のフェイルオーバー機能です。これらのクラスタリングオブジェクトは，同じインターフェースをサポートします。このようなクラスタが作成され，ネーミングコンテキストにバインドされると，フェイルオーバーの動作は ORB によって透過的に処理されます。通常，このクラスタに対してネーミングサービスクライアントが解決を行う場合は，VisiNaming サービスがクラスタからメンバーを返します。クラッシュしたり，一時的に使用できないメンバーがクラスタにある場合は，ORB と VisiNaming サービスが，次のクラスタメンバーをクライアントに渡すことで，透過的にフェイルオーバーを実行します。これにより，高可用性とフォールトトレランスが保証されます。

オブジェクトクラスタリングを使用するフェイルオーバー機能は，次のディレクトリに含まれている例に示されています。

```
<install_dir>/examples/vbe/ins/cluster_failover
```

VisiNaming オブジェクトクラスタ内の無効なオブジェクトリファレンスの削除

VisiNaming サービス内のオブジェクトリファレンスは，サーバーが使用できなくなることで無効になる場合があります。暗黙的なオブジェクトクラスタリングには，無効なリファレンスの削除の設定に使用できるさまざまな戦略が用意されています。ただし，この削除機能は，スマートラウンドロビン技術を使用する暗黙的なクラスタリングでのみ動作します。VisiNaming サービスは，`vbroker.naming.smrr.pruneStaleRef` プロパティによる削除設定付きで起動されます。このプロパティの値は，0，1（デフォルト），2 のいずれかです。削除機能の動作は次のように理解できます。

VisiNaming サービスは，名前とオブジェクトリファレンスの間のマッピングをメモリに保持します。クライアントが名前に基づいてオブジェクトリファレンスを要求すると，VisiNaming が名前を解決し，IOR を修正して，オブジェクトリファレンスをクライアントに渡します。IOR への修正は，このオブジェクトリファレンスによって表されるサーバーが使用不可能な場合に，クライアント ORB（このオブジェクトリファレンスが渡される ORB）が VisiNaming サービスに戻って別のオブジェクトリファレンスを探す（他の候補へのフェイルオーバー）ためのロジックの適用に関連します。クライアントがサーバーを発見できず，VisiNaming サービスに戻った場合，VisiNaming は，そのオブジェクトリファレンスを無効とマークします。

`vbroker.naming.smrr.pruneStaleRef` プロパティの値にしたがって，VisiNaming は，オブジェクトリファレンスを維持するか削除するかを決定します。指定できる値は次のとおりです。

- `vbroker.naming.smrr.pruneStaleRef =0`
  この場合，オブジェクトリファレンスが無効であることが検出されると，VisiNaming は，それを無効とマークするだけです。メモリからは削除しません。ただし，サーバーが同じ名前でオブジェクトリファレンスを再バインドしない限り，VisiNaming は，このリファレンスをクライアントに渡さなくなります。
- `vbroker.naming.smrr.pruneStaleRef =1`
  クライアントが VisiNaming サービスに戻り，オブジェクトリファレンスが無効であることが示されると，VisiNaming サービスは，ただちにメモリと永続的バックストア（バックストアを使用している場合）の両方からオブジェクトリファレンスを削除します。
- `vbroker.naming.smrr.pruneStaleRef =2`
  この場合，VisiNaming は，クライアントに渡す前に IOR を修正しません。クライアントがオブジェクトリファレンスで表されたサーバーにコンタクトできない場合は，クライアント ORB が OBJECT_NOT_EXISTS 例外をクライアントアプリケーションに生成します。VisiNaming サービスは，アクティブなオブジェクトリファレンスをクライアントアプリケーションに提供することを保証しません。

VisiNaming サービスクラスタによるフェイルオーバーと負荷分散

VisiNaming サービスの複数のインスタンスをクラスタリングして，負荷分散とフェイルオーバーに提供できます。この VisiNaming サービスインスタンスのクラスタを 199 ページの「クラスタ」で説明したオブジェクトバインディングのクラスタリングと混同しないでください。クライアントは，クラスタを構成する VisiNaming サービスインスタンスのいずれかにバインドでき，このクラスタによって複数の VisiNaming サービスインスタンス間で負荷を共有できます。特定の VisiNaming サービスインスタンスが無効になるか終了すると，そのクライアントは，同じクラスタ内の別の VisiNaming サービスインスタンスに自動的にフェイルオーバーします。

クラスタ内の VisiNaming サービスのすべてのインスタンスは，永続的バックストアにある基底の共通データを使用する必要があります。`vbroker.naming.cache.connectString` プロパティを介してネーミングサービスインスタンスで VisiBroker イベントサービス（また

は VisiNotify）インスタンスを使用できる場合は，キャッシング機能をネーミングサービスインスタンスで使用できます。バックストアの選択については多少の制限があります。詳細は，下の「メモ」を参照してください。

フェイルオーバーの発生は，クライアントにとっては透過的です。ただし，スレーブネーミングサーバーのサーバーオブジェクトは要求の着信によってオンデマンドでアクティブ化される必要があるので，多少の遅延が生じる可能性があります。また，反復子参照のような一時的なオブジェクトリファレンスは無効になります。一時的な反復参照を使用するクライアントは参照が無効になる事態を予想して対処しているため，これは異常ではありません。一般に VisiNaming サービスは，リソースを大量に消費する反復子オブジェクトを過度に保存することはなく，いつでもクライアントの反復子参照を無効にする可能性があります。これらの一時的な参照を除き，永続的な参照を使用するその他のクライアント要求はすべて VisiNaming サービスインスタンスに再送されます。

VisiNaming サービスクラスタのほかに，マスター／スレーブモデルもサポートされています。これは，2 つの VisiNaming サービスインスタンスで構成される特殊なクラスタで，フェイルオーバーが必要な場合には便利です。この 2 つの VisiNaming サービスインスタンスは，アクティブモードのマスターおよびスタンバイモードのスレーブとして，同時に実行する必要があります。両方の VisiNaming サービスがアクティブな場合，VisiNaming サービスを使用しているクライアントは常にマスターを優先します。マスターが予期せず終了した場合は，スレーブ VisiNaming サービスが機能を引き継ぎます。マスターからスレーブへの切り替えは，クライアントに透過的にシームレスに行われます。ただし，スレーブ VisiNaming サービスはマスターサーバーにはなりません。そのかわり，マスターサーバーが利用できなくなると，一時的にバックアップを提供します。ユーザーは，マスターサーバーの回復に必要な回復アクションを行う必要があります。マスターが回復した後は，新しいクライアントからの要求だけがマスターサーバーに送信されます。すでにスレーブネーミングサーバーにバインドされているクライアントが自動的にマスターに切り替わることはありません。

メモ　すでにスレーブネーミングサーバーにバインドされているクライアントでは，提供されるフェイルオーバーサポートは 1 レベルだけです。したがって，スレーブネーミングサーバーも停止してしまうと，VisiNaming サービスは使用できなくなります。

メモ　マスター／スレーブモードで設定された VisiNaming サービスクラスタは，JNDI アダプタまたは JDBC アダプタのいずれかを使用します。マスター／スレーブモードで設定されていないクラスタは，RDBMS 用の JDBC アダプタを使用する必要があります。クラスタリングされた各サービスは，明確に同じバックストアをポイントする必要があります。クラスタのバックストアの設定については，193 ページの「取り替え可能なバックストア」を参照してください。

VisiNaming サービスクラスタの設定

クラスタを構成する VisiNaming サービスインスタンスは，関連プロパティを下記のサンプルコードで示すように設定して開始する必要があります。設定は，`enableSlave` プロパティと `slaveMode` プロパティを使用してクラスタモードに設定します。クラスタを構成する VisiNaming サービスのインスタンスは，`serverAddresses` プロパティで指定されるホストおよびポート上で開始する必要があります。このコードでは，サンプルクラスタの 3 つの VisiNaming サービスインスタンスに対するホストとポートのエントリを示しています。`serverNames` プロパティには，VisiNaming サービスインスタンスのファクトリ名を一覧表示します。これらの名前は一意であり，順番は `serverAddresses` プロパティと同じです。最後に，`serverClusterName` プロパティでクラスタの名前を指定します。

メモ　VisiBroker 6.0 から，VisiNaming サービスには，プロキシサポートのためのプロパティがいくつか組み込まれています。

- `vbroker.naming.proxyEnable` は，VisiNaming サービスがプロキシを使用できるようにします。このプロパティをオフにすると（デフォルトはオフ），VisiNaming サービスは，プロキシ用の他のネーミングサービスプロパティを無視します。

- vbroker.naming.proxyAddresses は，クラスタ内の各ネーミングサービスにプロキシホストとプロキシポートを提供します。proxyAddresses の順番は serverAddresses と同じです。

C++ クライアントが VisiNaming サービスクラスタの負荷分散機能やフェイルオーバー機能を利用するには，論理プロパティ vbroker.naming.anyServiceOrder を設定する必要があります。osagent が使用されている場合，クライアントは，corbaloc メカニズムを使用して，クラスタ内の VisiNaming サービスインスタンスに解決する必要があります。

クラスタを構成するネーミングサービスインスタンスは，ネーミングサービスのキャッシング機能を活用できます。vbroker.naming.cacheOn プロパティと vbroker.naming.cache.connectString プロパティを使用して，ネーミングサービスクラスタのキャッシングを設定してください。詳細は，197 ページの「キャッシング機能」を参照してください。

次に，VisiNaming サービスクラスタの設定のサンプルコードを示します。

```
vbroker.naming.enableSlave=1
vbroker.naming.slaveMode=cluster
vbroker.naming.serverAddresses=host1:port1;host2:port2;host3:port3
vbroker.naming.serverNames=Server1:Server2:Server3
vbroker.naming.serverClusterName=ClusterX
vbroker.naming.proxyEnable=1 //1 以外の値は，プロキシが無効であることを示します。
vbroker.naming.proxyAddresses=proxyHost1:proxyPort1;proxyHost2:proxyPort2;proxyHost3:pro
xyPort3
```

メモ　vbroker.naming.proxyAddresses プロパティを使用する場合は，ホストとポートのペアの間をセミコロン（;）で区切ります。

マスター／スレーブモードでの VisiNaming サービスの設定

2 つの VisiNaming サービスが実行されている必要があります。一方をマスターに，もう一方をスレーブに指定します。両方のサーバーで同じプロパティファイルを使用できます。マスター／スレーブモードを設定する際のプロパティファイル内の関連プロパティの値を次の例に示します。

```
vbroker.naming.enableSlave=1
vbroker.naming.slaveMode=slave
vbroker.naming.masterServer=< マスターネーミングサーバー名 >
vbroker.naming.masterHost=< マスターのホスト IP アドレス >
vbroker.naming.masterPort=< マスターが監視するポート番号 >
vbroker.naming.slaveServer=< スレーブネーミングサーバー名 >
vbroker.naming.slaveHost=< スレーブのホスト IP アドレス >
vbroker.naming.slavePort=< スレーブネーミングサーバーのポートアドレス >
vbroker.naming.masterProxyHost=< マスターのプロキシホスト IP アドレス >
vbroker.naming.masterPortPort=< マスターのプロキシポート番号 >
vbroker.naming.slaveProxyHost=< スレーブのプロキシホスト IP アドレス >
vbroker.naming.slavePortPort=< スレーブのプロキシポート番号 >
```

メモ　マスターサーバーとスレーブサーバーの起動順序に指定はありません。

多数のクライアントが接続する環境での起動

多数のクライアントを抱える運用環境では，初期化中で要求を処理する準備ができていない起動段階にあるネーミングサービスにクライアントが接続しようとするのを防ぐことができない場合があります。起動が完了していないネーミングサービスは，受信した着信要求を破棄します。受信後に破棄する必要がある要求の数によっては，この動作が大量の CPU リソースを使用して，起動プロセス自体の妨げになり，ネーミングサービスの起動に時間がかかる場合があります。

この問題を解決し，ネーミングサービスが迅速に起動するようにするには，次の環境設定を使用します。

1 次のプロパティを true に設定します。

```
vbroker.se.iiop_tp.scm.iiop_tp.listener.deferAccept=true
```

2 次のプロパティを設定して，固定リスナーポートを使用します。

```
vbroker.se.iiop_tp.scm.iiop_tp.scm.listener.port=<port_number>
vbroker.se.iiop_tp.scm.iiop_tp.listener.portRange=0
```

これが正しく動作するためには，ネーミングサービスが実行されているホストで <port_number> を使用できる必要があります。portRange プロパティを 0 に設定する必要があります。このプロパティは，デフォルトの設定のままにすることも，明示的に設定することもできます。前述の port と portRange の両方の設定を適用する必要があることに注意してください。

このように設定されているネーミングサービスの起動中にクライアントが接続しようとすると，すべての接続が拒否されます。ネーミングサービスクラスタにアクセスしている場合，クライアントは初期化を完了した別のネーミングサービスにフェイルオーバーします。実行中のネーミングサービスがない場合，クライアントアプリケーションは OBJECT_NOT_EXIST 例外を受け取ります。

これらの設定は，SCM（サーバー接続マネージャ）単位で行います。必要な場合は，この機能を利用するようにすべての SCM を設定します。

ネーミングサービスで SSL を使用する場合は，前述の設定に加えて次の設定も必要です。

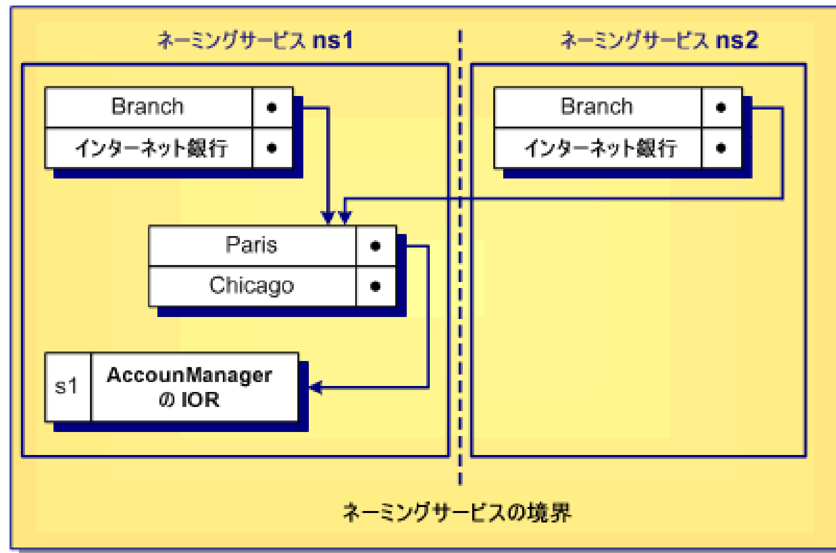
```
vbroker.se.iiop_tp.scm.ssl.listener.deferAccept=true
vbroker.se.iiop_tp.scm.ssl.listener.port=<port_number_for_ssl>
vbroker.se.iiop_tp.scm.ssl.listener.portRange=0
```

メモ deferAccept プロパティは，ネーミングサービスだけに使用してください。他のサービスまたはユーザーが記述したサーバーで使用すると，未定義の動作が起こる場合があります。

VisiNaming サービスフェデレーション

フェデレーションを使用すると，複数の VisiNaming サービスを 1 つの分散名前空間として動作するように設定できます。それには，ネーミングサービス内のネーミングコンテキストを他のネーミングサービスのネーミングコンテキスト内の名前にバインドする必要があります。これにより，1 つのオブジェクトに複数のネーミング階層からアクセスできるようになります。下の図には，ns1 および ns2 という 2 つのネーミングサービスのインスタンスが示されています。灰色のネーミングコンテキストは，対応するネーミングサービスの初期コンテキストです。AccountManager オブジェクト s1 は，ns1 の下のネーミングコンテキストに配置されています。

図 16.3 複数のアクセス階層を含むネーミングコンテキスト



図に示されているように，Paris を含むネーミングコンテキストは，ns1 ネーミングサービスの下にある Branch にバインドされ，ns2 ネーミングサービスの下にある Remote にもバインドされています。クライアントは，ns1: Branch/Paris/s1 または ns2: Branch/Paris/s1 のいずれかを解決することで，s1 に対する AccountManager オブジェクトの IOR を取得できます。どちらの場合も，同じ IOR が取得されます。

フェデレーションの設定は，上の例の ns2 のルートコンテキスト内の名前 Branch を，ns1 内の名前 Paris を含むネーミングコンテキストにバインドすることと同じで簡単です。次の場所にある例には，VisiNaming フェデレーションの使用が示されています。

```
<install_dir>/examples/vbe/ins/federation
```

VisiNaming サービスのセキュリティ

VisiBroker の VisiNaming サービスは，セキュリティサービスと統合されており，クライアント認証およびメソッドレベル承認という 2 つのレベルのセキュリティを提供します。これにより，どのクライアントが VisiNaming サービスを使用し，どのメソッドを呼び出すことができるかを詳細に制御できます。次のプロパティを使用して，セキュリティを有効または無効にしたり，セキュリティサービスを設定します。

表 16.6 VisiNaming サービスのセキュリティ関連プロパティ

プロパティ	値	デフォルト値	説明
vbroker.naming.security.disable	boolean	true	セキュリティサービスが無効かどうかを指定します。
vbroker.naming.security.authDomain	string	""	ネーミングサービスメソッドのアクセス承認に使用される承認ドメイン名を指定します。
vbroker.naming.security.transport	int	3	使用されるトランスポートを指定します。次の値がサポートされています。 ServerQoPPolicy.SECURE_ONLY=1 ServerQoPPolicy.CLEAR_ONLY=0 ServerQoPPolicy.ALL=3

表 16.6 VisiNaming サービスのセキュリティ関連プロパティ（続き）

プロパティ	値	デフォルト値	説明
vbroker.naming.security.requireAuthentication	boolean	false	ネーミングクライアント認証が必要かどうかを指定します。vbroker.naming.security.disable が true の場合は，このプロパティの値に関係なく，クライアント認証が行われません。
vbroker.naming.security.enableAuthorization	boolean	false	メソッドアクセス承認が有効かどうかを指定します。
vbroker.naming.security.requiredRolesFile	string	（なし）	プロテクトオブジェクト型の各メソッドの起動に必要な役割を含むファイルをポイントします。詳細は，209 ページの「メソッドレベル承認」を参照してください。

ネーミングクライアント認証

メモ　認証と承認の詳細は，「認証」と「承認」を参照してください。

SSL を使用するように VisiNaming を設定する

セキュリティ要件に応じて，さまざまなプロパティを設定して VisiNaming サービスを構成できます。セキュリティのプロパティとその説明の完全なリストについては，『セキュリティガイド』の「セキュリティプロパティ（Java）」または「セキュリティプロパティ（C++）」を参照してください。

重要　VisiNaming サービスでセキュリティを有効にするには，VisiSecure の有効なライセンスが必要です。

次に，SSL を使用するように VisiNaming サービスを設定するために使用できるプロパティの例を示します。

```
# ネーミングサービスでのセキュリティの有効化
vbroker.naming.security.disable=false

# セキュリティサービスの有効化
vbroker.security.disable=false

# SSL 層属性の設定
vbroker.security.peerAuthenticationMode=REQUIRE_AND_TRUST
vbroker.se.iiop_tp.scm.ssl.listener.trustInClient=true
vbroker.security.trustpointsRepository=Directory:./trustpoints

# ウォレットプロパティを使用して VisiNaming サービスの証明書 ID を設定
vbroker.security.wallet.type=Directory:./identities
vbroker.security.wallet.identity=delta
vbroker.security.wallet.password=Delt@$$$
```

クライアントが SSL を使用するように設定する方法については，『セキュリティガイド』の「セキュリティで保護された接続の作成（Java）」または「セキュリティで保護された接続の作成（C++）」を参照してください。

メモ　現在，IOR で corbaloc を使用してセキュリティおよびセキュリティで保護されたトランスポートを指定する方法はありません。したがって，SSL の使用時にネーミングクライアント側で corbaloc メソッドを使用して VisiNaming サービスをブートストラップすることはできません。ただし，SVCnameroot と文字列化された IOR メソッドは使用できます。

メソッドレベル承認

メソッドレベル承認は，次のオブジェクト型でサポートされています。

- Context
- ContextFactory
- Cluster
- ClusterManager

ネーミングサービスのセキュリティが有効で，enableAuthorization が true に設定されている場合は，これらのオブジェクト型の各メソッドについて承認されたユーザーだけが，対応するメソッドを起動できます。

ネーミングサービスは，メソッドレベル承認をサポートする 2 つの役割を事前に定義しています。

- 管理者の役割
- ユーザーの役割

必要に応じて，他の役割も定義できます。ユーザーは，この 2 つの役割についてロールマップを設定して，クライアントに役割を割り当てる必要があります。次は，ロールマップ定義の例です。

```
Administrator {
*CN=admin
*group=admin
uid=*, group=admin
}

User {
*CN=admin
*group=user
uid=*, group=user
}
```

上に示したオブジェクトの各メソッドを呼び出す前に，役割を指定する必要があります。それには，各メソッドに対して required_roles プロパティを使用します。次は，これらのプロパティと，対応するデフォルト値のリストです。これらのデフォルト値は，vbroker.naming.security.requiredRolesFile プロパティを使用して指定される required_roles を定義していない場合にのみ使用されます。これらのプロパティの値は，スペースまたはコンマで区切られます。

```
#
# naming_required_roles.properties
#

# すべての役割
required_roles.all=Administrator User

required_roles.Context.bind=Administrator
required_roles.Context.rebind=Administrator
required_roles.Context.bind_context=Administrator
required_roles.Context.rebind_context=Administrator
required_roles.Context.resolve=Administrator User
required_roles.Context.unbind=Administrator
required_roles.Context.new_context=Administrator User
required_roles.Context.bind_new_context=Administrator User
required_roles.Context.list=Administrator User
required_roles.Context.destroy=Administrator

required_roles.ContextFactory.root_context=Administrator User
required_roles.ContextFactory.create_context=Administrator
```

```
required_roles.ContextFactory.get_cluster_manager=Administrator User
required_roles.ContextFactory.remove_stale_contexts=Administrator
required_roles.ContextFactory.list_all_roots=Administrator
required_roles.ContextFactory.shutdown=Administrator

required_roles.Cluster.select=Administrator User
required_roles.Cluster.bind=Administrator
required_roles.Cluster.rebind=Administrator
required_roles.Cluster.resolve=Administrator User
required_roles.Cluster.unbind=Administrator
required_roles.Cluster.destroy=Administrator
required_roles.Cluster.list=Administrator User

required_roles.ClusterManager.create_cluster=Administrator
required_roles.ClusterManager.find_cluster=Administrator User
required_roles.ClusterManager.find_cluster_str=Administrator User
required_roles.ClusterManager.clusters=Administrator User
```

プログラムのコンパイルとリンク

ネーミングサービスを使用する C++ アプリケーションは，次の生成ファイルをインクルードする必要があります。

```
#include "CosNaming_c.hh"
#include "CosNamingExt_c.hh"
```

UNIX UNIX アプリケーションは，cosnm_r.so（マルチスレッド）ライブラリとリンクする必要があります。

Windows Windows アプリケーションは，cosnm_r.lib（cosnm_r_6.dll）（マルチスレッド）ライブラリとリンクする必要があります。

サンプルプログラム

VisiBroker には，VisiNaming サービスの使い方を紹介するサンプルプログラムがいくつか含まれています。それらのサンプルプログラムでは，VisiNaming サービスの新しい機能を解説しています。サンプルプログラムは <install_dir>/examples/vbe/ins ディレクトリにあります。また，<install_dir>/examples/vbe/basic/bank_naming ディレクトリには，VisiNaming サービスの基本的な使い方を示した Bank Naming サンプルがあります。

サンプルプログラムを実行する前に，183 ページの「VisiNaming サービスの実行」にしたがって VisiNaming サービスを起動しておいてください。さらに，次のいずれかの方法で，少なくとも 1 つのネーミングコンテキストを作成する必要があります。

- 183 ページの「VisiNaming サービスの実行」にしたがって VisiNaming サービスを起動します。初期コンテキストが自動的に作成されます。
- VisiBroker コンソールを使用します。
- クライアントを NamingContextFactory にバインドし，create_context メソッドを実行します。
- クライアントで ExtendedNamingContextFactory を実行します。

重要 ネーミングコンテキストが 1 つも作成されていない場合は，クライアントが CosNaming::NamingContext::bind を発行しようとすると，CORBA::NO_IMPLEMENT 例外が生成されます。

VisiNaming で使用する JdataStore HA の設定

ここでは，VisiNaming で使用する JDataStore High Available（HA）の設定について説明します。

このセクション全体で使用されている明示的なクラスタリングの例では，VisiNaming が JDataStore HA とともに使用されています。この例では，JDataStore が次のミラータイプを持つように設定されます。

- 1 つのプライマリミラー。読み取りと書き込みの両方のトランザクションを受け付けるミラータイプは，これだけです。一度に使用できるプライマリミラーは 1 つだけです。
- 3 つの読み取り専用ミラー。これらは読み取りトランザクションだけを実行でき，プライマリミラーデータベースとトランザクション的に整合性のあるビューを提供します。
- 1 つのディレクトリミラー。これは，ミラー設定テーブルなどのシステムセキュリティテーブルだけを含みます。これは，読み取り専用接続要求を読み取り専用ミラーにリダイレクトし，書き込み可能接続要求をプライマリミラーにリダイレクトします。また，すべての読み取り接続をすべての読み取り専用ミラーに負荷分散するという重要な機能を提供します。ただし，この機能は，このバージョンのネーミングサービスではサポートされていません。

JDataStore HA は，次の状況で自動フェイルオーバーをサポートします。

- プライマリミラーへの接続行われた後で障害が発生した場合，この接続は，接続オブジェクトのロールバックメソッドを呼び出すことで自動フェイルオーバーをトリガーできます。ただし，ここではこのシナリオについて説明しません。
- 接続要求が読み取り専用操作ではなく，現在のプライマリミラーにアクセスできない場合は，書き込み可能接続の要求を満たすために，ディレクトリミラーが自動的にフェイルオーバー操作をトリガーします。それには，読み取り専用ミラーの 1 つをプライマリミラーに昇格させます。

ディレクトリミラーに対して接続が行われる場合，VisiNaming は，JDataStore HA と一緒に動作します。プライマリミラーにアクセスできない場合は，読み取り専用ミラーの 1 つにフェイルオーバーします。VisiNaming の動作には，常に 1 つのプライマリミラーと少なくとも 2 つの読み取り専用ミラーが必要です。

メモ

- ディレクトリミラーは，ここで説明されているシナリオの単一障害ポイントです。別のディレクトリミラーをポイントするようにマスター／スレーブネーミングサービスを設定することで，より高い可用性を実現できます。
- JDataStoreHA は，JDataStore Version 7.04 以降でのみ動作します。

プライマリミラーの DB を作成する

JDataStore エクスプローラ（JdsExplorer）を使用して新しい DB を作成するには，［File］メニューから［New］を選択します。

各リスニング接続について JdsServer を呼び出す

この例では，次の接続を使用しています。

- JdsServer -port 2511（プライマリミラー）
- JdsServer -port 2512（読み取り専用ミラー）
- JdsServer -port 2513（読み取り専用ミラー）
- JdsServer -port 2514（読み取り専用ミラー）
- JdsServer -port 2515（ディレクトリミラー）

メモ　JdsServer は，必ず AutoFailover_* jds ファイルが配置されている場所から起動してください。vbroker.naming.url が正しく設定されていない限り，決して JdsServer を <JdataStore Install Directory>/bin から起動しないでください。必要な jar ファイルは次のとおりです。

- dbtools.jar
- dbswing.jar
- jdsremote.jar
- jdsserver.jar
- jds.jar

JDataStore HA を設定する

JDataStore HA を設定するには，次の手順にしたがう必要があります。

1 JDataStore を設定するために JDS サーバーコンソールを呼び出します。

2 JDataStore サーバーコンソールで，NS_AutoFailover という名前の新しいプロジェクトを作成します。

メモ　新しい DataSource を作成する場合は，プロトコルをリモートに設定し，サーバー名にコンピュータの IP を入れることをお勧めします。

3 （構造ペインで）DataSource1 をクリックして開き，編集できるようにします。

4 DataSource1 を右クリックし，コンテキストメニューから［Connect］を選択します。

5 （構造ペインで）［Mirror］を右クリックし，コンテキストメニューから［Add mirror］を選択します。

6 Mirror1 を編集して，Type プロパティを PRIMARY に設定します。
ホストがデフォルト値 localhost ではなく，ミラーが位置するコンピュータの IP を使用するように，各ミラーを設定します。ミラーごとに異なる IP アドレスを使用できますが，その IP のミラーに対して JdsServer を起動する必要があります。ディレクトリミラーは，各ミラーにアクセスできる必要があります。

7 Auto Failover および Instant Synchronization プロパティを true に設定します。

8 Mirror2 を追加し，それを読み取り専用ミラーに設定します。
事前に AutoFailover_Mirror2 を作成する必要はありません。これは，JDataStore HA によって自動的に作成されます。

9 すべての読み取り専用ミラーについて，Auto Failover および Instant Synchronization プロパティを true に設定します。

10 Mirror3 および Mirror4 について，上の 2 つの手順を繰り返します。

11 Mirror5 を追加し，それをディレクトリミラーに設定します。

12 このディレクトリミラーについて，Auto Failover および Instant Synchronization プロパティを false に設定します。

13 ［File］メニューから［Save Project "NS_AutoFailover.datasources"］を選択します。

14 （構造ペインで）［Mirrors］を右クリックし，［Synchronize all mirrors］を選択します。

15 （構造ペインで）［Mirror Status］をクリックし，Mirror1 の［Validate Primary］だけがチェックされていることを確認します。

VisiNaming の明示的クラスタリングの例を実行する

VisiNaming の明示的クラスタリングの例を実行するには，次の手順にしたがいます。

1 次のコマンドを使用して，osagent を起動します。

```
osagent
```

2 次のプロパティを含むファイルを autofailover.properties という名前で作成します。

```
vbroker.naming.backingStoreType=JDBC
vbroker.naming.poolSize=5
vbroker.naming.jdbcDriver=com.borland.datastore.jdbc.DataStoreDriver
vbroker.naming.url=jdbc:borland:dsremote://143.186.141.14/AutoFailover_Mirror5.jds
vbroker.naming.loginName=SYSDBA
vbroker.naming.loginPwd=masterkey
vbroker.naming.traceOn=0
vbroker.naming.jdsSvrPort=2515
vbroker.naming.logLevel=debug
```

3 次のコマンドを使用して，ネーミングサービスを起動します。

```
nameserv -VBJclasspath <JDS_Install>\lib\
jdsserver.jar -config autofailover.properties
```

4 次のコマンドを使用して，ServerA を起動します。

```
Server ServerA -ORBpropStorage ns_client.properties &
```

5 次のコマンドを使用して，ServerB を起動します。

```
Server ServerB -ORBpropStorage ns_client.properties &
```

6 次のコマンドを使用して，クライアントを起動します。

```
Client -ORBInitRef NameService=<nsIOR>
```

7 上の手順を数回繰り返し，出力を観察します。

1 つのプライマリミラーおよび 2 つの読み取り専用ミラーという最小要件を確認するには，次の手順にしたがいます。

1 2513 ポートを監視している JdsServer を停止します。

2 クライアント起動のステップを数回繰り返します。
動作は前の手順と同じです。

3 2514 ポートを監視している JdsServer を停止します。

4 クライアント起動のステップを数回繰り返します。
クライアントが BAD_PARAM 例外を生成し始めることがわかります。フェイルオーバーには少なくとも 2 つの読み取り専用ミラーを使用できる必要があるため，これは予測される動作です。

5 2513 および 2514 ポートを監視している JdsServer を再起動します。
これにより，3 つの読み取り専用ミラーを含む元の設定が復元されます。

JDatastore HA の自動フェイルオーバーを確認するには，次の手順にしたがいます。

1 プライマリミラーに設定されていたポート 2511 を監視する JdsServer を停止し，クライアント起動の手順を数回繰り返します。
読み取り専用ミラーの 1 つがプライマリミラーに昇格されることがわかります。

2 他のアクティブな読み取り専用ミラーを停止し，クライアント起動の手順を数回繰り返します。
フェイルオーバーには少なくとも 2 つの読み取り専用ミラーを使用できる必要があるため，クライアントが BAD_PARAM 例外を生成し始めます。

3 2511 ポートを監視している JdsServer を再起動します。
これは，以前にプライマリミラーに設定されていました。

4 クライアント起動のステップを数回繰り返します。

Mirror1 が読み取り専用ミラーに設定されることがわかります。このことは，ネーミングサービスが使用するディレクトリミラーへのデータソース接続を行うことで，JDS サーバーコンソールから確認できます。

VisiNaming のネーミングフェイルオーバーの例を実行する

VisiNaming サービスのフェイルオーバー機能を観察するには，次の例を実行します。

メモ　この手順を実行する前に，1 つのプライマリミラー（ポート 1111），3 つの読み取り専用ミラー（ポート 1112，1113，1114），および 2 つのディレクトリミラー（ポート 1115，1116）を含む JDataStore HA を作成します。

1 次のコマンドを使用して，osagent を起動します。

```
osagent
```

2 次のプロパティを含むファイルを autofailover.properties という名前で作成します。

```
# ネーミング
vbroker.naming.backingStoreType=JDBC
vbroker.naming.poolSize=5
vbroker.naming.jdbcDriver=com.borland.datastore.jdbc.DataStoreDriver
vbroker.naming.loginName=SYSDBA
vbroker.naming.loginPwd=masterkey
vbroker.naming.traceOn=0
vbroker.naming.jdsSvrPort=1115
#vbroker.naming.logLevel=debug
#enableslave のデフォルト値は 0 です。'1' はクラスタを示します。または
master-slave configuration
vbroker.naming.enableSlave=1
# マスタースレーブ設定を示します
vbroker.naming.slaveMode=slave
vbroker.naming.masterHost=143.186.141.14
vbroker.naming.masterPort=12372
vbroker.naming.masterServer=Master
vbroker.naming.slaveHost=143.186.141.14
vbroker.naming.slavePort=12373
vbroker.naming.slaveServer=Slave
```

3 次の例に示すように，JDataStore Server を起動します。

```
JdsServer.exe -port=1111
JdsServer.exe -port=1112
JdsServer.exe -port=1113
JdsServer.exe -port=1114
JdsServer.exe -port=1115
JdsServer.exe -port=1116
```

4 次のコマンドを使用して，ネーミングサービスマスターを起動します。

```
Server -ORBInitRef NameService=<Master Server IOR>
NamingClient -ORBInitRef NameService=<Master Server IOR>
```

5 次のコマンドを使用して，ネーミングサービススレーブを起動します。

```
Server -ORBInitRef NameService=<Slave Server IOR>
NamingClient -ORBInitRef NameService=<Slave Server IOR>
```

6 次のコマンドを使用して，サーバーを起動します。

```
vbj -DSVCnameroot=Master Server
```

7 次のコマンドを使用して，クライアントを起動します。

```
vbj -DSVCnameroot=Master Client
```

8 Enter キーを押して出力を観察します。
残高が値を返します。

9 ネーミングサービスマスターを停止し，上の手順を繰り返して出力を観察します。
残高が値を返します。

10 Enter キーを押して終了し，出力を観察します。
残高が値を返します。

2 つのディレクトリミラーが単一障害ポイントを処理するようすを観察するには, 次の手順にしたがいます。

1 1115 ポートを監視している JdsServer を停止します。
2 ネーミングサービスマスターを起動せず，クライアント起動の手順を繰り返します。
CannotProceed 例外が発生しますが，これは予測された動作です。
3 クライアント起動のステップを数回繰り返します。
残高が値を返します。値を返すことができるようになると，ポート 1117 を監視しているディレクトリミラーが使用されていることを観察できます。
4 クライアント起動の手順を繰り返し，Enter キーを 3 回押します。
残高が値を 3 回返します。

自動フェイルオーバーが 2 つのディレクトリミラーで動作するようすを観察するには，次の手順にしたがいます。

1 ポート 1111 を監視している JdsServer を停止します。
2 クライアント起動のステップを繰り返します。
3 Enter キーを 3 回押します。
値を返し始める前に，CannotProceed 例外が数回生成されます。値が返されると，ミラーの 1 つがプライマリミラーに昇格されていることがわかります。これは，JDS サーバーコンソールを使用して観察できます。

第 17 章

イベントサービスの使い方

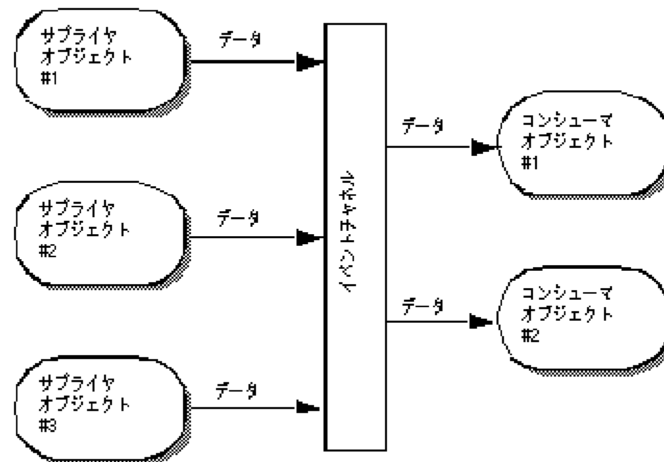
この節では，VisiBroker によるイベントサービスについて説明します。

メモ　OMG イベントサービスは，OMG 通知サービスに置き換えられています。VisiBroker イベントサービスは，下位互換性と軽量化の目的で引き続きサポートされています。ミッションクリティカルなアプリケーションには，VisiBroker VisiNotify の使用を強くお勧めします。詳細については，『VisiNotify の概要』を参照してください。

概要

イベントサービスパッケージは，オブジェクト間の通信を分離する機能を提供します。この機能では，**サプライヤ**通信モデルが提供されます。このモデルを使用すると，複数の**サプライヤオブジェクト**が複数の**コンシューマオブジェクト**にイベントチャネルを介して非同期にデータを送信できます。サプライヤ／コンシューマ通信モデルにより，ディスクが空き容量を使い果たしたなどの重要な状態の変化があれば，オブジェクトがこのようなイベントを必要とするほかのオブジェクトにそれを通知することができます。

図 17.1　サプライヤ／コンシューマ通信モデル



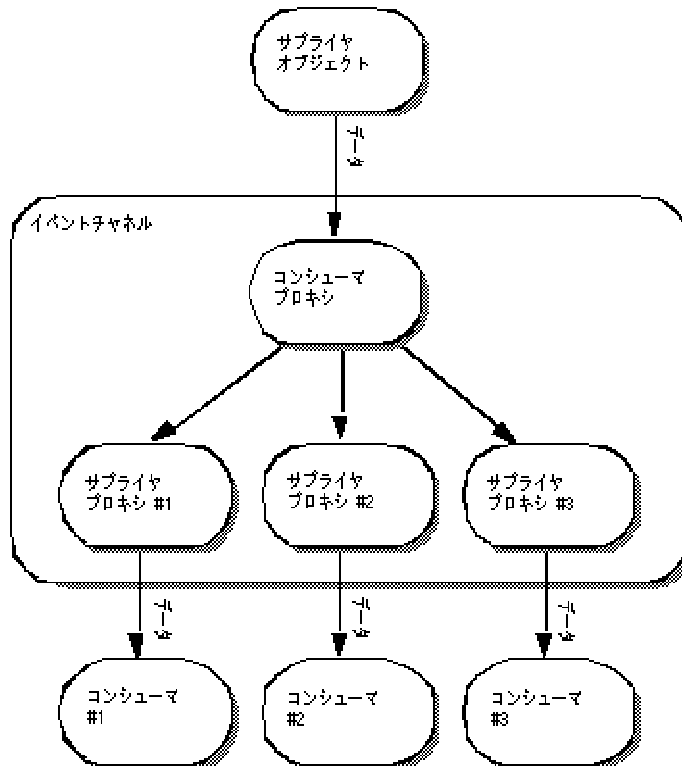
イベントチャネルを介して，3 つのサプライヤオブジェクトが 2 つのコンシューマオブジェクトと通信しているようすを示します。イベントチャネルへのデータの流れは，サプライヤオブジェクトによって処理され，イベントチャネルからのデータの流れは，コンシューマオブジェクトによって処理されます。上の図に示されている 3 つのサプライヤがそれぞれ毎秒 1 つのメッセージを送信する場合，各コンシューマは毎秒 3 つのメッセージを受け取り，イベントチャネルは毎秒合計 6 つのメッセージを転送します。

イベントチャネルは，イベントのコンシューマでもあり，サプライヤでもあります。サプライヤとコンシューマの間で通信されるデータは，`Any` クラスによって表され，任意の CORBA 型をタイプセーフな方法で渡すことができます。サプライヤオブジェクトとコンシューマオブジェクトは，標準の CORBA 要求を使用し，イベントチャネルを介して通信します。

プロキシコンシューマおよびプロキシサプライヤ

コンシューマとサプライヤは，**プロキシオブジェクト**を使用することにより，互いに完全に分離されています。それらは互いに直接対話するのではなく，`EventChannel` からプロキシオブジェクトを取得し，そのオブジェクトと通信します。サプライヤオブジェクトは**コンシューマプロキシ**を，またコンシューマオブジェクトは**サプライヤプロキシ**を取得します。`EventChannel` は，コンシューマおよびサプライヤのプロキシオブジェクト間のデータ転送を促進します。下の図は，1 つのサプライヤが複数のコンシューマにデータを配布するようすを示します。

図 17.2 コンシューマおよびサプライヤのプロキシオブジェクト



メモ 上に示したイベントチャネルは独立したプロセスとして示されていますが，サプライヤオブジェクトのプロセスの一部として実装される場合もあります。

OMG コモンオブジェクトサービス仕様

VisiBroker によるイベントサービスのインプリメンテーションは，次の 2 点を除くと，OMG コモンオブジェクトサービス仕様に準拠しています。

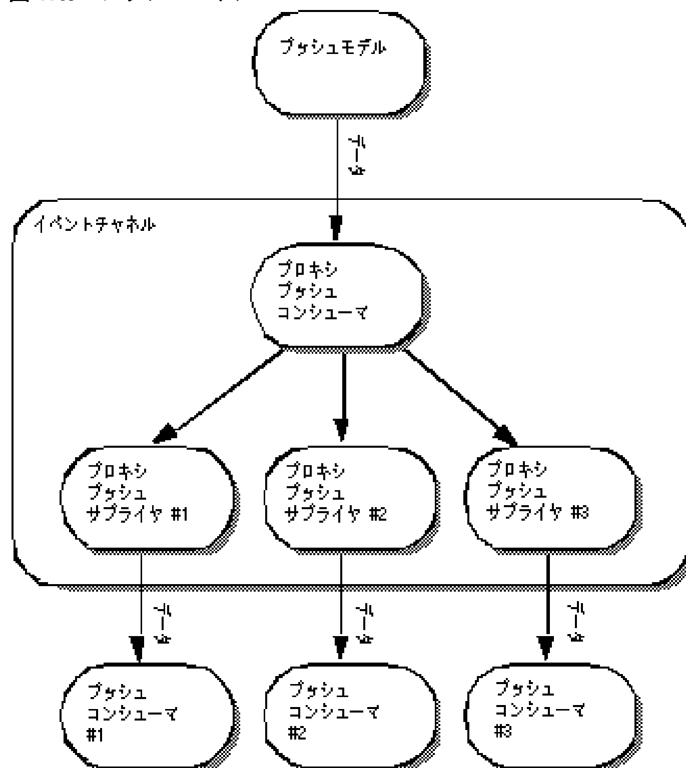
- VisiBroker によるイベントサービスは，共通イベントだけをサポートします。現在，VisiBroker によるイベントサービスでは，型付きのイベントはサポートされていません。
- VisiBroker によるイベントサービスは，イベントチャネルおよびコンシューマアプリケーションのいずれにも，データ配信の確認手段を提供しません。コンシューマ，サプライヤ，およびイベントチャネルの間の通信は，TCP/IP を使って実装され，これにより，チャネルおよびコンシューマの両方に信頼できるデータ配信が提供されます。ただし，これは，送信されたデータが実際にすべて受信者によって処理されたことを保証するものではありません。

通信モデル

イベントサービスは，サプライヤとコンシューマに**プル**および**プッシュ**の通信モデルを提供します。**プッシュモデル**では，サプライヤオブジェクトは，データをコンシューマ側に**プッシュ**することでその流れを制御します。**プルモデル**では，コンシューマオブジェクトは，サプライヤからデータを**プル**することでその流れを管理します。

`EventChannel` により，サプライヤとコンシューマは，チャネル上のほかのオブジェクトが使用するモデルを認識する必要がなくなります。つまり，プルサプライヤがプッシュコンシューマにデータを提供したり，プッシュサプライヤがプルコンシューマにデータを提供することができます。

図 17.3　プッシュモデル



メモ　上に示した `EventChannel` は独立したプロセスとして示されていますが，サプライヤオブジェクトのプロセスの一部として実装される場合もあります。

プッシュモデル

プッシュモデルは，2 つの通信モデルのうちでより一般的なモデルです。プッシュモデルの使用例には，ディスクの使用可能な空き容量を監視し，ディスクを使い切ったとき，関係するコンシューマに通知するサプライヤがあります。プッシュサプライヤは，監視しているイベントに応答して，データを ProxyPushConsumer に送信します。

プッシュコンシューマは, ほとんどの時間をイベントループで費やし, ProxyPushSupplier からデータが到着するのを待ちます。EventChannel は，ProxyPushSupplier から ProxyPushConsumer へのデータ転送を促進します。

下の図には，プッシュサプライヤとそれに対応する ProxyPushConsumer オブジェクトが示されています。また，3 つのプッシュコンシューマとそれぞれの ProxyPushSupplier オブジェクトも示されています。

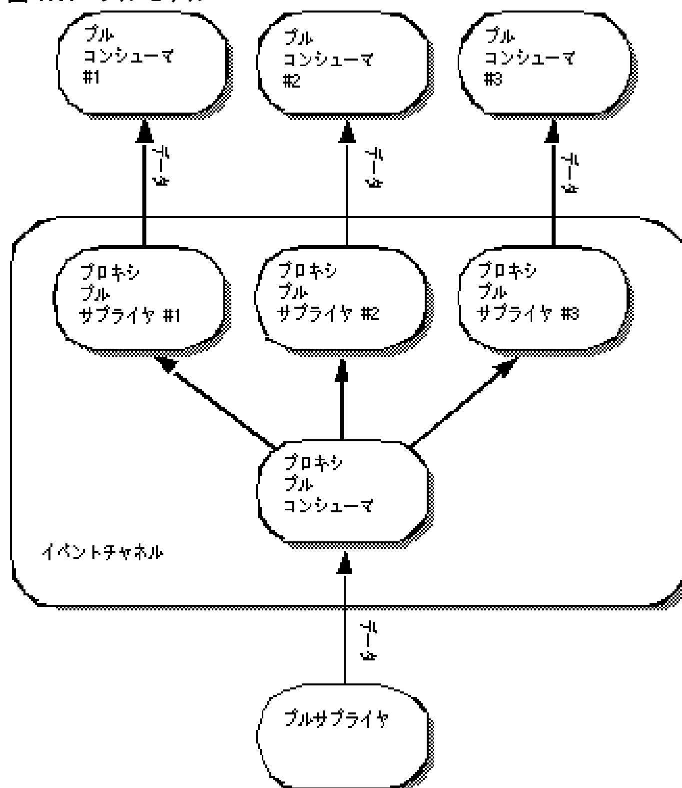
プルモデル

プルモデルでは，イベントチャネルが定期的にサプライヤオブジェクトからデータを引き出し，キューにそのデータを置いて，コンシューマオブジェクトがそのデータをプルできるようにします。プルコンシューマとしては, 1 つまた複数のネットワークモニタがネットワークルーターを定期的にポーリングして統計をとる例があります。

プルサプライヤは，ほとんどの時間をイベントループで費やし，ProxyPullConsumer からデータが到着するのを待ちます。プルコンシューマは，さらにデータを受け取る準備が整うと，ProxyPullSupplier にデータを要求します。EventChannel は，サプライヤからキューへデータを引き出し，ProxyPullSupplier がデータを使用できるようにします。

下の図には，プルサプライヤとそれに対応する ProxyPullConsumer オブジェクトが示されています。また，3 つのプルコンシューマとそれぞれの ProxyPullSupplier オブジェクトも示されています。

図 17.4　プルモデル



メモ　上に示したイベントチャネルは独立したプロセスとして示されていますが，サプライヤオブジェクトのプロセスの一部として実装される場合もあります。

イベントチャネルの使い方

EventChannel を作成し，サプライヤまたはコンシューマと接続して使用するには

1 EventChannel を作成して起動します。

Windows：

```
prompt> start vbj com.inprise.vbroker.CosEvent.EventServer -ior <iorFilename>
            <channelName>
```

UNIX：

```
prompt> vbj com.inprise.vbroker.CosEvent.EventServer -ior <iorFilename>
            <channelName> &
```

メモ　EventChannel のインスタンスは 1 つだけサポートされます。EventChannel へのすべてのバインドは，orb.resolve_initial_references("EventService") への呼び出しを介して行われます。EventService は，ハードコーディングされた EventChannel の名前です。

2 EventChannel に接続します。

3 チャネルから管理オブジェクトを取得し，それを使ってプロキシオブジェクトを取得します。

4 プロキシオブジェクトに接続します。

5 データの転送または受信を開始します。

これらのステップで使用するメソッドは，接続するオブジェクトがサプライヤかコンシューマかによって異なり，さらに使用する通信モデルによっても異なります。下の表は，サプライヤの正しい方法を示します。

表 17.1　サプライヤの EventChannel への接続

手順	プッシュサプライヤ	プルサプライヤ
EventChannel にバインド	CosEventChannelAdmin::EventChannel::_narrow(orb::resolve_initial_references("EventService"))	CosEventChannelAdmin::EventChannel::_narrow(orb::resolve_initial_references("EventService"))
SupplierAdmin を取得	EventChannel::for_suppliers()	EventChannel::for_suppliers()
コンシューマプロキシを取得	SupplierAdmin::obtain_push_consumer()	SupplierAdmin::obtain_pull_consumer()
サプライヤを EventChannel に追加	ProxyPushConsumer::connect_push_supplier()	ProxyPullConsumer::connect_pull_supplier()
データの転送	ProxyPushConsumer::push()	Implements pull() and try_pull()

下の表は，コンシューマの正しい方法を示します。

表 17.2　EventChannel へのコンシューマの接続

手順	プッシュコンシューマ	プルコンシューマ
EventChannel にバインド	CosEventChannelAdmin::EventChannel::_narrow(orb::resolve_initial_references("EventService"))	CosEventChannelAdmin::EventChannel::_narrow(orb::resolve_initial_references("EventService"))
ConsumerAdmin を取得	EventChannel::for_consumers()	EventChannel::for_consumers()
サプライヤプロキシを取得	ConsumerAdmin::obtain_push_supplier()	ConsumerAdmin::obtain_pull_supplier()
コンシューマを EventChannel に追加	ProxyPushSupplier::connect_push_consumer()	ProxyPushSupplier::connect_pull_consumer()
データの転送	Implements push()	ProxyPushSupplier::pull() and try_pull()

イベントチャネルの作成

VisiBroker は，イベントサービスクライアントがオンデマンドでイベントチャネルを作成できるように，CosEventChannelAdmin　モジュールで独自のインターフェース EventChannelFactory を提供しています。この機能を使用するには，次のように，使用するオペレーティングシステムでイベントサービスを開始します

Windows

```
start vbj -Dvbroker.events.factory=true
   com.inprise.vbroker.CosEvent.EventServer <factoryName>
```

UNIX

```
vbj -Dvbroker.events.factory=true
   com.inprise.vbroker.CosEvent.EventServer <factoryName>
```

プロパティ vbroker.events.factory は，チャネルオブジェクトのかわりに <factoryName> という名前（デフォルト値は“VisiEvent”）のファクトリオブジェクトを作成するようにサービスに指示します。ファクトリの IOR をファイルに書き込むには，-ior オプションを使用してファイル名を指定します。デフォルトでは，IOR はコンソールに書き込まれます

作成したファクトリオブジェクトは，ファイル（またはコンソール）に書き込まれた IOR を使用するか，osagent のバインドメカニズムを使用してファクトリオブジェクト名を渡すことで，クライアント側でバインドされます。ファクトリオブジェクトリファレンスが取得されると，これを使用して，イベントチャネルオブジェクトを作成，ルックアップ，または破棄することができます。ファクトリオブジェクトから取得されたイベントチャネルオブジェクトは，サプライヤとコンシューマを接続するために使用できます。

プッシュサプライヤおよびコンシューマのサンプル

この節では，プッシュサプライヤおよびコンシューマアプリケーションのサンプルについて説明します。

プッシュサプライヤ／コンシューマサンプル

この節では，プッシュサプライヤおよびコンシューマアプリケーションのサンプルについて説明します。サプライヤアプリケーションを実行すると，アプリケーションはユーザーにデータの入力を要求し，入力されたデータをコンシューマアプリケーションにプッシュします。コンシューマアプリケーションはデータを受け取り，それを画面に書き出します。

プッシュサプライヤアプリケーションは PushModel.C ファイルに実装され，プッシュコンシューマアプリケーションは PushView.C ファイルに実装されます。これらのファイルは，<install_dir>/examples/vbe/events ディレクトリにあります。

PushSupplier クラスの派生

サプライヤを実装するための最初の手順は，下に示す PushSupplier インターフェースから独自の PushModel クラスを派生させることです。

```
module CosEventComm {
        interface PushSupplier {
           void disconnect_push_supplier();
        };
};
```

下のサンプルコードに，C++ で実装された PushModel クラスを示します。disconnect_push_supplier メソッドが EventChannel によって呼び出され，チャネルが破棄されるとサプライヤは切断されます。このインプリメンテーションでは，単にメッセージを出力して終了します。PushModel オブジェクトが持続する場合，このメソッドも deactivate_obj を呼び出しオブジェクトを停止します。

```
// PushModel.C
#include "CosEventComm_s.hh"
#include "CosEventChannelAdmin_c.hh"
```

```
class PushModel : public POA_CosEventComm::PushSupplier, public VISThread {
public:
   void disconnect_push_supplier() {
      cout << "Model::disconnect_push_supplier()" << endl;
      try {
         PortableServer::ObjectId_var objId =
         PortableServer::string_to_ObjectId("PushModel");
         _myPOA->deactivate_object(objId);
      }
      catch(const CORBA::Exception& e) {
         cout << e << endl;

      }
   }
};
```

PushSupplier の実装

サプライヤインプリメンテーションの最初の部分は，ほぼ決まりきった作業です。何らかの初期化を行った後，ローカルなスコープが設定された結果，ローカルなスコープ付きの PushModel オブジェクトになります。

```
int main(int argc, char* const* argv)
{
   try {
      // ORB を初期化します。
      CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

      // ルート POA へのリファレンスを取得します。
      CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
      //POA の serverPOA を作成します。
      ...

      CPushModel* model = NULL;
      CosEventChannelAdmin::ProxyPushConsumer_var pushConsumer = NULL;

      model = new PushModel(orb, pushConsumer, serverPOA);
      CORBA::String_var supplier_name(CORBA::string_dup("PushModel"));
      PortableServer::ObjectId_var objId =
      PortableServer::string_to_ObjectId(supplier_name);
      serverPOA->activate_object_with_id(objId, model);
      // POA マネージャをアクティブ化します。
      serverPOA->the_POAManager()->activate();
      CORBA::Object_var reference = serverPOA->servant_to_reference(model);
      cout << "Created model: " << reference << endl;
   }
   ...
}
```

このサンプルでは，コマンドラインオプションを使って PushSupplier を実装します。コマンドラインオプションが「m」の場合は，PushModel オブジェクトが初期化およびインスタンス化されます。

コマンドラインオプションが「p」の場合，EventChannel にバインドし，EventChannel から SupplierAdmin オブジェクトを取得します。アプリケーションでは，特定の EventChannel に対してオブジェクト名を指定することもできます。リアルインプリメンテーションでは，引数としてオブジェクトをアプリケーションに渡すことができます。また，ネーミングサービス（VisiNaming）が使用可能な場合は，そのネーミングサービスからオブジェクトを取得できます。詳細については，第 16 章「VisiNaming サービスの使い方」を参照してください。次に SupplierAdmin オブジェクトを使用して，EventChannel から pushConsumer オブジェクトのプロキシを取得します。

コマンドラインオプションが「c」の場合，pushSupplier オブジェクトは EventChannel に接続します。

```
...
if (cmd == 'p') {
   if (channel == NULL) {
      cout << "Need to locate an [e]vent channel" << endl;
   }
   else {
      pushConsumer = channel->for_suppliers()->obtain_push_consumer();
      cout << "Obtained push consumer: " << pushConsumer << endl;
      continue;
   }
}
else if (cmd == 'c' ) {
   if (model == NULL) {
      cout << "Need to create a [m]odel" << endl;
   }
   else if (pushConsumer == NULL) {
      cout << "Need to obtain a [p]ush consumer" << endl;
   }
   else {
      cout << "Connecting..." << endl;
      pushConsumer->connect_push_supplier(model->_this());
      model->start();
      continue;
   }
}
```

サプライヤアプリケーションの別のスレッドは，ユーザーに文字列の入力を求め，文字列の入力を待ち，その文字列を Any 型のオブジェクトに変換します。最後に，そのデータがコンシューマプロキシオブジェクトに「プッシュ」されます。

```
. . .
while(true) {
   VISPortable::vsleep(_delay);
   try {
      char buf[81];
      std::string str;
      sprintf(buf, "%s%d", "Hello #", ++_counter);
      str = buf;

      CORBA::Any_var message = new CORBA::Any();
      *message <<= str.c_str();
      cout << "Supplier pushing: " << str.c_str() << endl;

      _pushConsumer->push(*message);
   }
   catch(CosEventComm::Disconnected e) {
      cout << "Disconnected #" << _counter << endl;
   }
   catch(CORBA::OBJECT_NOT_EXIST e)
   {
      cout << "Push Consumer has been disconnected" << endl;
      return;
   }
   catch(const CORBA::Exception& e) {
      cout << e << endl;
      disconnect_push_supplier();
      return;
   }
   catch(...) {
      cout << "Unexpected exception" << endl;
      disconnect_push_supplier();
```

```
      return;
   }
}
. . .
```

サンプルプッシュサプライヤの完全なインプリメンテーション

```
#include "corba.h"
#include "CosEventComm_s.hh"
#include "CosEventChannelAdmin_c.hh"
#include "vport.h"
#include <string>

USE_STD_NS
class PushModel : public POA_CosEventComm::PushSupplier, public VISThread{
   public:
      PushModel(CORBA::ORB_ptr orb,
               CosEventComm::PushConsumer_ptr pushConsumer,
               PortableServer::POA_ptr myPOA) :
      _orb(orb), _pushConsumer(pushConsumer), _myPOA(myPOA), _counter(0),
                  _delay(1)
      {}
      void delay(int time) { delay = time; }
      void start() {
         // スレッドを起動します。
         run();
      }
      void disconnect_push_supplier() {
         cout << "Model::disconnect_push_supplier()" << endl;
         try {
            PortableServer::ObjectId_var objId =
               PortableServer::string_to_ObjectId("PushModel");
            _myPOA->deactivate_object(objId);
         }
         catch(const CORBA::Exception& e) {
            cout << e << endl;
         }
      }
      // begin() コールバックを実装します。
      void begin() {
         while(true) {
         VISPortable::vsleep(_delay);
         try {
            char buf[81];
            std::string str;
            sprintf(buf, "%s%d", "Hello #", ++_counter);
            str = buf;
            CORBA::Any_var message = new CORBA::Any();
            *message <<= str.c_str();
            cout << "Supplier pushing: " << str.c_str() << endl;
            _pushConsumer->push(*message);
         }
         catch(CosEventComm::Disconnected e) {
            cout << "Disconnected #" << _counter << endl;
         }
         catch(CORBA::OBJECT_NOT_EXIST e)
         {
            cout << "Push Consumer has been disconnected" << endl;
            return;
         }
         catch(const CORBA::Exception& e) {
            cout << e << endl;
```

```
               disconnect_push_supplier();
               return;
            }
            catch(...) {
               cout << "Unexpected exception" << endl;
               disconnect_push_supplier();
               return;
            }
         }
      }

   private :
         int _delay;
         int _counter;
         CORBA::ORB_var _orb;
         PortableServer::POA_var _myPOA;
         CosEventComm::PushConsumer_var _pushConsumer;
   };

   int main(int argc, char* const* argv)
   {
      try {
         // ORB を初期化します。
         CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

         // ルート POA へのリファレンスを取得します。
         CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
         PortableServer::POA_var rootPOA = PortableServer::POA::_narrow(obj);

         // 永続的 POA のポリシーを作成します。
         CORBA::PolicyList policies;
         policies.length(1);
         policies[(CORBA::ULong)0] =
            rootPOA->create_lifespan_policy(PortableServer::PERSISTENT);

         PortableServer::POAManager_var poa_manager = rootPOA->the_POAManager();

         // 適切なポリシーで serverPOA を作成します。
         PortableServer::POA_var serverPOA =
            rootPOA->create_POA("event_service_poa", poa_manager, policies);

         CosEventChannelAdmin::EventChannel_var channel = NULL;
         PushModel* model = NULL;
         CosEventChannelAdmin::ProxyPushConsumer_var pushConsumer = NULL;

         while(true) {
            try {

               cout << "-> ";
               cout.flush();
               char cmd;
               if (cin >> cmd ) {
                  if (cmd == 'e') {
                     obj = orb->resolve_initial_references("EventService");
                     channel = CosEventChannelAdmin::EventChannel::_narrow(obj);
                     cout << "Located event channel: " << channel << endl;
                     continue;
                  }
                  else if (cmd == 'p') {
                     if (channel == NULL) {
                        cout << "Need to locate an [e]vent channel" << endl;
                     }
                     else {
```

```
      pushConsumer = channel->
                        for_suppliers()_>obtain_push_consumer();
      cout << "Obtained push consumer: " << pushConsumer << endl;
      continue;
   }
}
else if (cmd == 'm') {
   if (pushConsumer == NULL) {
      cout << "Need to obtain a [p]ush consumer" << endl;
   }
   else {
      model = new PushModel(orb, pushConsumer, serverPOA);
      CORBA::String_var
                     supplier_name(CORBA::string_dup("PushModel"));
      PortableServer::ObjectId_var objId =
         PortableServer::string_to_ObjectId(supplier_name);
      serverPOA->activate_object_with_id(objId, model);
      // POA マネージャをアクティブ化します。
      serverPOA->the_POAManager()->activate();
      CORBA::Object_var reference = serverPOA->
                        servant_to_reference(model);
      cout << "Created model: " << reference << endl;
      continue;
   }
}
else if (cmd == 's') {
   if (model == NULL) {
      cout << "Need to create a [m]odel" << endl;
   }
   else {
      int delay;
      if (cin >> delay ) {
         if (delay < 0)
            cout << "[s]leep delay must be positive" ;
         else
            model->delay(delay);
      }
      else {
         cerr << "Invalid argument to [s]leep" << endl;
      }
   }
}
else if (cmd == 'c' ) {
   if (model == NULL) {
      cout << "Need to create a [m]odel" << endl;
   }
   else if (pushConsumer == NULL) {
      cout << "Need to obtain a [p]ush consumer" << endl;
   }
   else {
      cout << "Connecting..." << endl;
      pushConsumer->connect_push_supplier(model->_this());
      model->start();
      continue;
   }
}
else if (cmd == 'd') {
   if (pushConsumer == NULL) {
      cout << "Need to obtain a [p]ush consumer" << endl;
   }
else {
   cout << "Disconnecting..." << endl;
   pushConsumer->disconnect_push_consumer();
```

```
                    continue;
                 }
              }
              else if (cmd == 'q') {
                 cout << "Quitting..." << endl;
                 CORBA::ORB::shutdown();
                 break;
              }
              else {
   cout << "Commands: e [e]vent channel" << endl
                    << "          s <# seconds> set [s]leep delay" << endl
                    << "          p               [p]ush consumer"    << endl
                    << "          m               [m]odel"            << endl
                    << "          c               [c]onnect"          << endl
                    << "          d               [d]isconnect"       << endl
                    << "          q               [q]uit"             << endl;
              }
           }
        }
        catch(const CORBA::SystemException& e) {
           cerr << e << endl;
        }
     }
     }
     catch(const CORBA::Exception& e) {
        cerr << e << endl;
     }
     return 0;
  }
```

PushConsumer クラスの派生

次のサンプルコードは，サプライヤアプリケーションの最初の部分を示したもので，下に示す PushConsumer インターフェースから派生した PushView クラスを定義します。

```
module CosEventComm {
         exception Disconnected();
         interface PushConsumer {
            void push(in any data) raises(Disconnected);
            void disconnect_push_consumer();
         };
};
```

push メソッドは Any 型を受け取り，それを文字列に変換して出力を試みます。disconnect_push_supplier メソッドが EventChannel によって呼び出され，チャネルが破棄されるとサプライヤは切断されます。

```
// PushView.C
#include "CosEventComm_s.hh"
#include "CosEventChannelAdmin_c.hh"
class PushView : public POA_CosEventComm::PushConsumer
{
   public:
      void push(const CORBA::Any& data) {
         cout << "Consumer being pushed: " << data << endl;
      }

      void disconnect_push_consumer() {
         cout << "PushView::disconnect_push_consumer" << endl;
      }
};
```

PushConsumer の実装

コマンドラインオプションが「v」の場合，PushConsumer オブジェクトがインスタンス化およびアクティブ化されます。その他のコマンドラインオプションの場合，PushConsumer オブジェクトは EventChannel にバインドし，サプライヤプロキシオブジェクトを取得してコンシューマオブジェクトに接続し，プッシュ要求の受信を待ちます。

```
// PushView.C
#include "CosEventComm_s.hh"
#include "CosEventChannelAdmin_c.hh"
. . .
int main(int argc, char* const* argv)
{
   try {
      // ORB を初期化します。
      CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

      // ルート POA へのリファレンスを取得します。
      CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
      PortableServer::POA_var rootPOA = PortableServer::POA::_narrow(obj);

      // 永続的 POA のポリシーを作成します。
      CORBA::PolicyList policies;
      policies.length(1);
      policies[(CORBA::ULong)0] =
      rootPOA->create_lifespan_policy(PortableServer::PERSISTENT);

      PortableServer::POAManager_var poa_manager = rootPOA->the_POAManager();
      // 適切なポリシーで serverPOA を作成します。
      PortableServer::POA_var serverPOA =
         rootPOA->create_POA("event_service_poa", poa_manager, policies);

      CosEventChannelAdmin::EventChannel_var channel = NULL;
      PushView* view = NULL;
      CosEventChannelAdmin::ProxyPushSupplier_var pushSupplier = NULL;

      while(true) {
         try {
            cout << "-> ";
            cout.flush();
            char cmd;
            if (cin >> cmd) {
               if (cmd == 'e') {
                  obj = orb->resolve_initial_references("EventService");
                  channel = CosEventChannelAdmin::EventChannel::_narrow(obj);
                  cout << "Located event channel: " << channel << endl;
                  continue;
               }
               else if (cmd == 'v') {
                  view = new PushView();
                  CORBA::String_var consumer_name(CORBA::string_dup("PushView"));
                  PortableServer::ObjectId_var objId =
                  PortableServer::string_to_ObjectId(consumer_name);
                  serverPOA->activate_object_with_id(objId, view);
                  // POA マネージャをアクティブ化します。
                  serverPOA->the_POAManager()->activate();
                  CORBA::Object_var reference = serverPOA
                                                    ->servant_to_reference(view);
                  cout << "Created view: " << reference << endl;
                  continue;
               }
               else if (cmd == 'p') {
                  if (channel == NULL) {
```

```
                    cout << "Need to locate an [e]vent channel" << endl;
                 }
                 else {
                    pushSupplier = channel->for_consumers()
                                                  ->obtain_push_supplier();
                    cout << "Obtained push consumer: " << pushSupplier << endl;
                    continue;
                 }
              }
              else if (cmd == 'c' ) {
                 if (view == NULL) {
                    cout << "Need to create a [v]iew" << endl;
                 }
                 else if (pushSupplier == NULL) {
                    cout << "Need to obtain a [p]ush supplier" << endl;
                 }
                 else {
                    cout << "Connecting..." << endl;
                    pushSupplier->connect_push_consumer(view->_this());
                    continue;
                 }
              }
              else if (cmd == 'd') {
                 if (pushSupplier == NULL) {
                    cout << "Need to obtain a [p]ush supplier" << endl;
                 }
                 else {
                    cout << "Disconnecting..." << endl;
                    pushSupplier->disconnect_push_supplier();
                    continue;
                 }
              }
              else if (cmd == 'q') {
                 cout << "Quitting..." << endl;
                 break;
              }
               cout << "Commands: e            [e]vent channel"    << endl
                    << "          p            [p]ush supplier"    << endl
                    << "          v            [v]iew"             << endl
                    << "          c            [c]onnect"          << endl
                    << "          d            [d]isconnect"       << endl
                    << "          q            [q]uit"            << endl;
           }
        }
        catch(const CORBA::SystemException& e) {
           cerr << e << endl;
        }
     }
  }
  catch(const CORBA::Exception& e) {
     cerr << e << endl;
  }
}
```

キューの長さの設定

環境によっては，コンシューマアプリケーションがサプライヤアプリケーションより動作が遅い場合があります。サプライヤからのメッセージの速度に追いつくことができないコンシューマには，未処理のメッセージがたまります。maxQueueLength パラメータを使用してこのメッセージの数を制限すると，メモリ不足の状態を回避できます。

サプライヤが毎秒 10 のメッセージを生成し，コンシューマが毎秒 1 つのメッセージだけを処理できる場合，キューはすぐにいっぱいになります。キュー内のメッセージは一定の最大長を持ち，いっぱいのキューにメッセージを追加しようとすると，チャネルはキュー内の最も古いメッセージを除去して，新しいメッセージのための領域を確保します。

各コンシューマは別のキューを持っているので，早いコンシューマは見逃しませんが，遅いコンシューマはメッセージを見逃すこともあります。下のサンプルコードは，各コンシューマの未処理メッセージ数を 15 に限定する方法を示しています。

```
CosEventChannel -maxQueueLength=15 MyChannel
```

メモ maxQueueLength を指定しないか，不正な数を指定した場合は，デフォルトのキューの長さ 100 が使用されます。

プログラムのコンパイルとリンク

イベントサービスを使用するアプリケーションでは，次の生成ファイルをインクルードする必要があります。

```
#include "CosEventComm_s.hh"
#include "CosEventChannelAdmin_c.hh"
```

UNIX： UNIX アプリケーションは，次のどちらかのライブラリとリンクする必要があります。

- libcosev.a
- libcosev.so

Windows： Windows アプリケーションは，cosev_r.lib（cosev_r.dll）ライブラリとリンクする必要があります。

第 18 章

VisiBroker サーバーマネージャの使用

VisiBroker サーバーマネージャを使用すると，クライアントアプリケーションがオブジェクトサーバーを監視および管理したり，オブジェクトサーバーのプロパティを実行時に参照して設定したり，サーバーマネージャオブジェクトのメソッドを参照して呼び出すことができます。サーバーマネージャは，コンテナと呼ばれる要素を使用して，主要な ORB コンポーネントを表します。コンテナは，プロパティやオペレーションのほか，ほかのコンテナも保持できます。

メモ　サーバーマネージャのコンテナと J2EE のコンテナとを混同しないでください。サーバーマネージャのコンテナは，ORB コンポーネントと選択された実行時プロパティを論理的にグループ化したものです。

サーバーマネージャの概要

ここでは，サーバーでのサーバーマネージャの有効化，サーバーマネージャリファレンスの取得，コンテナの使用，Storage インターフェース，およびサーバーマネージャの IDL について説明します。

サーバーでのサーバーマネージャの有効化

デフォルトでは，VisiBroker サーバーの管理は有効になっていません。VisiBroker サーバーを管理するには，サーバーを起動するコマンドで次のプロパティを設定する必要があります。

```
vbroker.orb.enableServerManager=true
```

このプロパティは，コマンドラインまたはサーバーのプロパティファイルのいずれかを介して指定できます。

サーバーマネージャリファレンスの取得

サーバーマネージャと対話するには，最初にサーバーのサーバーマネージャへのリファレンスを取得します。このリファレンスは，最上位コンテナをポイントします。クライアントは次の 2 つの方法でリファレンスを取得できます。

1. サーバーランナーは，プロパティオプション vbroker.serverManager.name を使用して，サーバーマネージャを指定できます。たとえば，次のコマンドを使用します。

   ```
   prompt> Server -Dvbroker.serverManager.name=BigBadBoss
   ```

   これは，サーバーマネージャの名前 "BigBadBoss" をスマートエージェントの名前空間に登録します。これ以降，クライアントは，この名前にバインドし，リファレンスのオペレーション呼び出しを開始できます。このプロパティは，プロパティファイルでも設定できます。この方法でサーバーマネージャを検索できるのは，このサーバーによって実装されるほかのオブジェクトのオブジェクトリファレンスをクライアントが持たない場合です。次に例を示します。

   ```
   ServerManager::Container_var cont = ServerManager::Container::_bind("BigBadBoss");
   ```

2. サーバーによって実装されるほかのオブジェクトのオブジェクトリファレンスをクライアントが持つ場合，クライアントは，そのオブジェクトの _resolve_reference("ServerManager") を実行することで，そのオブジェクトに対応する ORB の ServerManager を取得できます。次のコードでは，サーバーマネージャの最上位コンテナを Bank::AccountManager オブジェクトから取得します。

   ```
   Bank::AccountManager_var manager = Bank::AccountManager::_bind("/bank_agent_poa",
   managerId);
   ServerManager::Container_var cont;
   CORBA::Object_var objCont = manager->_resolve_reference("ServerManager");
   ```

   サーバーマネージャのインターフェースを使用するために，クライアントコードは，servermgr_c.hh をインクルードする必要があります。

コンテナの使用

クライアントアプリケーションは，最上位コンテナへのリファレンスを取得すると，次の操作を実行できます。

- 最上位コンテナのプロパティを取得，設定，および追加する。
- 最上位コンテナ内のコンテナを反復処理する。
- コンテナを取得，設定，または追加する。
- コンテナで定義されたオペレーションを呼び出す。
- コンテナのストレージを取得または設定する。
- プロパティをプロパティストレージから復元またはプロパティストレージに永続化する。

最上位コンテナにプロパティやオペレーションはなく，ORB コンテナを保持するだけです。一方，その ORB コンテナには，ORB プロパティや shutdown メソッドのほか，RootPOA，Agent，OAD などのコンテナが含まれます。

コンテナとの対話の方法については，235 ページの「Container インターフェース」を参照してください。239 ページの「サーバーマネージャの例」節では，Java や C++ での対話を示します。

Storage インターフェース

サーバーマネージャには，任意の形式で実装できるストレージという抽象概念があります。各コンテナは，各自のプロパティをそれぞれの方法で格納するように選択できます。プロ

パティをデータベースに格納する場合もあれば，ファイルなどの別の方法で格納する場合もあります。Storage インターフェースは，サーバーマネージャの IDL で定義されます。

各コンテナは，同じメソッドを使ってストレージを取得および設定するだけでなく，オプションで親のすべての子コンテナにストレージを設定できます。同様に，各コンテナは同じメソッドを使用して，ストレージとの間でプロパティを読み書きします。

Storage インターフェースとそのメソッドについては，「Storage インターフェース」を参照してください。

Container インターフェース

Container インターフェースは，オブジェクト，プロパティ，オペレーションなどを論理的にグループ化するためのインターフェースと関連メソッドを定義します。

Container のメソッド

コンテナは，プロパティ，オペレーション，およびほかのコンテナを保持できます。主要な ORB コンポーネントは，コンテナとして表されます。最上位のコンテナは ORB 自体に対応し，いくつかの ORB プロパティ，shutdown メソッド，およびよく使用されるほかのコンテナ（RootPOA や Agent など）を含んでいます。

ここでは，Container インターフェースで実行できる C++ のメソッドについて説明します。これらのメソッドは次の 4 つのカテゴリに分けられます。

- プロパティの操作とクエリーに関連するメソッド
- オペレーションに関連するメソッド
- 子コンテナに関連するメソッド
- ストレージに関連するメソッド

プロパティの操作とクエリーに関連するメソッド

```
CORBA::StringSequence list_all_properties();
```

コンテナ内にあるすべてのプロパティの名前を StringSequence として返します。

```
PropertySequence get_all_properties();
```

コンテナ内にあるすべてのプロパティの名前，値，読み取り／書き込みの状態を含む PropertySequence を返します。

```
Property get_property(in string name raises(NameInvalid);
```

入力パラメータとして渡されたプロパティ **name** の値を返します。

```
void add_property(in string name, in any value) raises(NameInvalid, ValueInvalid,
ValueNotSettable);
```

プロパティ **name** の値を要求された **value** に設定します。

```
void persist_properties(in boolean recurse) raises(StorageException);
```

コンテナは，関連付けられている 237 ページの「Storage インターフェースのメソッド」にプロパティを実際に保存します。ストレージがコンテナに関連付けられていない場合は，StorageException が生成されます。パラメータ recurse=true を指定して呼び出すと，子コンテナのプロパティもストレージに保存されます。すべてのプロパティを保存するか，変更されたプロパティだけを保存するかは，コンテナによって異なります。

```
void restore_properties(in boolean recurse) raises(StorageException);
```

ストレージからプロパティを取得するようにコンテナに指示します。コンテナは，管理しているプロパティを正確に認識しており，それらをストレージから読み取ろうとします。

ORB に付属するコンテナは，ストレージからの復元をサポートしていません。この機能をサポートするコンテナは，独自に作成する必要があります。

オペレーションに関連するメソッド

```
::CORBA::StringSeqence list_all_operations();
```

コンテナでサポートされているすべてのオペレーションの名前を返します。

```
OperationSequence get_all_operations();
```

すべてのオペレーション，オペレーションのパラメータ，およびパラメータのタイプコードを返します。これで，適切なパラメータを使ってオペレーションを呼び出すことができます。

```
Operation get_operation(in string name) raises(NameInvalid);
```

name で指定されたオペレーションのパラメータ情報を返します。この情報を使用して，オペレーションを呼び出すことができます。

```
any do_operation(in Operation op) raises(NameInvalid, ValueInvalid, OperationFailed);
```

オペレーションのメソッドを呼び出し，その結果を返します。

子コンテナに関連するメソッド

```
::CORBA::StringSequence list_all_containers();
```

現在のコンテナの子コンテナの名前をすべて返します。

```
NamedContainerSequence get_all_containers();
```

すべての子コンテナを返します。

```
NamedContainer get_container(in string name) raises(NameInvalid);
```

name パラメータで指定された子コンテナを返します。この名前の子コンテナがない場合は，`NameInvalid` 例外が生成されます。

```
void add_container(in NamedContainer container) raises(NameAlreadyPresent,
ValueInvalid);
```

このコンテナの子コンテナとして，**container** を追加します。

```
void set_container(in string name, in Container value)  raises(NameInvalid,
ValueInvalid, ValueNotSettable);
```

name パラメータで指定された子コンテナを **value** パラメータで指定されたコンテナに変更します。

ストレージに関連するメソッド

```
void set_storage(in Storage s, in boolean recurse);
```

このコンテナのストレージを設定します。`recurse=true` の場合は，すべての子コンテナに対してもストレージが設定されます。

```
Storage get_storage();
```

コンテナの現在のストレージを返します。

Storage インターフェース

サーバーマネージャには，任意の形式で実装できる*ストレージ*という抽象概念があります。各コンテナは，プロパティの保存先をデータベースやフラットファイルなどの形式から選択できます。VisiBroker ORB に用意されているストレージのインプリメンテーションでは，フラットファイルが使用されます。

Storage インターフェースのメソッド

```
void open() raises (StorageException);
```

ストレージを開き，プロパティを読み書きできるようにします。データベースに基づくインプリメンテーションの場合は，このメソッドによってデータベースにログインします。

```
void close() raises (StorageException);
```

ストレージを閉じます。また，このメソッドは，最後に Container::persist_properties が呼び出されてから変更されたプロパティがあれば，ストレージを更新します。データベースに基づくインプリメンテーションでは，このメソッドによってデータベース接続が閉じられます。

```
Container::PropertySequence read_properties() raises(StorageException);
```

ストレージからすべてのプロパティを読み取ります。

```
Container::Property read_property(in string propertyName) raises(StorageException,
Container::NameInvalid);
```

ストレージから読み取った **propertyName** のプロパティ値を返します。

```
void write_properties(in Container::PropertySequence p) raises(StorageException);
```

ストレージにプロパティシーケンスを保存します。

```
void write_property(in Container::Property p) raises(StorageException);
```

ストレージに 1 つのプロパティを保存します。

サーバーマネージャに対するアクセスの制限

サーバーマネージャを取得するクライアントは ORB 全体を制御できるため，セキュリティが重要になります。次のプロパティは，ユーザーによるサーバーマネージャ機能へのアクセスを制限します。

プロパティ	デフォルト値	説明
vbroker.orb.enableServerManager	false	このプロパティを True に設定すると，サーバーマネージャが有効になります。
vbroker.serverManager.enableOperations	true	コンテナのオペレーションを呼び出すための権限を制御します。false に設定されると，クライアントは，コンテナの do_operation を呼び出すことができなくなります。
vbroker.serverManager.enableSetProperty	true	クライアントからのプロパティの設定を制御します。false に設定されると，クライアントは，コンテナのプロパティを変更できなくなります。

サーバーマネージャ IDL

サーバーマネージャの IDL は次のとおりです。

```
module ServerManager {
interface Storage;

exception StorageException {
    string reason;
};

interface Container
{
    enum RWStatus {
        READWRITE_ALL,
```

```
        READONLY_IN_SESSION,
        READONLY_ALL
    };

    struct Property {
        string name;
        any value;
        RWStatus rw_status;
    };
    typedef sequence<Property> PropertySequence;

    struct NamedContainer {
        string name;
        Container value;
        boolean is_replaceable;
    };
    typedef sequence<NamedContainer> NamedContainerSequence;

    struct Parameter {
        string name;
        any value;
    };
    typedef sequence<Parameter> ParameterSequence;

    struct Operation {
        string name;
        ParameterSequence params;
        ::CORBA::TypeCode result;
    };
    typedef sequence<Operation> OperationSequence;

    struct VersionInfo {
        unsigned long major;
        unsigned long minor;
    };

    exception NameInvalid{};
    exception NameAlreadyPresent{};
    exception ValueInvalid{};
    exception ValueNotSettable{};
    exception OperationFailed{
        string real_exception_reason;
    };

    ::CORBA::StringSequence list_all_properties();
    PropertySequence get_all_properties();
    Property get_property(in string name) raises (NameInvalid);
    void add_property(in Property prop)
    raises(NameAlreadyPresent, NameInvalid, ValueInvalid);
    void set_property(in string name, in any value)
    raises(NameInvalid, ValueInvalid, ValueNotSettable);

    ::CORBA::StringSequence get_value_chain(in string propertyName) raises
(NameInvalid);
    void persist_properties(in boolean recurse) raises (StorageException);
    void restore_properties(in boolean recurse) raises (StorageException);

    ::CORBA::StringSequence list_all_operations();
    OperationSequence get_all_operations();
    Operation get_operation(in string name)
    raises (NameInvalid);
    any do_operation(in Operation op)
    raises(NameInvalid, ValueInvalid, OperationFailed);

    ::CORBA::StringSequence list_all_containers();
    NamedContainerSequence get_all_containers();
```

```
    NamedContainer get_container(in string name)
    raises (NameInvalid);
    void add_container(in NamedContainer container)
    raises(NameAlreadyPresent, ValueInvalid);
    void set_container(in string name, in Container value)
    raises(NameInvalid, ValueInvalid, ValueNotSettable);


    void set_storage(in Storage s, in boolean recurse);
    Storage get_storage();

    readonly attribute VersionInfo version;
};

interface Storage
{
    void open() raises (StorageException);
    void close() raises (StorageException);
    Container::PropertySequence read_properties() raises
    (StorageException);
    Container::Property read_property(in string propertyName)
    raises (StorageException, Container::NameInvalid);
    void write_properties(in Container::PropertySequence p) raises
    (StorageException);
    void write_property(in Container::Property p) raises (StorageException);
};

};
```

サーバーマネージャの例

以下では，次の処理を行う例を示します。

1. 最上位コンテナへのリファレンスを取得する。
2. すべてのコンテナとそれらのプロパティを再帰的に取得する。
3. 複数のコンテナのプロパティを取得，設定，および保存する。
4. ORB コンテナの shutdown() メソッドを呼び出す。

これらのサンプルファイルは，次のディレクトリにあります。

```
<install_dir>/examples/vbe/ServerManager/
```

次の例では，bank_agent サーバーを使用します。このサーバーを起動するには，プロパティストレージファイルを渡す必要があります。プロパティファイルには，初期値として，サーバーマネージャを有効にしたり，サーバーマネージャの名前を設定するためのプロパティが含まれています。ユーザーがプロパティを変更すると，サーバーマネージャによってプロパティファイル内のプロパティが更新されます。サーバーマネージャを有効にしたり，サーバーマネージャの名前を設定するためのプロパティは，コマンドラインオプションとして渡すことができます。ただし，セッション中にいずれかのプロパティを変更して保存する場合には，プロパティファイルが必要になります。

プロパティファイルには，初期値として，次のプロパティが含まれています。

```
# サーバーのプロパティ
vbroker.orb.enableServerManager=true
vbroker.serverManager.name=BigBadBoss
```

サーバーは，コマンドラインから次のように起動されます。

```
prompt> Server -ORBpropStorage prop.txt
```

最上位コンテナへのリファレンスの取得

サーバーマネージャが名前付きで起動されているため，この例では，2 番めの bind メソッドを使用します（234 ページの「サーバーマネージャリファレンスの取得」を参照）。

```
ServerManager::Container_var cont = ServerManager::Container::_bind("BigBadBoss");
```

すべてのコンテナとそれらのプロパティの取得

次の例は，get_all_properties，get_all_operations，および get_all_containers を使用して，現在のコンテナの下にあるすべてのコンテナにすべてのプロパティとオペレーションを再帰的に照会する方法を示します。

```
void SrvrmgrUtil::displayContainer(char * name, ServerManager::Container_ptr cont) {
  try {
        ServerManager::Container::PropertySequence * props = cont->get_all_properties();
        for (int i =0; i < props->length() ; i++) {
                printProperty((*props)[i]);
        }
        ServerManager::Container::OperationSequence * ops = cont->get_all_operations();
        for (int j = 0; j < ops->length(); j++)
                printOperation((*ops)[j]);
        }
        catch (ServerManager::Container::NameInvalid& ne) {
        cerr << ne <<endl;
        } catch (ServerManager::Container::ValueInvalid & ve) {
                cerr << ve <<endl;
        } // 残りの例外を main 関数に渡します。
        ServerManager::Container::NamedContainerSequence* nc = cont-
>get_all_containers();
        for (int j =0 ; j < nc->length(); j++) {
                displayContainer((*nc)[j].name, (*nc)[j].value);
        }
}
```

プロパティの取得と設定，およびファイルへの保存

次のコードは，コンテナにプロパティを照会する方法を示します。コンテナが最上位コンテナでない場合は，上位コンテナからすべての親をたどって最上位コンテナに到達する必要があります。get メソッドと set メソッドは，そのプロパティを所有するコンテナでのみ呼び出すことができます。

メモ READONLY_ALL という RW_STATUS 値を持つプロパティは，設定できません。

```
// プロパティを照会します。
ServerManager::Container::NamedContainer_var orbCont = cont->get_container("ORB");
ServerManager::Container::NamedContainer_var sesCont =
                orbCont->value->get_container("ServerEngines");
ServerManager::Container::NamedContainer_var seCont =
                sesCont->value->get_container("iiop_tp");
ServerManager::Container::NamedContainer_var scmCont =
                seCont->value->get_container("iiop_tp");
SrvrmgrUtil::displayProperty("vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.inUseThreads",
                scmCont->value);

CORBA::Any_var a = new CORBA::Any;
a <<= (CORBA::ULong) 34001UL;
scmCont->value->set_property("vbroker.se.iiop_tp.scm.iiop_tp.listener.port", a);
scmCont->value->persist_properties(true);
```

Container でのオペレーションの呼び出し

ORB コンテナはオペレーション shutdown をサポートします。このオペレーションを取得するには，コンテナの get_operation を呼び出します。

```
void SrvrmgrUtil::invokeShutdown(ServerManager::Container_ptr orbCont)
{
        ServerManager::Container::Operation_var shutOp = orbCont-
>get_operation("shutdown");
        shutOp->params[0].value <<= CORBA::Any::from_boolean(0UL);
        orbCont->do_operation(shutOp.in());
}
```

get_operation 呼び出しから返される operation には，デフォルトのパラメータがあります。パラメータのデフォルト値が適切でない場合は，これらの値を変更してから do_operation メソッドを呼び出す必要があります。

カスタムコンテナ

ユーザーアプリケーションは，コンテナを定義してサーバーマネージャに追加することができます。コンテナは 2 つのプロパティを管理し，1 つのオペレーションを定義します。さらに，独自のストレージを使ってプロパティを格納することもできます。次の 2 つのプロパティがあります。

プロパティ	説明
manager.lockAllAccounts	このプロパティは，READWRITE_ALL という読み取り／書き込み状態を持ち，サーバーの実行中に変更して適用できます。このプロパティの目的は，クライアントアプリケーションが AccountManager を使用できないようにすることです。このプロパティの初期値は，起動時にサーバーによって読み取られ，サーバーのシャットダウン／再起動時に同じファイルに保存されます。
manager.numAccounts	このプロパティは，READONLY_ALL という読み取り／書き込み状態を持ち，読み取り専用です。このプロパティの目的は，AccountManager で Account の数を提供することです。このプロパティの値はストレージに書き込まれません。

オペレーションは次のとおりです。

オペレーション	説明
shutdown	サーバーをシャットダウンし，再起動は行いません。シャットダウンする前に，manager.lockAllAccounts プロパティがプロパティファイルに書き込まれ（永続化され）ます。

完全なサンプルについては，次のディレクトリを参照してください。

<install_dir>/examples/vbe/ServerManager/custom_container/]"-->

サーバーマネージャ : カスタムコンテナの書き込み "-->

カスタムコンテナの書き込み手順は次のとおりです。

1. サーバーマネージャの IDL で定義された Container インターフェースを実装します。
2. Container インターフェースを実装するサーバントをインスタンス化し，POA でアクティブ化します。
3. サーバーマネージャの最上位コンテナへのリファレンスを取得します。カスタムコンテナをコンテナ階層に追加します。

サーバーマネージャを有効にしてサーバーを起動すると，クライアントはカスタムコンテナと対話できるようになります。

アプリケーションが独自のストレージを実装するには，サーバーマネージャの IDL で定義された `Storage` インターフェースを実装する必要があります。基本的な手順は，カスタムコンテナの実装手順と同じです。

第 19 章

VisiBroker ネイティブメッセージングの使用

はじめに

ネイティブメッセージングは，CORBA および RMI/J2EE（RMI-over-IIOP）アプリケーション用の OMG 準拠 2 フェーズ呼び出しフレームワークで，言語に依存しておらず，可搬性があり，相互運用が可能で，かつサーバーに対して透過的です。

2 フェーズ呼び出し（2PI）

オブジェクト指向の用語では，呼び出しとは，ターゲットオブジェクト上で行われるメソッドの呼び出しをいいます。概念的には，呼び出しは次の 2 フェーズの通信で構成されます。

- 第 1 フェーズのターゲットへの要求の送信
- 第 2 フェーズのターゲットからの応答の受信

CORBA，RMI/J2EE，.NET など従来のオブジェクト指向分散フレームワークでは，オブジェクト上の呼び出しは **1 フェーズ呼び出し**（1PI）であり，送信フェーズと受信フェーズは個々に公開されるのではなく，単一の操作内に一緒にカプセル化されます。1 フェーズ呼び出しでは，クライアントの呼び出し元スレッドは，第 1 フェーズ終了後，第 2 フェーズが完了または中断するまで操作上でブロックします。

第 1 フェーズ終了後にクライアントをアンブロックでき，かつ第 2 フェーズを別個に実行できる場合，そのような呼び出しを **2 フェーズ呼び出し**（2PI：Two-phase invocation）といいます。また，2 つの呼び出し段階が完了する前にアンブロックする操作を，ネイティブメッセージングでは**早期リターン**（PR：premature return）といいます。

2PI では，クライアントアプリケーションは，要求送信フェーズが修了後，ただちにアンブロックできます。したがって，クライアントは，応答を待機する間，呼び出し元スレッドを停止して転送接続を保持する必要がありません。クライアントは，別のクライアント実行コンテキストから，あるいは別の転送接続を通して，応答を取得または受信できます。

ポーリング／プルモデルとコールバックモデル

2 フェーズ呼び出しでは，クライアントアプリケーションは，各要求を送信後，次のいずれかを実行できます。つまり，インフラストラクチャによって提供されるポーリングオブジェクトを使って応答を積極的にポーリングおよびプルするか，あるいはインフラストラクチャが要求を通知し，指定された非同期コールバックハンドラ上で応答を返信するのを消極的に待機することができます。通常，この 2 つの動作は，それぞれ同期**ポーリング／プル**モデルおよび非同期**コールバック**モデルと呼ばれます。

非ネイティブメッセージングと IDL の変形

CORBA Messaging などネイティブでないメッセージングでは，ネイティブの IDL または RMI インターフェース上でネイティブのオペレーションシグニチャを使って 2 フェーズ呼び出しを実行することはできません。かわりに，クライアントアプリケーションは，さまざまな呼び出し段階で，さまざまな応答取得モデルを使用して，さまざまに変形されたオペレーションを呼び出す必要があります。

たとえば CORBA Messaging では，ターゲット上で `foo（<parameter_list>）`オペレーションの 2 フェーズ呼び出しを実行するには，ネイティブのシグニチャ `foo()` ではなく，次の変形シグニチャのいずれかを使って要求送信を実行します。

```
// ポーリング／プルモデルの場合
sendp_foo(<input_parameter_list>);

// コールバックモデルの場合
sendc_foo(<callback_handler>, <input_parameter_list>);
```

応答ポーリングオペレーションのシグニチャは次のようになります。

```
foo(<timeout>, <return_and_output_parameter_list_as_output>);
```

応答配信コールバックオペレーションのシグニチャは次のようになります。

```
foo(<return_and_output_parameter_list_reversed_as_input>);
```

これらの変形オペレーションは，アプリケーションで指定された元のインターフェースに追加された追加シグニチャであるか，または追加の型固有のインターフェースまたは valuetype で定義された追加シグニチャのいずれかです。

非ネイティブメッセージングで IDL を変更する場合，次のような問題点があります。

- 元の IDL インターフェースとオペレーションシグニチャの直観的なわかりやすさが損なわれます。
- Java RMI の場合など，ほかのオペレーション変形と競合する可能性があります。
- 元の IDL インターフェースによってすでに使用されているオペレーションシグニチャと矛盾する可能性があります。
- インターフェースのバイナリ互換性が失われます。たとえば，シグニチャが変形されている場合，変形されていない場合のいずれでも，インターフェースは，言語マッピングでバイナリ互換であるとは限りません。
- IDL オペレーションとネイティブの GIOP メッセージとの間の自然なマッピングが重視されないため，ポータブルインターセプタのようなほかの OMG CORBA 機能と併用すると矛盾が発生したりジレンマに陥ります。

ネイティブメッセージングソリューション

ネイティブメッセージングは，アプリケーションによって定義されている**ネイティブな** IDL 言語マッピングと**ネイティブな** RMI インターフェースだけを，インターフェースをまったく変形せずに，またアプリケーション固有のインターフェースまたは valuetype を追加せずに使用します。

たとえば，ネイティブメッセージングでは，要求の foo(<parameter_list>) への送信や，ポーリング／プルモデルまたはコールバックモデルでの応答の取得（または受信）は，完全にネイティブな foo(<parameter_list>) オペレーションを使用して，ネイティブの IDL または RMI インターフェース上で実行されます。変形されたオペレーションシグニチャやインターフェース，または valuetype が導入されたり使用されることはありません。

この完全にネイティブで変形を伴わない手法は，明快でわかりやすいだけでなく，オペレーションシグニチャ変形による競合や名前の競合，矛盾などを完全に排除できます。

リクエストエージェント

OMG のセキュリティサービスやトランザクションサービスと同様に，ネイティブメッセージングは，オブジェクトサービスレベルのソリューションであり，完全に相互運用可能なブローカーサーバーであるリクエストエージェントと，OMG ポータブルインターセプタ仕様に完全に準拠しているクライアント側の可搬性のあるリクエストインターセプタに基づいています。

2 フェーズ呼び出しを実行する際，ネイティブメッセージングアプリケーションは，要求をターゲットオブジェクトへ直接送信しません。かわりに，指定されたリクエストエージェント上に作成されたデリゲート要求プロキシ上で，要求呼び出しが実行されます。要求プロキシは，指定されたターゲットオブジェクトに呼び出しをデリゲートし，クライアントのコールバックハンドラに応答を配信するか，またはクライアントのポーリング／プル時に応答を返す役割を果たします。

したがって，リクエストエージェントをクライアントアプリケーションが認識する必要があります。それには，通常，OMG 準拠の ORB 初期化コマンド引数を使ってクライアント ORB を初期化します。

```
-ORBInitRef  RequestAgent=<request_agent_ior_or_url>
```

このコマンドにより，クライアントアプリケーションは，この ORB からのリクエストエージェントリファレンスを初期サービスとして解決します。次に例を示します。

```
// C++ でのリクエストエージェントリファレンス取得
CORBA::Object_var ref
    = orb->resolve_initial_references("RequestAgent");
NativeMessaging::RequestAgentEx_var agent
    = NativeMessaging::RequestAgentEx::_narrow(ref);
```

デフォルトでは，リクエストエージェントの URL は次のとおりです。

```
corbaloc::<host>:<port>/RequestAgent
```

ここで，<host> はリクエストエージェントサーバーのホスト名またはドット付きの IP アドレスです。また，<port> は，このサーバーの TCP リスナーポート番号です。デフォルトでは，ネイティブメッセージングのリクエストエージェントはポート 5555 を使用します。

ネイティブメッセージングの Current オブジェクト

OMG のセキュリティサービスやトランザクションサービスと同様に，ネイティブメッセージングは，スレッドローカルの Current オブジェクトを使用して，2 フェーズ呼び出しを実行するための追加補助パラメータを提供およびアクセスします。このパラメータには，ブロッキングタイムアウト，リクエストタグ，Cookie，ポーラーリファレンス，応答の有効性フラグなどが含まれます。これらのパラメータのセマンティクス定義と使用方法については，後で説明します。同様に，ネイティブメッセージングの Current オブジェクトリファレンスは，ORB から初期サービスとして解決できます。次に例を示します。

```
// C++ での Current オブジェクトリファレンス取得
CORBA::Object_var ref
    = orb->resolve_initial_references("NativeMessagingCurrent");
NativeMessaging::Current_var current
    = NativeMessaging::Current::_narrow(ref);
```

コアオペレーション

2 フェーズフレームワークでは，通常の呼び出しはすべて，クライアントアプリケーションによる管理が可能な 2 つの独立した段階で実行できます。しかし，この 2 フェーズ呼び出しサービスを実行または使用する際に，フレームワークやクライアントがフレームワークのほかの基本コア機能を必要とする場合があります。基本コア機能にアクセスするために使用されるオペレーションを**コアオペレーション**といいます。この場合，次のことが望まれます。

- コアオペレーションが常に 1 フェーズ呼び出しで実行される。つまり，コアオペレーション上での呼び出しは，完了または中断するまで，常にブロックする。
- コアオペレーションは，かかわっている通常の 2 フェーズ呼び出しのすべてに対して，常に直交する。

ネイティブメッセージングでは，すべての擬似オペレーションはコアオペレーションとして予約されます。

メモ このマニュアルでは，明示的に述べていない限り，「呼び出し」または「オペレーション」は，コアオペレーションではない双方向オペレーションを意味します。

StockManager サンプル

ここでは，StockManager サンプルを使ってネイティブメッセージングの使用例を示します。このサンプルは，`<install_dir>/examples/vbe/NativeMessaging/stock_manager` ディレクトリにある製品添付の完全バージョンを短縮したものであり，CORBA Messaging StockManager サンプルと同等の機能を例示するために提供されています。

次のサンプルは，サーバーオブジェクトに次のように定義された IDL インターフェース StockManager があることを前提としています。

```
// 場所：<install_dir>/examples/vbe/NativeMessaging/
//       stock_manager/StockManager.idl
interface StockManager {
    boolean add_stock(in string symbol, in float price);
    boolean find_closest_symbol(inout string symbol);
};
```

従来の 1 フェーズ呼び出し `add_stock()` または `find_closest_symbol()` は，対象となる株式管理サーバーで，銘柄記号を追加または検索します。呼び出しコードのサンプルを次に示します。

```
// 呼び出し，呼び出し元に戻るまでブロック
CORBA::Boolean stock_added
    = stock_manager->add_stock("ACME", 100.5);
CORBA::String_var symbol = (const char*)"ACMA";
CORBA::Boolean closest_found
    = stock_manager->find_closest_symbol(symbol.inout());
```

この 1 フェーズ呼び出しの場合，呼び出しは，クライアントが戻り値または例外を受信するまでブロックされます。

ネイティブメッセージングを使用すると，2 フェーズ呼び出しを同じ株式管理サーバー上で実行できます。これらの呼び出しに対する応答は，以降の 247 ページの「ポーリング／プルモデル」と 248 ページの「コールバックモデル」に示すように，同期ポーリング／プルモデルまたは非同期コールバックモデルを使って取得または戻すことができます。

メモ このマニュアルでは，C++ の StockManager サンプルコードを示します。対応する Java コードは，『*VisiBroker for Java 開発者ガイド*』の「VisiBroker ネイティブメッセージングの使用」にあります。

ポーリング／プルモデル

ポーリング／プルモデルでは，2 フェーズ呼び出しの結果は，クライアントアプリケーションによってプルバックされます。ネイティブメッセージングのポーリング／プル 2 フェーズ呼び出しの手順を次に示します。

1. ネイティブメッセージングリクエストエージェントから要求プロキシを作成します。このプロキシは，特定のターゲットオブジェクト（このサンプルでは株式管理サーバー）用に作成され，要求をターゲットにデリゲートするために使用されます。
2. このプロキシの**型付き受信者**または **I インターフェース**を取得します。この型付き受信者は，クライアントアプリケーションが要求をプロキシに送信するために使用します。プロキシの型付き受信者は，ターゲットオブジェクトと同じ IDL インターフェースをサポートします。この例では，型付き受信者は StockManager インターフェースをサポートし，型付き StockManager スタブにナローイングできます。
3. 型付き受信者スタブ上で呼び出しを複数回行い，第 1 フェーズの呼び出しを実行します。デフォルトでは，型付き受信者上の呼び出しは，ダミーの出力と戻り値と一緒に返されます。これを**早期リターン**といいます。プロキシの型付き受信者から例外を生成することなく早期リターンを受信した場合，それは，2 フェーズ呼び出しが正常に開始されたことを示します。また，要求が受け付けられ，リクエストエージェントによって別個のポーリングオブジェクトに割り当てられたことを示します。2 フェーズ呼び出しのポーリングオブジェクトは，ローカルの NativeMessaging Current から利用できます。型付き受信者と同様に，すべてのポーリングオブジェクトも，ターゲットオブジェクトと同じ IDL インターフェース（このサンプルでは StockManager）をサポートします。
4. 可用性をポーリングし，応答をポーリングオブジェクトからプルバックして，呼び出しの第 2 フェーズを実行します。クライアントアプリケーションはポーリングオブジェクトを対応する型付き受信者スタブ（このサンプルでは StockManager）にナローイングし，要求送信フェーズで呼び出されたのと同じオペレーションを呼び出します。ポーリングオブジェクト上で呼び出しを実行する場合，入力パラメータは無視されます。また，エージェントは，デリゲートされたターゲットオブジェクトへは新しい要求を配信しません。エージェントは，ポーリングオブジェクト上で実行されたすべての呼び出しをポーリング／プルリクエストとして扱います。通常，NativeMessaging Current を介してタイムアウト値を補助パラメータとして提供し，最大ポーリングブロッキングタイムアウトを指定できます。タイムアウト前に応答があった場合，ポーリング呼び出しは，実際の呼び出しから出力パラメータと戻り値とともに**処理完了リターン**を受信します。そうではなく，タイムアウト経過後も応答がなかった場合，ポーリングは最終的に早期リターンをもう一度返します。アプリケーションは，Native Messaging Current の `reply_not_available` 属性を使用して，ポーリングリターンが早期リターンかどうかを判別する必要があります。

次のサンプルコードは，ネイティブメッセージングを使って株式管理オブジェクト上でポーリング／プル 2 フェーズ呼び出しを実行する方法を示したものです。

```
// 場所：<install_dir>/examples/vbe/NativeMessaging/
//       stock_manager/polling_client_main.C

// 1. 対象 stock_manager サーバー上に非ブロッキング要求を作成するための
//    要求プロキシをリクエストエージェントから作成します。
NativeMessaging::RequestProxy_var proxy
    = agent->create_request_proxy(
        stock_manager, "",
        NULL, NativeMessaging::PropertySeq(0));

// 2. プロキシの要求（型付き）受信者を取得します。
CORBA::Object_var ref;
StockManager_var stock_manager_rcv
    = StockManager::_narrow(ref = proxy->the_receiver());

// 3. 複数の要求を型付き受信者に送信し，
```

```
//    ネイティブメッセージングの Current オブジェクトから応答ポーラーを取得します。
StockManager_var pollers[2];
stock_manager_rcv->add_stock("ACME", 100.5);
pollers[0] = StockManager::_narrow(ref = current->the_poller());
CORBA::String_var symbol = (const char*)"ACMA";
stock_manager_rcv->find_closest_symbol(symbol.inout());
pollers[1] = StockManager::_narrow(ref = current->the_poller());

// 4. 2 つの関連付けられた応答をポーリング／プルします。
current->wait_timeout(max_timeout);

CORBA::Boolean stock_added;
do { stock_added = pollers[0]->add_stock("", 0.0); }
while(current->reply_not_available());

CORBA::Boolean closest_found;
do { closest_found = pollers[1]->find_closest_symbol(symbol.inout()); }
while(current->reply_not_available());
```

メモ

- ネイティブメッセージングでは，2 フェーズ呼び出しの要求送信フェーズと応答ポーリング／プル段階はすべて同じオペレーションシグニチャを使用します。2 フェーズ呼び出しの 2 つの段階の両方で使用されるこのオペレーションは，実際のターゲットの IDL インターフェースで定義されているのとまったく同じ**ネイティブな**オペレーションです。
- ポーリングオブジェクトは，ネットワーク上の位置透過性を持つ通常の CORBA オブジェクトです。したがって，ネイティブメッセージングでは，2 フェーズ呼び出しの要求送信フェーズと応答ポーリング段階は，同じクライアント実行コンテキスト内で，かつ同じ転送接続を使って行う必要はありません。
- ポーリング／プル段階で例外が発生した場合，アプリケーションは，Current の `reply_not_available` 属性を使用して，例外が応答ポーリング／プルのエラーによるものなのか，デリゲートされた要求の実際の結果である例外を正常にプルしたものなのかを判別する必要があります。`TRUE` は，例外がクライアントとエージェント間のポーリング／プルエラーであることを示します。`FALSE` は，例外がデリゲートされた要求の実際の結果であることを示します。
- 早期リターンでは，ネイティブメッセージングは，すべての非プリミティブの出力パラメータと戻り値を null に設定します。これは，ネイティブメッセージングが CORBA 環境ではなくローカルの Current オブジェクトを使用する点を除いて，OMG の例外処理以外の C++ マッピングに似ています。

追加機能，さまざまなポーリング／プルモデル，ネイティブメッセージグの API の構文およびセマンティクスの仕様については，251 ページの「高度な項目」と 257 ページの「ネイティブメッセージングの API 仕様」で説明します。

コールバックモデル

ネイティブメッセージングのコールバックモデルを使用すると，アプリケーションは，要求をプロキシの型付き受信者に送信後すぐにアンブロックされます。これらの呼び出しに対する応答は，要求プロキシ作成時に指定されたコールバック応答受信者に配信されます。

ネイティブメッセージングの 2 フェーズ呼び出しをコールバックモデルで実行する手順を次に示します。

1. ネイティブメッセージングリクエストエージェントから要求プロキシを作成します。このプロキシは，特定のターゲットオブジェクト用に作成されます。ポーリング／プルモデルと同様に，このプロキシは，指定されたターゲットに要求をデリゲートするために使用されます。応答受信者コールバックハンドラは，ポーリング／プルモデルでは null リファレンスですが，この要求プロキシ作成時にも指定されます。リクエストエージェントは，このプロキシによってデリゲートされた要求に対して新しく行われた応答をすべてコールバックハンドラに配信します。

2 ポーリング／プルモデルの 2 番めの手順と同様に，このプロキシの**型付き受信者**または **I インターフェース**を取得し, それを型付き I スタブ（このサンプルでは StockManager スタブ）にナローイングします。

3 ポーリング／プルモデルの 3 番めの手順と同様に，プロキシの型付き受信者スタブ上で呼び出しを複数回実行して第 1 呼び出し段階を実行します。デフォルトでは，型付き受信者上の呼び出しは，ダミーの出力と戻り値と一緒に返されます。これを**早期リターン**といいます。プロキシの型付き受信者上での例外なしの早期リターンは，2 フェーズ呼び出しが正常に開始されたことを示します。

4 応答を受信して，呼び出しの第 2 フェーズを完了します。コールバックモデルでは，これは，完全に独立した実行コンテキスト内で非同期に実行されます。クライアントアプリケーションは，応答受信者オブジェクトを実装し，アクティブ化します。このコールバックオブジェクトはタイプ固有のオブジェクトではなく, 実際のターゲットの IDL インターフェースに依存しません。このコールバックハンドラのキーとなるオペレーションは，サンプルコードの後で説明する reply_available() メソッドです。

次のサンプルコードは, コールバックモデルの 2 フェーズ呼び出しを stock manager オブジェクト上で実行するためにネイティブメッセージングを使用するための最初の 3 手順を示したものです。

```
// 場所：<install_dir>/examples/vbe/NativeMessaging/
//       stock_manager/callback_client_main.C

// タイプに依存しないコールバックハンドラリファレンスを取得します。
NativeMessaging::ReplyRecipient reply_recipient = …;

// 1. 対象 stock_manager サーバー上に非ブロッキング要求を作成するための
//    要求プロキシをリクエストエージェントから作成します。
NativeMessaging::RequestProxy_var proxy
    = agent->create_request_proxy(
        stock_manager, "", reply_recipient,
        NativeMessaging::PropertySeq(0));

// 2. プロキシの要求（型付き）受信者を取得します。
StockManager_var stock_manager_rcv
    = StockManager::_narrow(obj = proxy->the_receiver());

// 3. 2 つの要求を受信者に送信します。
stock_manager_rcv->add_stock("ACME", 100.5);
CORBA::String_var symbol = (const char*)"ACMA";
stock_manager_rcv->find_closest_symbol(symbol.inout());
```

ここで，reply_recipient コールバックハンドラは，特定のアプリケーションのターゲットタイプに関係なく，NativeMessaging::ReplyRecipient オブジェクトです。ReplyRecipient インターフェースは次のように定義されます。

```
// 場所：<install_dir>/idl/NativeMessaging.idl

interface NativeMessaging::ReplyRecipient {
    void reply_available(
        in object reply_holder,
        in string operation,
        in sequence<octet> the_cookie);
    );
};
```

reply_available() の reply_holder パラメータは**反射コールバック**リファレンスと呼ばれ，ポーリング／プルモデルの応答ポーラーオブジェクトと同じです。これは，ポーリング／プルモデルのクライアントがポーリングオブジェクトから応答結果をプルバックするのと同じ方法で，reply_available() インプリメンテーションが応答結果をプルバックするために使用できます。

メモ 応答をコールバックハンドラに配信する際，ネイティブメッセージングは，**二重ディスパッチ**パターンを使ってコールバックモデルをポーリング／プルモデルに**反転**させます。この

とき，応答受信者インプリメンテーションは，応答を取得するために型付き reply_holder リファレンス上で 2 番めの（反射）コールバックを実行します。

次のコードは，reply_available() メソッドのサンプルインプリメンテーションです。

```
// 場所：<install_dir>/examples/vbe/NativeMessaging/
//       stock_manager/AsyncStockRecipient.C

void StockManagerReplyRecipientImpl::reply_available(
CORBA::Object_ptr reply_holder,
    const char* operation,
    const CORBA::OctetSequence& cookie)
{
    StockManager_var poller
        = StockManager::_narrow(reply_poller);

    // 反射コールバックを使って応答を取得します。
    if( strcmp(operation, "add_stock") == 0 ) {
        // add_stock() の戻り値を取得します。
        CORBA::Boolean stock_added
            = poller->add_stock("", 0.0);
        ...
    }
    else
    if( strcmp(operation, "find_closest_symbol") == 0 ) {
        CORBA::String_var symbol = (const char*)"";
        // find_closest_symbol() の戻り値を取得します。
        CORBA::Boolean closest_found
            = poller->find_closest_symbol(symbol.inout());
        ...
     }
...
}
```

メモ

- ネイティブメッセージングでは，2　フェーズ呼び出しの要求送信フェーズと応答受信フェーズはどちらも同じオペレーションを使用します。2 フェーズ呼び出しの 2 つの段階の両方で使用されるオペレーションは，実際のターゲットの IDL インターフェースで定義されているのとまったく同じ**ネイティブな**オペレーションです。
- 応答受信者オブジェクトは，通常の CORBA オブジェクトで，ネットワーク上での位置が透過的です。したがって，ネイティブメッセージングでは，応答受信者コールバックオブジェクトは，必ずしも要求送信元のクライアントプロセス内にあるとは限りません。
- reply_available() インプリメンテーションが reply_holder から応答を取得する際に例外が発生した場合，アプリケーションは Current の reply_not_available 属性を使用して，例外がエラーの取得を報告しているのか，デリゲートされた要求の実際の結果が例外であり，その結果を正常に取得したことを報告しているのかを判別する必要があります。TRUE は，この例外がクライアントとエージェント間の応答取得エラーであることを示します。FALSE は，この例外がデリゲートされた要求の実際の結果であることを示します。
- reply_holder での応答取得オペレーションは，reply_available() メソッドの範囲内でのみ実行する必要があります。アプリケーションが reply_available() から戻ると，reply_holder は有効でなくなります。

追加機能，さまざまなポーリング／プルモデル，ネイティブメッセージングの API 仕様については，251 ページの「高度な項目」と 257 ページの「ネイティブメッセージングの API 仕様」で説明します。

高度な項目

グループポーリング

前記の例で示したように，特定の要求プロキシが複数の要求をデリゲートできます。ただし，要求によって処理時間が異なるため，要求からの応答は，必ずしも要求が呼び出された順序で準備完了状態になっているとは限りません。個々の要求を 1 つずつポーリングするかわりに，グループポーリングを使用すると，特定の要求プロキシによって複数の要求がデリゲートされているポーリングクライアントアプリケーションは，多重化集合内で応答の可用性を判別できます。

要求をグループポーリングに参加させるには，特定のプロキシに送信される要求にタグを付ける必要があります。リクエストタグは，グループ，つまり要求プロキシのスコープ内で要求を識別するために，クライアントによって割り当てられます。ネイティブメッセージングでは，スコープ（要求プロキシ）内で一意でなければならないことを除いて，リクエストタグの内容に制約はありません。タグが付けられていない要求（空白タグの要求）はグループポーリングには関与せず，そのような要求の応答の可用性は，グループポーリングの結果では報告されません。

グループポーリングを使用する手順を次に示します。

1 タグ付き要求を送信します。要求にタグを付けるには，クライアントアプリケーションは，型付き受信者インターフェースで（要求の配信前に）呼び出しを行う前に，ローカルのネイティブメッセージング Current オブジェクトの `request_tag` 属性を設定します。リクエストタグの内容は，スコープ（プロキシ）内で一意である限り，アプリケーションが自身の都合に合わせて指定できます。

2 プロキシの `poll(max_timeout, unmask)` オペレーションを呼び出すことにより，個々のポーラーではなく要求プロキシに対して応答の可用性をポーリングします。このオペレーションは，タイムアウトになるまで，またはプロキシによってデリゲートされたタグ付き要求が処理完了リターンを返す準備が完了するまで，ブロックします。処理完了リターンを返す準備が完了すると，タグは，返されるリクエストタグシーケンスに入れられます。空白のタグシーケンスが返された場合，タイムアウトが終了したことを示します。

3 グループポーリングの戻り値によって処理完了リターンの準備ができたことを報告した個々のポーラーから，応答結果を取得します。

次のサンプルコードは，ネイティブメッセージングのグループポーリング機能を使用する上記の手順の例を示したものです。

```
// 場所：<install_dir>/examples/vbe/NativeMessaging/
//       stock_manager/group_polling_client.C

// タグ付き要求を 1 つ送信します。
current->request_tag(NativeMessaging::RequestTag(2,2, (CORBA::Octet*)"0"));
stock_manager_rcv->add_stock("ACME", 100.5);
pollers[0] = StockManager::_narrow(ref = current->the_poller());

// タグ付き要求をもう 1 つ送信します。
current->request_tag(NativeMessaging::RequestTag(2,2, (CORBA::Octet*)"1"));
CORBA::String_var symbol = (const char*)"ACMA";
stock_manager_rcv->find_closest_symbol(symbol.inout());
pollers[1] = StockManager::_narrow(ref = current->the_poller());

...

// プロキシで要求の可用性をポーリングし，応答を取得します。
NativeMessaging::RequestTagSeq_var tags;
while(TRUE) {
    // 可用性をポーリングします。
    try {
```

```
        tags = proxy->poll(max_timeout, TRUE);
    }
    catch(NativeMessaging::RequestAgent::PollingGroupIsEmpty&) {
        proxy->destroy(TRUE);
        break;
    }

    // 応答を取得します。
    for(int i=0;i<tags->length();i++) {
        int id = atoi((const char*)((tags.in())[i].get_buffer()));

        switch(id) {
            case 0: // 最初に送信されたタグ付き要求
                CORBA::Boolean stock_added;
                stock_added = pollers[0]->add_stock("", 0.0);
                break;

            case 1: // 2 番めに送信されたタグ付き要求
                CORBA::Boolean closest_found;
                closest_found = pollers[1]->find_closest_symbol(symbol.inout());
                break;

            default:
                break;
        }
    }
}
```

メモ

- 各呼び出し後，Current の request_tag 属性は自動的に空または null にリセットされます。
- すでに別の 2PI またはプロキシによって使用された request_tag を使ってプロキシで 2PI を開始しようとすると，マイナーコードが NativeMessaging::DUPLICATED_REQUEST_TAG の CORBA BAD_INV_ORDER 例外が生成されます。
- 要求プロキシの poll() オペレーションの unmask パラメータは，poll() がすべての処理完了要求のマスクを解除するかどうかを指定します。マスクを解除した場合，その次の poll() は，それらの処理完了要求を処理せず，報告しません。
- プロキシ上のすべての要求にタグが付けられておらず, マスクも解除されていない場合，poll() は PollingGroupIsEmpty 例外を生成します。

応答受信者における Cookie と応答逆多重化

前記の例で示したように，特定の要求プロキシが複数の要求をデリゲートできます。コールバックモデルでは，これらの要求への応答はすべて，プロキシ作成時に指定された同じ応答受信者オブジェクトに送り返されます。問題は, クライアントが 1 つの ReplyRecipient コールバックハンドラ上でさまざまな応答をどのように逆多重化するかです。

OMG CORBA Messaging を使用するアプリケーションも同じ問題に直面します。多くのコールバックオブジェクトがアクティブ化されるのを避けるために，CORBA Messaging では，アプリケーションが POA のデフォルトサーバントまたはサーバントマネージャを使ってコールバックオブジェクトを操作し，コールバックリファレンスごとに異なるオブジェクト ID を割り当てることが推奨されています。これは，応答受信者プロセス内で多くのコールバックオブジェクトがアクティブ化されるのを避けることができますが，各コールバック要求を送信するためにオブジェクトリファレンスを作成してマーシャリングする必要があるため，柔軟性のない非効率的な方法です。

ネイティブメッセージングでは，2 つの逆多重化メカニズムをサポートしており，それらは，逆多重化粒度の必要性に応じて一緒にまたは単独で使用できます。オペレーションシグニチャによる逆多重化は，粗粒度ですが便利なメカニズムで，ReplyRecipient の

reply_available() コールバックメソッド内で使用できます。このメカニズムは，前述のサンプルの一部で使用されています。

ネイティブメッセージングのコールバックモデルにおけるさらに効率的な逆多重化メカニズムは，**要求 Cookie** の使用です。要求 Cookie は，Octet のシーケンス（またはバイト配列）です。Cookie の内容は，要求を送信する前にネイティブメッセージングの Currento オブジェクトでクライアントアプリケーションによって指定されます。指定された Cookie は，その要求の応答を配信する際に応答受信者の reply_available() メソッドに渡されます。Cookie の内容に制約はまったくなく，一意性も必要ありません。Cookie の内容は，コールバックの逆多重化の際にアプリケーション自身にとって便利で効率がよいように，アプリケーションが決定します。

次のサンプルコードは，Cookie を要求に割り当てる方法を示したものです。

```
// Cookie 付きの要求を送信します。
current->the_cookie(CORBA::OctetSeq(9,9,"add stock"));
stock_manager_rcv->add_stock("ACME", 100.5);

// 別の Cookie を付けた別の要求を送信します。
current->the_cookie(CORBA::OctetSeq(11,11,"find symbol"));
CORBA::String_var symbol = (const char*)"ACMA";
stock_manager_rcv.find_closest_symbol(symbol.inout());
```

次のサンプルコードは，応答受信者が添付 Cookie を使って逆多重化を行う方法を示したものです。

```
void StockManagerReplyRecipientImpl::reply_available(
    CORBA::Object_ptr reply_poller,
    const char* operation,
    const CORBA::OctetSequence& cookie)
{
    StockManager_var poller
        = StockManager::_narrow(reply_poller);

    CORBA::String_var id = PortableServer::
        ObjectId_to_string(cookie.get_buffer());

    // 反射コールバックを使って応答を取得します。
    if( strcmp(id, "add stock") == 0 ) {
        CORBA::Boolean stock_added
            = poller->add_stock("", 0.0);

        ...
 }
    else
    if( strcmp(id, "find symbol") == 0 ) {
        CORBA::String_var symbol = (const char*)"";
        CORBA::Boolean closest_found
            = poller->find_closest_symbol(symbol.inout());
        ...
    }
        ...
}
```

2 フェーズ呼び出しへの展開

従来の 1 フェーズ呼び出しと比較して，2 フェーズ呼び出しでは，応答ポーリング通信の往復が追加で発生します。長時間かかる重いタスクでは，ほとんど発生しない追加の通信往復による遅延は重要ではありません。しかし，軽い短時間の呼び出しでは，この遅延が望ましくない場合があります。

アプリケーションにとって，軽い短時間の呼び出しは追加の遅延を発生させずに 1 フェーズで完了でき，重くて長時間かかる呼び出しは，自動的に 2 フェーズで実行してクライアントの実行コンテキストと転送接続を保持しなくて済むのが，理想的です。

ネイティブメッセージングでは，**2 フェーズ呼び出しへの展開**機能を使用してこれを実現できます。デフォルトでは，プロキシの型付き受信者で呼び出しを行うと，必ず，応答結果と一緒に早期リターンがポーリングバックされるか，別の呼び出し段階で後でコールバックを通して配信されます。**2 フェーズ呼び出しへの展開**機能を使用すると，プロキシの型付き受信者での呼び出しは，指定されたタイムアウトが終了する前に完了できる場合は，ブロックして処理完了リターンを生成します。そうではなく，タイムアウトが終了するまでに呼び出しを完了できない場合は，早期リターンを生成することにより，2 フェーズ呼び出しに展開します。プロキシの型付き受信者での呼び出しが 2 フェーズ呼び出しに展開したかどうかを確認するには，ローカルのネイティブメッセージング Current オブジェクトの `reply_not_available` 属性をリターン後に調べます。

この機能を使用するには，次の操作を実行します。

- 要求プロキシは，`WaitReply` プロパティの値を `TRUE` に設定して作成する必要があります。
- 呼び出しの前に，ネイティブメッセージング Current の `wait_timeout` 属性をゼロ以外の値（ミリ秒）に設定します。
- 型付き受信者での各呼び出しの後で，ローカルのネイティブメッセージング Current オブジェクトの呼び出し後の `reply_not_available` 属性を調べることにより，リターンが早期リターンかどうかを判別します。
- リターンが早期リターンの場合，後で別の段階で応答をポーリングするために，ローカルの Current から返されたポーリングオブジェクトを取得します。

次のサンプルコードは，2 フェーズ呼び出しへの展開の使用方法を示したものです。

```
// WaitReply プロパティを TRUE に設定して要求プロキシを作成します。
NativeMessaging::PropertySeq props;
props.length(1);
props[0].id = (const char*)"WaitReply";
CORBA::Any::from_boolean fb((CORBA::Boolean)1);
props[0].value <<= fb;

NativeMessaging::RequestProxy_var proxy
    = agent->create_request_proxy(stock_manager, "", NULL, props);

// このプロキシの型付き受信者を取得します。
CORBA::Object_var ref;
StockManager_var stock_manager_rcv
    = StockManager::_narrow(ref = proxy->the_receiver());

// wait_timeout 属性を 3 秒に設定します。
current->wait_timeout(3000);
// 受信者上で呼び出しを行います。
CORBA::Boolean stock_added = stock_manager_rcv->add_stock("ACME", 100.5);

// 2 フェーズ呼び出しに展開したかどうかを確認します。
if( ! current->reply_not_available() ) {
    // 展開されていません。上記は処理完了リターンです。
    // ジョブは完了しました。
    return;
}

// 2 フェーズ呼び出しに展開しました。
// ポーラーを取得して応答をポーリングする必要があります。
StockManager_var poller
    = StockManager::_narrow(ref = current->the_poller());
do { stock_added = pollers->add_stock("", 0.0); }
while(current->reply_not_available())
```

メモ
- プロキシの型付き受信者上のオペレーションが，タイムアウトになって 2 フェーズ呼び出しに展開する前に完了できる場合，ポーラーは生成されず，応答を配信するためのコールバックも応答受信者上に作成されません。

- プロキシでブロックする際，または応答をポーリングする際に例外が発生した場合，アプリケーションは，ネイティブメッセージングの Current の reply_not_available 属性を使用して，例外が要求配信または応答ポーリングでのエラーを報告しているのか，または要求をデリゲートした実際の結果を報告しているのかを判別する必要があります。この属性の値が TRUE の場合，この例外が，クライアントとエージェント間における応答の配信またはポーリングのエラーであることを示します。FALSE は，この例外が要求をデリゲートした実際の結果であることを示します。

応答ドロップ

コールバックモデルでは，リクエストエージェントは，デフォルトで，呼び出しの結果をすべて，戻り値か例外かに関係なく，応答受信者に送り返します。応答ドロップを使用すると，特定のタイプの応答結果をフィルタアウトできます。この機能は，たとえば，アプリケーションが結果が返されない 1 方向の要求を呼び出すが，呼び出しが失敗した場合には通知を受ける場合に便利です。

ネイティブメッセージングでは，アプリケーションは要求プロキシ作成時に ReplyDropping プロパティを指定できます。このプロパティは，応答受信者に送信されないようにフィルタアウトする戻り型を指定します。このプロパティの値は Octet（またはバイト）で，次のフィルタ規則にしたがいます。

- 値が 0x01 の場合，通常の応答をドロップします。
- 値が 0x02 の場合，システム例外をドロップします。
- 値が 0x04 の場合，ユーザー例外をドロップします。

たとえば，このプロパティの値を 0x06 に設定すると，リクエストエージェントは，このプロキシによってデリゲートされた要求上のシステム例外とユーザー例外をすべてドロップします。

次のサンプルコードは，ReplyDropping プロパティの設定方法を示したものです。

```
// ReplyDropping プロパティの値を 0x01（通常の応答をすべてドロップ）に
// 設定して要求プロキシを作成します
NativeMessaging::PropertySeq props;
props.length(1);
props[0].id = (const char*)"ReplyDropping";
CORBA::Any::from_octet fo((CORBA::Octet)0x01);
props[0].value <<= fo;

NativeMessaging::RequestProxy_var proxy
    = agent->create_request_proxy(stock_manager, "",
        reply_recipient, props);

    ...
```

メモ

- 応答ドロップは，コールバックモデルにだけ適用されます。create_request_proxy() に渡される reply_recipient リファレンスが null の場合, 応答ドロッププロパティは無視されます。
- create_request_proxy() の応答ドロッププロパティの値が 0x00 ではなく，かつ reply_recipient リファレンスが *null* でない場合，このプロキシの型付き受信者での呼び出しは, ネイティブメッセージングの Current に対してポーリングオブジェクトを返しません。

コレクションの手動破棄

デフォルトでは，ポーラーオブジェクトは，ポーリングオペレーションの結果，処理完了リターンが生成されるとすぐに破棄されます。コールバックモデルでは，コールバックが返されると，アプリケーションがコールバックの reply_available() オペレーション内で応答を取得したかどうかにかかわらず，リクエストエージェントもポーラーを破棄します。

破棄されたオブジェクトに対してポーリングを行うと，CORBA OBJECT_NOT_EXIST 例外が生成され，Current の reply_not_available 属性は TRUE に設定されます。

RequestManualTrash プロパティの値を TRUE に設定して要求プロキシを作成すると，このプロキシによってデリゲートされる要求のポーラーオブジェクトは，自動的に破棄されません。応答が可能になった後でこれらのポーラーオブジェクトに対してポーリングを行うと，ポーリングが等べきになり，毎回同じ結果が返されます。

これらのポーラーオブジェクトは，アプリケーションで必要がなくなった場合，手動で破棄できます。ポーラーオブジェクトを手動で破棄するには，リクエストエージェントで destroy_request() オペレーションを破棄するポーラーをパラメータとして指定して呼び出します。たとえば，次のようにします。

```
agent->destroy_request(poller);
```

メモ　自動破棄プロキシによってデリゲートされた要求のポーラーも，手動で破棄できます。そのようなポーラーで応答がまだ可能になっていないか，またはポーリングバックされていない場合，この機能は有効です。

非抑制早期リターンモード

ネイティブメッセージングの主要な概念は，呼び出しの第 1 フェーズ後のネイティブオペレーションのアンブロックです。ネイティブメッセージングでは，これを**早期リターン**といいます。ネイティブメッセージングには，**抑制**モードと**非抑制**モードの 2 つの早期リターンモードがあります。ここまではすべて，デフォルトの抑制モードを使って説明を行ってきました。抑制モードでは，早期リターンは，ダミー出力と戻り値を含む以外は，通常のオペレーションの戻りと同じです。これは，ネイティブメッセージングが追加の環境パラメータではなくスレッドローカルの Current オブジェクトを使用する点を除いて，OMG の C++ マッピングの例外処理以外の処理における例外リターンに似ています。

抑制早期リターンモードは便利ですが，クライアント側のマッピングのサポートを必要とします。つまり，IDL プリコンパイラによって生成されたクライアント側のスタブコードが早期リターンの例外を補足して抑制することを前提としています。クライアントアプリケーションを ORB に移植する場合，IDL プリコンパイラは早期リターンが抑制されたクライアント側スタブコードを生成しないため，非抑制早期リターンモードを使用します。

ネイティブメッセージングの非抑制早期リターンモードでは，ネイティブオペレーションは，RNA 例外，つまりマイナーコード REPLY_NOT_AVAILABLE の CORBA NO_RESPONSE 例外を生成するだけでアンブロックできます。非抑制早期リターンモードを使用するには，アプリケーションで，ネイティブメッセージングの Current に対して suppress_mode(false) を呼び出して抑制モードをオフにする必要があります。また，これに伴って，アプリケーションは，RNA 例外を補足して処理する必要があります。

メモ　コードを抑制モードと非抑制モードの両方に移植できるようにするには，アプリケーションで，RNA 例外とマイナーコードではなく，Current の reply_not_available 属性を非抑制モードで使用して，戻りの完了状態を判別することをお勧めします。

次のサンプルコードは，非抑制モードでの StockManager のポーリング例を示したものです。このコードはすべての ORB に移植できるだけでなく，抑制モードにも移植できます。

```
// 場所：<install_dir>/examples/vbe/NativeMessaging/
//       stock_manager/polling_client_portable.C

static void yield_non_rna(const CORBA::NO_RESPONSE& e)
{
    if(e.minor() != NativeMessaging::REPLY_NOT_AVAILABLE ) {
        throw e;
    }
}

...

// このマクロは，RNA，つまりマイナーコードが NativeMessaging::REPLY_NOT_AVAILABLE
```

```
// の NO_RESPONSE 例外を抑制します
#define SUPPRESS_RNA(stmt)  ¥
    try { stmt; } ¥
    catch(const CORBA::NO_RESPONSE& e) { yield_non_rna(e); }

...

// 抑制モードをオフにします。
current->suppress_mode(FALSE);

// 複数の要求を型付き受信者に送信し,
// ネイティブメッセージングの Current オブジェクトから応答ポーラーを取得します。
StockManager_var pollers[2];
SUPPRESS_RNA( stock_manager_rcv->add_stock("ACME", 100.5) )
pollers[0] = StockManager::_narrow(ref = current->the_poller());

CORBA::String_var symbol = (const char*)"ACMA";
SUPPRESS_RNA(stock_manager_rcv->find_closest_symbol(symbol.inout()))
pollers[1] = StockManager::_narrow(ref = current->the_poller());

// 関連付けられている応答をポーリングします。
current->wait_timeout(max_timeout);

CORBA::Boolean stock_added;
do { SUPPRESS_RNA(stock_added = pollers[0]->add_stock("", 0.0)) }
while(current->reply_not_available());

CORBA::Boolean closest_found;
do { SUPPRESS_RNA(closest_found
    = pollers[1]->find_closest_symbol(symbol.inout())) }
while(current->reply_not_available());
```

コールバックモデルでのポーラー生成の抑制

デフォルトでは,ポーラーはコールバックモデルでも生成されます。これにより,次の操作が可能です。

- アプリケーションは,要求が完了する前に要求を破棄できます。
- アプリケーションは,応答受信者と関係なく応答を取得できます。

ただし,ポーラーリファレンスの生成と送信は,付加的なオーバーヘッドを発生させます。ネイティブメッセージングでは,コールバックモデルでのポーラーリファレンスの生成を抑制(無効化)できます。

コールバックモデルでのポーラーの生成を抑制するには,CallbackOnly プロパティを TRUE に設定して要求プロキシを作成する必要があります。この場合 null のポーラーが返されます。

ネイティブメッセージングの API 仕様

メモ　ネイティブメッセージングの IDL 定義のいくつかのオペレーションと属性については,このマニュアルでは扱っていません。扱っていないのは,付加価値を付けるための機能,非推奨になった機能,将来の拡張のために予約されている機能などです。

RequestAgentEx インターフェース

ネイティブメッセージングリクエストエージェントのインターフェースです。リクエストエージェントは,指定されたターゲットオブジェクトに呼び出しをデリゲートし,クライアントのコールバックハンドラに戻り値を配信するか,またはクライアントのポーリング

時に戻り値を返す役割を果たします。詳細については，245 ページの「リクエストエージェント」を参照してください。

create_request_proxy()

```
RequestProxy
create_request_proxy(
    in object target,
    in string repository_id,
    in ReplyRecipient reply_recipient,
    in PropertySeq properties)
  raises(InvalidProperty);
```

create_request_proxy() メソッドは，2 フェーズ呼び出しを指定したターゲットオブジェクトにデリゲートするための要求プロキシを作成します。

引数	説明
target	このプロキシによってデリゲートされるすべての要求のターゲット。
repository_id	このプロキシから型付き受信者，応答ポーラー，応答ホルダーに割り当てられたリポジトリ ID。このパラメータが空文字列の場合，ターゲットのリポジトリ ID が使用されます。この ID は，型付き受信者，応答ポーラー，応答ホルダーで _is_a() セマンティクスを満たすためにネイティブメッセージングが使用します。
reply_recipient	応答受信者コールバックハンドラ。応答が可能になると，リクエストエージェントは reply_available() オペレーションをコールバックして応答結果を送り返します。null_reply_recipient は，ポーリング／プルモデルを意味します。
properties	プロキシのデフォルト以外のセマンティクスを指定するためのプロパティ。次のプロパティがサポートされています。 • WaitReply：デフォルト値が FALSE の論理プロパティ。詳細については，253 ページの「2 フェーズ呼び出しへの展開」を参照してください。 • RequestManualTrash：デフォルト値が FALSE の論理プロパティ。詳細については，255 ページの「コレクションの手動破棄」を参照してください。 • ReplyDropping：デフォルト値が 0x00 の Octet 値のプロパティ。詳細については，255 ページの「応答ドロップ」を参照してください。 • CallbackOnly：デフォルト値が FALSE の論理プロパティ。詳細については，257 ページの「コールバックモデルでのポーラー生成の抑制」を参照してください。

例外	説明
InvalidProperty	この例外は，無効なプロパティ名または値がプロパティリストで使用されていることを示します。プロパティ名は，例外から取得できます。

destroy_request()

```
void
destroy_request(
    in object poller)
  raises(RequestNotExist);
```

このメソッドは，ポーラーオブジェクトを手動で破棄するために使用されます。詳細については，255 ページの「コレクションの手動破棄」を参照してください。

引数	説明
poller	破棄するポーラー。

例外	説明
RequestNotExist	この例外は，破棄するポーラーを使用できないことを示します。

RequestProxy インターフェース

要求プロキシは，要求を指定されたターゲットに指定されたセマンティクスプロパティを使ってデリゲートするために，アプリケーションがリクエストエージェントから作成します。258 ページの「create_request_proxy()」を参照してください。

the_receiver

```
readonly attribute object the_receiver;
```

この属性は，プロキシの型付き受信者リファレンスです。プロキシの型付き受信者は，指定されたターゲットと同じ IDL インターフェースをサポートし，アプリケーションがプロキシによってデリゲートするために要求を送信する送信先です。

メモ

- デフォルトでは，プロキシの型付き受信者でオペレーションを呼び出すと，2 フェーズ呼び出しが開始され，このプロキシによってデリゲートされます。これらの呼び出しはアンブロックされ，別の応答ポーラーを生成します。
- WaitReply プロパティの値を TRUE に設定してプロキシが作成され，ゼロ以外の wait_timeout 値を持つ要求が the_receiver で呼び出された場合，リクエストエージェントは，タイムアウトが終了する前に，要求を 1 フェーズ呼び出しとしてデリゲートしようとします。タイムアウトが終了する前にエージェントがターゲットから応答を受信しなかった場合，エージェントはクライアントをアンブロックし，要求は 2 フェーズ呼び出しに展開します。the_receiver での呼び出しをアンブロック後は，アプリケーションは，Current の reply_not_available 属性を使って要求が 2 フェーズ呼び出しに展開したかどうかを判別できます。261 ページの「reply_not_available」を参照してください。
- IDL の一方向オペレーションの呼び出しは，本来，1 フェーズだけであり，したがって，プロキシの型付き受信者での一方向呼び出しはポーラーオブジェクトを生成しません。エージェントは，一方向呼び出しを呼び出しの第 2 フェーズを通過させずにターゲットに転送します。
- プロキシの型付き受信者でのコアオペレーションは同期的に処理され，処理完了リターンまたは例外が生成されるまでブロックされます。型付き受信者でコアオペレーションを呼び出しても，2 フェーズ呼び出しは開始されません。たとえば，プロキシの型付き受信者での _non_existent() 呼び出しは，実際のターゲットではなく，受信者自身を ping するだけです。

poll()

```
RequestIdSeq
poll(
    in unsigned long timeout,
    in boolean unmask)
  raises(PollingGroupIsEmpty);
```

このメソッドは，グループポーリングを実行します。詳細については，251 ページの「グループポーリング」を参照してください。

引数	説明
timeout	タグ付き要求が使用可能になるまでこのメソッドが待機する最大時間をミリ秒単位で指定します。タグ付き要求がタイムアウト終了前に利用可能にならなかった場合，空の RequestIdSeq が返されます。
unmask	返されるシーケンスにタグが含まれるタグ付き要求のマスクを解除するかどうかを指定します。マスクを解除したタグ付き要求は，その後のグループポーリングには含まれません。

例外	説明
PollingGroupIsEmpty	この例外は，タグ付き要求またはマスクを解除された要求で，このプロキシで保留中になっているものは存在しないことを示します。

destroy()

```
void
destroy (
    in boolean destroy_requests);
```

このメソッドは，要求プロキシを破棄します。

引数	説明
destroy_requests	TRUE の場合，このプロキシによってデリゲートされたすべての要求は破棄されます。

ローカルインターフェースの Current オブジェクト

ローカルのネイティブメッセージングの Current オブジェクトは，2 フェーズ呼び出しの前後に追加情報を指定およびアクセスするためにアプリケーションが使用します。Current オブジェクトは，初期リファレンスとしてローカルの ORB から解決できます。詳細については，245 ページの「ネイティブメッセージングの Current オブジェクト」を参照してください。

suppress_mode()

```
void
suppress_mode(
    in boolean mode);
```

これは，現在の早期リターンモードを設定します。抑制モードでは，第 1 フェーズ終了後にアンブロックされ，ダミー出力と戻り値を含む以外は，通常どおりに戻ります。非抑制モードでは，2 フェーズ呼び出しは RNA 例外（マイナーコードが NativeMessaging::REPLY_NOT_AVAILABLE の CORBA NO_RESPONSE 例外）によって第 1 フェーズ後にアンブロックされます。詳細については，256 ページの「非抑制早期リターンモード」を参照してください。

引数	説明
mode	抑制モードを使用するかどうかを指定します。

wait_timeout

```
attribute unsigned long wait_timeout;
```

この属性は，2 フェーズ呼び出しが要求を送信したり，応答をポーリングする際にブロックする最大時間をミリ秒単位で指定します。タイムアウトになると，ネイティブメッセージングは早期リターンを使って呼び出しをアンブロックします。

the_cookie

```
attribute Cookie the_cookie;
```

この属性は，プロキシの型付き受信者での呼び出しの直後に送信される Cookie を指定します。デフォルトでは，Cookie は空です。空でない Cookie を使用すると，reply_recipient は，アプリケーション固有の逆多重化で行うことができる以上のことを実行できます。詳細については，252 ページの「応答受信者における Cookie と応答逆多重化」を参照してください。

request_tag

```
attribute RequestTag request_tag;
```

この属性は，プロキシの型付き受信者での呼び出しの直後に続く要求を一意に識別します。デフォルトでは，タグは最初に空です。また，要求の送信後，空にリセットされます。タ

グが空でない要求は，グループポーリングの対象となります。259 ページの「poll()」と 251 ページの「グループポーリング」を参照してください。

メモ
- 各呼び出し後，Current の request_tag 属性は自動的に空または null にリセットされます。
- 以前にプロキシで別の 2PI によって使用された request_tag を使ってプロキシで 2PI を開始しようとすると，マイナーコードが NativeMessaging::DUPLICATED_REQUEST_TAG の CORBA BAD_INV_ORDER 例外が生成されます。

the_poller

```
readonly attribute object the_poller;
```

この属性は，プロキシの型付き受信者上で行われた呼び出しを通して要求を配信した直後に，ポーラーオブジェクトリファレンスを返します。ポーラーオブジェクトは，2 フェーズ呼び出しの応答ポーリング／プル段階を実行するためにクライアントアプリケーションが使用します。

メモ
- クライアントアプリケーションは，2 フェーズ呼び出し開始時に使用されたのと同じオペレーションを指定されたポーラーオブジェクトに呼び出し，戻り値をポーリングして取得する必要があります。2 フェーズ呼び出し開始時に使用されたのとは異なるポーラー上にオペレーションを呼び出すと，CORBA BAD_OPERATION 例外が生成され，Current の reply_not_available 属性の値が TRUE になります。
- ポーリングオブジェクトは，ネットワーク上の位置透過性を持つ通常の CORBA オブジェクトです。したがって，ネイティブメッセージングでは，2 フェーズ呼び出しの要求送信フェーズと応答ポーリング段階は，必ずしも同じクライアント実行コンテキスト内で, かつ同じ転送接続を使って行われるとは限りません。クライアントアプリケーションは，呼び出しの第 1 フェーズを完了してポーラーオブジェクトを取得した後，まったく別のクライアント実行コンテキスト，別のプロセス内で，別の転送接続を通してポーリングを実行できます。
- 応答ポーリング／プル段階で例外が発生した場合，アプリケーションは Current の reply_not_available 属性を使用して，例外が応答ポーリング／プルのエラーを報告しているのか，デリゲートされた要求の実際の結果が例外であり，その結果を正常にプルしたことを報告しているのかを判別する必要があります。TRUE は，この例外がクライアントとエージェント間のポーリング／プルエラーであることを示します。FALSE は，この例外がデリゲートされた要求の実際の結果であることを示します。
- ポーラーオブジェクト上で実行されるコアオペレーションは，そのポーラーオブジェクト上で保留中になっている 2 フェーズ呼び出しに対して直交します。たとえば，ポーラーで _is_a() または _non_existent() を実行しても，保留中の 2 フェーズ呼び出しで応答をポーリング／プルすることにはならず，リポジトリ ID を比較して，ポーラーオブジェクト自身の存在の有無をチェックするだけです。

reply_not_available

```
readonly attribute boolean reply_not_available;
```

この属性は，プロキシの型付き受信者，応答ポーラー，または応答ホルダーでアンブロックされた呼び出し（通常の戻りまたは例外のいずれか）の結果を次の表に示すように報告します。

Reply_not_available	True	False	True	False
呼び出されたオブジェクト	プロキシの型付き受信者	応答ポーラーまたは応答ホルダー		
通常の戻り，例外なし	2PI 開始（早期リターン）	2PI 完了	（ポーラーのみ）応答不可（早期リターン）	2PI 完了

Reply_not_available	True	False	True	False
RNA 例外（非抑制モード）	2PI 開始（早期リターン）	N/A	（ポーラーのみ）応答不可（早期リターン）	N/A
RNA 以外の例外	2PI 開始エラー	2PI 完了（ターゲットエラー）	ポーリング／プルエラー	2PI 完了（ターゲットエラー）

この表で使用されている語句について，次に説明します。

- **2PI 開始**：プロキシの型付き受信者で実行されたオペレーションの結果が通常の戻りまたは `RNA` 例外（非抑制モード）であり，Current の `reply_not_available` 属性が `TRUE` であるときに報告される結果です。これは，ネイティブメッセージングにおける 2 つの早期リターンモードの 1 つです。デフォルトでは，この開始された 2 フェーズ呼び出しの応答ポーラーは，呼び出し後に Current で使用できます。
- **2PI 開始エラー**：プロキシの型付き受信者で実行されたオペレーションの結果が `RNA` 以外の例外であり，Current の `reply_not_available` 属性が `TRUE` のときに報告される結果です。この結果は，エージェントが 2 フェーズ呼び出しを拒否したか，またはクライアントがエージェントの早期応答メッセージの受信に失敗したことを示します。Current で応答ポーラーを使用することはできません。早期応答メッセージを受信する際の通信エラーによってこの結果が発生した場合，エージェントは，依然として要求をデリゲートし，応答受信者に対するコールバックを生成する場合もあります。
- **2PI 完了**：プロキシの型付き受信者，応答ポーラーまたは応答ホルダーで実行されたオペレーションの結果が通常の戻りまたは CORBA 例外であり，Current の `reply_not_available` 属性が `FALSE` であるときに報告される結果です。オペレーションの結果が `RNA` 以外の例外であり，`reply_not_available` 属性の値が `TRUE` の場合，この例外は，ターゲットにデリゲートされた要求の実際の結果です。
- **応答不可**：応答ポーラーで実行されたオペレーションの結果が通常の戻りまたは `RNA` 例外であり，Current の `reply_not_available` 属性が `TRUE` であるときに報告される結果です。これは，2 つの早期リターンモードの 1 つです。
- **ポーリング／プルエラー**：応答ポーラーまたは応答ホルダーで実行されたオペレーションの結果が `RNA` 以外の例外であり，Current の `reply_not_available` 属性が `TRUE` のときに報告される結果です。この結果は，一致しないオペレーションを呼び出したり，ポーラーがすでに破棄されているなど，応答を取得する際の使い方またはシステムのエラーを示します。
- **該当なし**：該当する結果がありません。発生しません。

ReplyRecipient インターフェース

`ReplyRecipient` オブジェクトは，コールバックモデルで応答結果を受信するためにネイティブメッセージングアプリケーションが実装します。248 ページの「コールバックモデル」と 252 ページの「応答受信者における Cookie と応答逆多重化」のサンプルを参照してください。

reply_available()

```
void
reply_available(
    in object reply_holder,
    in string operation,
    in Cookie the_cookie);
```

このメソッドは，応答を配信する際のリクエストエージェントによるコールバックです。実際の応答結果は，通常の戻りか例外かにかかわらず，`reply_holder` 入力オブジェクトによって保持され，そのオブジェクトに対してコールバックを実行することによって取得できます。例外が `reply_holder` での呼び出しから生成された場合，アプリケーションは，Current

の reply_not_available 属性を使用して，例外がエラーの取得を報告しているのか，デリゲートされた要求の実際の結果を報告しているのかを判別する必要があります。TRUE は，この例外がクライアントとエージェント間の取得エラーであることを示します。FALSE は，この例外がデリゲートされた要求の実際の結果であることを示します。

248 ページの「コールバックモデル」の例を参照してください。

引数	説明
reply_holder	reply_available() メソッドの範囲内で，このオブジェクトリファレンスは，応答ポーラーと同じセマンティクスを保持します。reply_holder での応答取得オペレーションは，reply_available() メソッドの範囲内でのみ実行する必要があります。アプリケーションが reply_available() から戻ると，reply_holder は有効でなくなります。
operation	元のオペレーションシグニチャ。アプリケーションによって粗粒度の逆多重化に使用されます。reply_holder リファレンスで行われる呼び出しは，このパラメータと同じオペレーションシグニチャを持つ必要があります。別のオペレーションを使って reply_holder で呼び出しを行うと，Current の reply_not_available 属性値が TRUE の CORBA BAD_OPERATION 例外が生成されます。
the_cookie	元の要求の Cookie。アプリケーションによって細粒度の逆多重化用に使用されます。

コアオペレーションのセマンティクス

ネイティブメッセージングは，すべての擬似オペレーションを**コアオペレーション**として予約します。コアオペレーションは，次の規則を満たします。

- 常に 1 フェーズ呼び出しで実行されます。コアオペレーションは，処理完了リターンまたは RNA 以外の例外が生成されるまで，常にブロックします。
- プロキシの型付き受信者で呼び出された場合，2　フェーズ呼び出しを開始して実際のターゲットに転送されることはありません。たとえば，プロキシの型付き受信者で _non_existent() を呼び出しても，ターゲットではなく，受信者自身の存在の有無を確認する ping が行われるだけです。
- コアオペレーションは，応答ポーラーまたは応答ホルダー上で保留中の 2 フェーズ呼び出しに対して直交します。たとえば，応答ポーラーまたは応答ホルダーで _is_a() または _non_existent() を呼び出しても，保留中の 2 フェーズ呼び出しの応答結果は取得されず，リポジトリ ID を比較し，ポーラーまたはホルダーオブジェクト自身の存在の有無を確認するだけです。

ネイティブメッセージングの相互運用性仕様

ここで説明する内容は，ネイティブメッセージングアプリケーションの開発者向けではなく，サードパーティのネイティブメッセージングベンダー向けです。

ネイティブメッセージングはネイティブ GIOP を使用

CORBA Messaging などネイティブでないメッセージングでは，OMG GIOP プロトコルは直接のメッセージプロトコルとして使用されません。別の特別なメッセージルーティングプロトコルのためのトンネリングプロトコルとして使用されます。

たとえば，CORBA Messaging で，次の変形オペレーションを呼び出した場合，

```
sendc_foo(<input_parameter_list>);
```

ヘッダーに sendc_foo オペレーションがあり，ペイロードとして <input_parameter_list> を保持するネイティブの OMG GIOP 要求メッセージは生成されません。かわりに，GIOP 要求を通してトンネリングするルーティングメッセージが送信されます。

ネイティブメッセージングは，ネイティブの OMG GIOP をメッセージレベルのプロトコルとして直接使用します。

- エージェント，要求プロキシの型付き受信者，応答ポーラー，応答受信者，応答ホルダーリファレンスなどでのメソッド呼び出しは，呼び出されたオペレーションの正しい名前をヘッダーに持ち，送信されるペイロードとして OMG GIOP で定義されている正しい入力パラメータを持つネイティブの GIOP 要求メッセージを生成します。
- 早期リターンは，RNA 例外，特に，マイナーコードが REPLY_NOT_AVAILABLE の CORBA NO_RESPONSE 例外を含むネイティブの GIOP 応答メッセージです。
- 処理完了リターンは，ターゲットからの <return_value_and_output_parameter_list> または例外をペイロードとして正確に保持するネイティブの GIOP 応答メッセージです。

ネイティブメッセージングのサービスコンテキスト

OMG のセキュリティサービスやトランザクションサービスと同様に，ネイティブメッセージングも，特定のセマンティクな結果を実現するためにサービスコンテキストを使用します。クライアント側のネイティブメッセージングエンジンは，たとえば OMG 準拠の PortableInterceptor で実装されますが，必要なサービスコンテキストを作成して特定の発信要求に追加し，着信応答内の同じ種類のサービスコンテキストから情報を抽出します。

ネイティブメッセージングのサービスコンテキストが使用する context_id は，NativeMessaging::NMService です。context_data は，次のように定義されたカプセル化された NativeMessaging::NMContextData です。

```
module NativeMessaging {
    …

    const IOP::ServiceID NMService = ...

    struct RequestInfo {
        RequestTag request_tag;
        Cookie the_cookie;
        unsigned long wait_timeout;
    };

    union NMContextData switch(short s) {
        case 0: RequestInfo req_info;
        case 1: unsigned long wait_timeout;
        case 2: object the_poller;
        case 3: string replier_name;
    };
};
```

ネイティブメッセージングでのコンテキストデータごとの規定の使用方法を次の表に示します。

送信先または受信元	プロキシの型付き受信者	応答ポーラー	応答ホルダー
要求	req_info	wait_timeout	未定義
標準応答（NO_EXCEPTION）	未定義		
RNA 例外	the_poller	NMService コンテキストなし	N/A
呼び出し元ターゲットからの RNA 以外の例外	replier_name		
エージェント内の RNA 以外の例外	NMService コンテキストなし		

この表で使用されている語句について，次に説明します。

- **req_info**：NMContextData は，コアオペレーション以外の双方向オペレーションがプロキシの型付き受信者に送信するすべての要求に対して規定されます。このコンテキストは，ネイティブメッセージングの Current からの request_tag，Cookie，および wait_timeout を 2 フェーズ呼び出しを開始するための補助パラメータとして保持します。このコンテキストの内容は，リクエストエージェントが要求にタグを付けたり，

Cookie 付きコールバックを配信する，2 フェーズ呼び出しに展開する前に待機する，などの目的で使用します。前節の該当する項目を参照してください。

- **wait_timeout** : NMContextData は，応答ポーラーに送信されるすべての通常の要求（双方向，コア以外）に対して，ポーリングのための補助パラメータとしてネイティブメッセージング Current の `wait_timeout` を使用して，規定されます。内容，つまり `wait_timeout` は，処理完了リターンまたは早期リターンの前に呼び出しをブロックするためにリクエストエージェントが使用します。前節の該当する項目を参照してください。
- **the_poller** : NMContextData は，プロキシの型付き受信者オブジェクトで 2 フェーズ呼び出しを開始する際のすべての成功したリターンに対して規定されます。コンテキストの内容，つまりポーラーリファレンスは，抽出され，ネイティブメッセージング Current の `the_poller` 属性にコピーされます。
- **replier_name** : NMContextData は，要求をデリゲートした結果，正常な戻り値として例外が返された場合に，すべての例外リターンに対して規定されます。このコンテキストは，例外リターンが要求をデリゲートした結果生成されたエラーでない場合は，実行されません。文字列の実際の内容は，空で，将来の拡張のために予約されています。
- **未定義** : この場合，ネイティブメッセージングは NMService コンテキストを使用しません。
- **該当なし** : 適用されません。発生しません。

NativeMessaging タグ付きコンポーネント

`NativeMessaging::TAG_NM_REF` タグが付けられたコンポーネントは，要求プロキシおよびポーラーリファレンスの型付き受信者に埋め込む必要があります。このタグ付きコンポーネントの `component_data` は，Octet 値をカプセル化します。つまり，`component_data` の最初の Octet 値は，バイトオーダーのバイト値であり，2 番めのバイト値は Octet 値です。この Octet 値が `0x01` の場合，リファレンスが要求プロキシの型付き受信者であることを，また `0x02` の場合は，リファレンスがポーラーリファレンスであることを示します。

このコンポーネントを使用して，PortableInterceptor の `send_request()` メソッドは，要求がネイティブメッセージングの要求プロキシの `the_receiver` リファレンス，応答ポーラーなどに送信しているかどうかを判別し，発信要求にサービスコンテキストを追加するかどうか，またどのサービスコンテキストを追加するかを決定します。

Borland ネイティブメッセージングの使用

リクエストエージェントとクライアントモデルの使用

Borland リクエストエージェントの起動

リクエストエージェントサービスを起動するには，コマンド `requestagent` を実行します。このコマンドを `requestagent -?` のように実行すると，使い方に関する情報を表示できます。

Borland リクエストエージェントの URL

ネイティブメッセージングを使用するには，リクエストエージェントをクライアントアプリケーションが認識する必要があります。それには，通常，OMG 準拠の ORB 初期化コマンド引数を使ってクライアント ORB を初期化します。

```
-ORBInitRef RequestAgent=<request_agent_ior_or_url>
```

これにより，クライアントアプリケーションは，ORB からのリクエストエージェントリファレンスを初期サービスとして解決します。次に例を示します。

```
// C++ でのリクエストエージェントリファレンス取得
CORBA::Object_var ref
```

```
      = orb->resolve_initial_references("RequestAgent");
NativeMessaging::RequestAgentEx_var agent
      = NativeMessaging::RequestAgentEx::_narrow(ref);
```

デフォルトでは，リクエストエージェントの URL は次のとおりです。

```
corbaloc::<host>:<port>/RequestAgent
```

ここで, <host> はリクエストエージェントサーバーのホスト名またはドット付きの IP アドレスです。また，<port> は，このサーバーの TCP リスナーポート番号です。デフォルトでは, ネイティブメッセージングのリクエストエージェントは, ポート 5555 を使用します。

Borland ネイティブメッセージングクライアントモデルの使用

C++ での Borland ネイティブメッセージングのクライアント側モデルは，OMG のポータブルインターセプタとして実装され，ネイティブメッセージングクライアントコンポーネントと呼ばれます。ネイティブメッセージングの C++ クライアントコンポーネントは，ネイティブメッセージングクライアントアプリケーション（コールバックオブジェクトを含む）とリンクすることにより（またはアプリケーションにロードすることにより），暗黙的に有効化／無効化されます。

Borland リクエストエージェントの vbroker プロパティ

vbroker.requestagent.maxThreads

要求呼び出しの最大スレッド数を指定します。デフォルト値は 0 で，これは制限がないことを意味します。負数は指定できません。

vbroker.requestagent.maxOutstandingRequests

サービスを受けるために待機する要求の最大キューサイズを指定します。このプロパティは，maxThreads プロパティがゼロ以外の値に設定されている場合にだけ有効になります。デフォルト値は 0 で，これは制限がないことを意味します。負数は指定できません。キューサイズが最大サイズと同じになったときに要求を受信すると，要求は，キューに空きができるまでタイムアウトの間待機します。266 ページの「vbroker.requestagent.blockingTimeout」を参照してください。

vbroker.requestagent.blockingTimeout

要求がキューに追加される前に待機する最大時間をミリ秒単位で指定します。デフォルト値は 0 で，これは待機しないことを意味します。負数は指定できません。値を 0 に設定した場合，要求を受信したときにキューがいっぱいの場合，リクエストエージェントは，CORBA::IMP_LIMIT 例外を生成します。ゼロ以外の値に設定した場合は，要求は指定されたタイムアウトの間待機します。タイムアウト終了後，要求は，キューが空で，作業スレッドが使用可能な場合，ただちに実行されます。また，キューに空きがあり，要求がサービスを受けるまでキューに留まる場合は，待機キューに入れられます。キューにまだ空きがない場合は，リクエストエージェントによって CORBA::IMP_LIMIT 例外が生成されます。

vbroker.requestagent.router.ior

OMG メッセージングルーターの IOR を指定します。デフォルト値は，空の文字列です。

vbroker.requestagent.listener.port

リクエストエージェントが使用する TCP リスナーポートを指定します。デフォルト値は，5555 です。

vbroker.requestagent.requestTimeout

このプロパティは，エージェントがクライアントのために応答結果を保持する最大時間をミリ秒単位で指定します。リクエストエージェントが要求への応答結果を受信したが，クライアントが結果をプルしない，または要求を破棄しない場合，リクエストエージェント

は，このプロパティによって設定される要求タイムアウトが終了すると，要求を（応答結果とともに）破棄します。このプロパティのデフォルト値は，infinity です。これは，エージェントが，クライアントアプリケーションによって（手動または自動で）破棄されるまで，応答結果を保持することを意味します。

CORBA メッセージング との相互運用性

ネイティブメッセージングリクエストエージェントは，OMG の型なしメッセージングルーターと相互運用性があります。特に，リクエストエージェントは，要求を指定されたターゲットに直接送信するのではなく，要求を OMG の型なしルーターを通してルーティングするように設定できます。それには，リクエストエージェントは，266 ページの「vbroker.requestagent.router.ior」プロパティの値に有効な CORBA メッセージングルーターの IOR を設定して起動する必要があります。

以前のバージョンの VisiBroker ネイティブメッセージングからの移行

VisiBroker 6.0 はネイティブメッセージング IDL が変更されており, VBE 5.x ネイティブメッセージングを使って記述されたアプリケーションのソースレベルとバイナリレベルの互換性に影響する可能性があります。

VBE 5.x アプリケーションの開発者が注意する必要がある変更が 2 つあります。1 つは Property 構造体に関するもので，もう 1 つは OctetSeq の定義です。

Property

VisiBroker 5.x では，Property 構造体は次のように定義されていました。

```
module NativeMessaging {
  struct Property {
        string  name;
        any     value;
  };
};
```

VisiBroker 6.0 では，Property は CORBA::NameValuePair の typedef です。次のように指定されています。

```
typedef CORBA::NameValuePair    Property;
typedef CORBA::NameValuePairSeq PropertySeq;
```

したがって，ネイティブメッセージング 5.x アプリケーションを移行する場合は，Property に CORBA::NameValuePair を使用する必要があります。

OctetSeq

VisiBroker 5.x では，OctetSeq は次のように定義されていました。

```
typedef sequence<octet>      OctetSeq;
```

RequestTag と Cookie は次のように定義されていました。

```
typedef OctetSeq             RequestTag;
typedef OctetSeq     Cookie;
```

VisiBroker 6.0 では，この定義は削除され，RequestTag と Cookie は次のように定義されます。

```
typedef CORBA::OctetSeq      RequestTag;
typedef CORBA::OctetSeq      Cookie;
```

この変更によって，ReplyRecipient インターフェースの reply_available() メソッドを次のように変更する必要があります。

```
void reply_available (CORBA::Object_ptr replyHolder,
 const char* operation, const NativeMessaging::OctetSeq& cookie)
```

これを次のように変更します。

```
void reply_available (CORBA::Object_ptr replyHolder,
 const char* operation, const CORBA::OctetSeq& cookie)
```

したがって，ネイティブメッセージング 5.x アプリケーションを移行する場合は，RequestTag と Cookie に CORBA::OctetSeq を使用する必要があります。

これらの変更は手動で行う必要があります。使用できる移行ツールはありません。すべての VisiBroker 5.x ネイティブメッセージングアプリケーションは，VisiBroker 6.0 リクエストエージェントと「ネットワーク経由」の互換性があります。

以前のバージョンの VisiBroker ネイティブメッセージングからの移行

VisiBroker 6.0 はネイティブメッセージング IDL が変更されており, VisiBroker 5.x ネイティブメッセージングを使って記述されたアプリケーションのソースレベルとバイナリレベルの互換性に影響する可能性があります。主な変更点は，Property 構造体に関するものです。VisiBroker 5.x では，この構造体は次のように定義されていました。

```
module NativeMessaging {
  struct Property {
        string  name;
        any     value;
  };
};
```

VisiBroker 6.0 では，Property は CORBA::NameValuePair の typedef です。次のように指定されています。

```
typedef CORBA::NameValuePair     Property;
typedef CORBA::NameValuePairSeq  PropertySeq;
```

したがって，ネイティブメッセージング 5.x アプリケーションを移行する場合は，Property に CORBA::NameValuePair を使用する必要があります。これらの変更は手動で行う必要があります。使用できる移行ツールはありません。すべての VisiBroker 5.x ネイティブメッセージングアプリケーションは，VisiBroker 6.0 リクエストエージェントと「ネットワーク経由」の互換性があります。

第 20 章

オブジェクトアクティベーションデーモン（OAD）の使い方

この章では，オブジェクトアクティベーションデーモン（OAD）の使い方について説明します。

オブジェクトとサーバーの自動アクティブ化

オブジェクトアクティベーションデーモン（OAD）は，VisiBroker によるインプリメンテーションリポジトリのインプリメンテーションです。インプリメンテーションリポジトリは，サーバーがサポートするクラス，インスタンス化されるオブジェクト，およびそれらの ID に関する情報の実行時リポジトリを提供します。通常のインプリメンテーションリポジトリが提供するサービスに加えて，OAD は，クライアントがオブジェクトを参照するときに，インプリメンテーションを自動的にアクティブ化するためにも使用します。オブジェクトインプリメンテーションを OAD に登録すると，この自動アクティブ化機能をオブジェクトに提供できます。

オブジェクトインプリメンテーションは，コマンドラインインターフェース（oadutil）で登録できます。また，OAD の VisiBroker ORB インターフェースも使用できます。279 ページの「OAD の IDL インターフェース」を参照してください。どちらの場合も，リポジトリ ID，オブジェクト名，アクティブ化ポリシー，およびインプリメンテーションを表す実行可能プログラムを指定する必要があります。

メモ　VisiBroker for Java および C++ で生成したサーバーは，VisiBroker OAD でインスタンス化できます。

OAD は独立したプロセスであり，オブジェクトサーバーをオンデマンドでアクティブ化するホストで起動するだけです。

インプリメンテーションリポジトリデータの検索

OAD に登録されたすべてのオブジェクトインプリメンテーションに関するアクティブ化情報は，インプリメンテーションリポジトリに保存されます。デフォルトでは，インプリメンテーションリポジトリデータは，<install_dir>/adm/impl_dir ディレクトリの impl_rep という名前のファイルに保存されます。

サーバーのアクティブ化

OAD は，クライアント要求に応じてサーバーをアクティブ化します。VisiBroker クライアントと VisiBroker 以外の IIOP 準拠クライアントが OAD を介してサーバーをアクティブ化できます。

プロトコルを使用するクライアントは，VisiBroker サーバーのリファレンスの使用時に，そのサーバーをアクティブ化できます。エキスポートされたサーバーのオブジェクトリファレンスは OAD を指し，クライアントは IIOP の規則にしたがって子のサーバーに転送されます。ネーミングサービスなどを介して，サーバーのオブジェクトリファレンスを正しく永続化するには，常に同じポートで OAD を起動する必要があります。たとえば，ポート 16050 で OAD を起動するには，次のように入力します。

```
prompt> oad -VBJprop vbroker.se.iiop_tp.scm.iiop_tp.listener.port=16050
```

メモ　ポート 16000 がデフォルトのポートですが，listener.port プロパティを設定すればこれを変更できます。

OAD の使い方

OAD はオプションの機能です。この機能を使用すると，クライアントがオブジェクトにアクセスしようとしたとき，そのオブジェクトが自動的に起動されるように登録しておくことができます。OAD を起動するには，その前にスマートエージェントを起動する必要があります。詳細については，第 14 章「スマートエージェントの使い方」を参照してください。

OAD の起動

Windows：　OAD を起動するには，次の手順にしたがいます。

- <install_dir>¥bin¥ にある oad.exe を使用します。

  または

- 次のようにコマンドプロンプトに入力します。

  ```
  prompt> oad
  ```

oad コマンドは，次のコマンドライン引数を受け取ります。

オプション	説明
-verbose	詳細モードをオンにします。
-version	このツールのバージョンを表示します。
-path <path>	インプリメンテーションリポジトリを格納するためのプラットフォーム固有ディレクトリを指定します。これは，環境変数で指定した設定を上書きします。
-filename <repository_filename>	インプリメンテーションリポジトリの名前を指定します。指定しない場合のデフォルトは ALL です。この指定は，ユーザーの環境変数設定を上書きします。
-timeout <#_of_seconds>	子サーバーのプロセスが，要求された ORB オブジェクトをアクティブ化するまで，OAD が待機する時間を指定します。デフォルトのタイムアウトは 20 秒です。制限なく待機する場合は，この値を 0 に設定します。子サーバーのプロセスが，要求されたオブジェクトをタイムアウト時間内にアクティブ化しない場合，OAD はその子プロセスを終了し，クライアントは CORBA::NO_IMPLEMENT 例外を受け取ります。より詳細な情報を表示する場合は，verbose オプションをオンにします。
-IOR <IOR_filename>	OAD の文字列化された IOR を保存するファイルの名前を指定します。
-kill	オブジェクトが OAD から完全に登録を解除されると，そのオブジェクトの子プロセスを終了します。

オプション	説明
-no_verify	登録の有効性の検証をオフにします。
-?	コマンドの使い方を表示します。
-readonly	-readonly オプションで OAD を起動すると，登録されているオブジェクトに変更を加えることはできません。オブジェクトを登録または登録解除しようとすると，エラーが返されます。-readonly オプションでは，通常，インプリメンテーションリポジトリに変更を加えた後の変更を防ぐため，readonly モードで OAD を再起動します。

OAD は，Windows サービスとしてインストールされるので，Windows に用意されているサービスマネージャを使って制御できます。

UNIX： OAD を開始するには，次のコマンドを入力します。

```
prompt> oad &
```

OAD ユーティリティの使い方

oadutil コマンドは，VisiBroker システムで使用可能なオブジェクトインプリメンテーションを手動で登録，登録解除，およびリストする手段を提供します。oadutil コマンドは Java 言語で実装されており，コマンドラインインターフェースを使用します。各コマンドにアクセスするには，実行する処理の種類を最初の引数として渡して oadutil コマンドを起動します。

メモ oadutil コマンドを使用するには，ネットワーク上の少なくとも 1 つのホストでオブジェクトアクティベーションデーモンプロセス（oad）を実行しておく必要があります。

oadutil コマンドの構文は次のとおりです。

```
oadutil {list|reg|unreg} [options]
```

このツールのオプションは，list，reg，または unreg のどれを指定するかによって異なります。

インターフェース名をリポジトリ ID に変換する

インターフェース名とリポジトリ ID は，アクティブ化されたオブジェクトが実装するインターフェースの型を表す 2 つの方法です。IDL で定義されたインターフェースには，一意のリポジトリ識別子が割り当てられます。インターフェースリポジトリや OAD との通信，および VisiBroker ORB 自身への呼び出しの多くでは，この文字列で種類を識別します。

OAD にオブジェクトを登録したり，OAD への登録を解除する場合は，oadutil コマンドで，オブジェクトの IDL インターフェース名とオブジェクトのリポジトリ ID のいずれかを指定します。

インターフェース名は，次のようにリポジトリ ID に変換されます。

1. インターフェース名の前に「IDL:」を追加します。
2. 先頭以外のスコープ解決演算子（::）をすべてスラッシュ（/）に置き換えます。
3. インターフェース名の前に「:1.0」を追加します。

   たとえば，次の IDL インターフェース名があるとします。

   ```
   ::Module1::Module2::IntfName
   ```

   これは，次のリポジトリ ID に変換されます。

   ```
   IDL:Module1/Module2/IntfName:1.0
   ```

インターフェース名からリポジトリ ID を生成するデフォルトの動作は，#pragma ID と #pragma プレフィクスのメカニズムでオーバーライドできます。ユーザー定義の IDL ファイル内で #pragma ID のメカニズムを使用して，標準以外のリポジトリ ID を指定した場合

は，上記の変換プロセスが機能しません。その場合は，-r リポジトリ ID 引数で，オブジェクトのリポジトリ ID を指定する必要があります。

C++ のオブジェクトインプリメンテーションで最下位の派生インターフェースのリポジトリ ID を取得するには，すべての CORBA オブジェクトに定義されたメソッド <interface_name>._repository_id() を使用します。

oadutil リストによるオブジェクトの一覧表示

oadutil list ユーティリティを使用して，OAD に登録されたすべての VisiBroker ORB オブジェクトインプリメンテーションを一覧表示できます。各オブジェクトの情報は次のとおりです。

- VisiBroker ORB オブジェクトのインターフェース名
- そのインプリメンテーションが提供するオブジェクトのインスタンス名
- サーバーインプリメンテーションの実行可能プログラムのフルパス名
- VisiBroker ORB オブジェクトのアクティブ化ポリシー（共有または非共有）
- インプリメンテーションが OAD に登録されたときに指定されたリファレンスデータ
- アクティブ化のときにサーバーに渡される引数のリスト
- アクティブ化のときにサーバーに渡される環境変数のリスト

oadutil list コマンドは，OAD に登録されたすべての VisiBroker ORB オブジェクトインプリメンテーションを返します。各 OAD には固有のインプリメンテーションリポジトリデータベースがあり，そこに登録情報が保存されています。

メモ　oadutil list コマンドを使用するには，ネットワーク上の少なくとも 1 つのホストで OAD プロセスを実行しておく必要があります。

oadutil list コマンドの構文は次のとおりです。

```
oadutil list [options]
```

oadutil list コマンドは，次のコマンドライン引数を受け取ります。

オプション	説明
-i <interface name>	特定の IDL インターフェース名のオブジェクトに対するインプリメンテーション情報をリストします。-i，-r，-s，または -poa のうち，指定できるのは一度に 1 つです。 メモ：VisiBroker ORB との通信では，インターフェース名ではなく，常にオブジェクトのリポジトリ ID が参照されます。インターフェース名を指定した場合に実行される変換については，271 ページの「インターフェース名をリポジトリ ID に変換する」を参照してください。
-r <repository id>	特定のリポジトリ ID のインプリメンテーション情報をリストします。リポジトリ ID の指定方法の詳細については，271 ページの「インターフェース名をリポジトリ ID に変換する」を参照してください。-i，-r，-s，または -poa のうち，指定できるのは一度に 1 つです。
-s <service name>	特定のサービス名のインプリメンテーション情報をリストします。-i，-r，-s，または -poa のうち，指定できるのは一度に 1 つです。
-poa <poa_name>	特定の POA 名のインプリメンテーション情報をリストします。-i，-r，-s，または -poa のうち，指定できるのは一度に 1 つです。
-o <object name>	特定のオブジェクト名のインプリメンテーション情報をリストします。コマンドステートメントでインターフェース名またはリポジトリ ID を指定した場合にだけ，このオプションを使用できます。このオプションは，-s または -poa 引数の使用時には使用できません。
-h <OAD host name>	特定のリモートホストで実行されている OAD に登録されているオブジェクトのインプリメンテーション情報を一覧表示します。
-verbose	詳細モードをオンにし，メッセージを標準出力に出力します。

オプション	説明
-version	このツールのバージョンを表示します。
-full	OAD に登録されているすべてのインプリメンテーションの状態を一覧表示します。

次のローカルのリスト要求では，インターフェース名とオブジェクト名を指定しています。

```
oadutil list -i Bank::AccountManager -o BorlandBank
```

次のリモートリスト要求の例では，ホストの IP アドレスを指定しています。

```
oadutil list -h 206.64.15.198
```

oadutil によるオブジェクトの登録

oadutil コマンドでは，コマンドラインまたはスクリプトからオブジェクトインプリメンテーションを登録できます。パラメータは，インターフェース名とオブジェクト名の組み合わせ，サービス名，および POA 名のいずれかと，インプリメンテーションを起動する実行可能プログラムへのパス名です。アクティブ化ポリシーを指定しなかった場合は，デフォルトで共有サーバーポリシーが適用されます。開発およびテスト段階では，インプリメンテーションを記述し，手動でそれを起動することもできます。インプリメンテーションの配布準備が整うと，oadutil だけでインプリメンテーションを OAD に登録できます。

メモ オブジェクトインプリメンテーションの登録時には，オブジェクトインプリメンテーションの構築時と同じオブジェクト名を使用します。OAD に登録できるのは，名前付きオブジェクト（グローバルスコープを持つオブジェクト）だけです。

oadutil reg コマンドの構文は次のとおりです。

```
oadutil reg [options]
```

メモ oadutil reg コマンドを使用するには，ネットワーク上の少なくとも 1 つのホストで oad プロセス（oad）を実行しておく必要があります。

oadutil reg コマンドのオプションは，次のコマンドライン引数を受け取ります。

オプション	必須	説明
-i <interface name>	はい	特定の IDL インターフェース名を指定します。-i，-r，-s，または -poa のうち，指定できるのは一度に 1 つです。リポジトリ ID の指定方法の詳細については，271 ページの「インターフェース名をリポジトリ ID に変換する」を参照してください。
-r <repository id>	はい	特定のリポジトリ ID を指定します。-i，-r，-s，または -poa のうち，指定できるのは一度に 1 つです。
-s <service name>	はい	特定のサービス名を指定します。-i，-r，-s，または -poa のうち，指定できるのは一度に 1 つです。
-poa <poa_name>	はい	オブジェクトインプリメンテーションのかわりに POA を登録するには，このオプションを使用します。-i，-r，-s，または -poa のうち，指定できるのは一度に 1 つです。
-o <object name>	はい	特定のオブジェクトを指定します。コマンドステートメントでインターフェース名またはリポジトリ ID を指定した場合にだけ，このオプションを使用できます。このオプションは，-s または -poa 引数の使用時には使用できません。
-cpp <file name to execute>	はい	-o/-r/-s/-poa 引数に一致するオブジェクトを作成および登録する実行可能ファイルへのフルパスを指定します。-cpp 引数で登録できるアプリケーションは，スタンドアロンの実行可能ファイルだけです。
-java <full class name>	はい	メインルーチンを保持する Java クラスの完全名を指定します。このアプリケーションは，-o/-r/-s/-poa 引数に一致するオブジェクトを作成および登録する必要があります。-java 引数で登録されたクラスは，コマンド vbj <full_classname> で実行されます。
-host <OAD host name>	いいえ	OAD を実行すリモートホストを指定します。
-verbose	いいえ	詳細モードをオンにし，メッセージを標準出力に出力します。
-version	いいえ	このツールのバージョンを表示します。

オプション	必須	説明
-cos_name <CosName>	いいえ	この登録のバインド先になる CosName を指定します。メモ：これはサービスまたは POA の登録では機能しません。
-d <referenceData>	いいえ	アクティブ化のときにサーバーに渡されるリファレンスデータを指定します。
-a arg1 -a arg2	いいえ	子の実行可能ファイルのコマンドライン引数に渡される引数を指定します。引数は，複数の -a（arg）パラメータで渡すことができます。これらの引数の伝達により，子の実行可能ファイルを作成します。
-e env1 -e env2	いいえ	子の実行可能ファイルに渡される環境変数を指定します。引数は，複数の -e（env）パラメータで渡すことができます。これらの引数の伝達により，子の実行可能ファイルを作成します。
-p <shared|unshared>	いいえ	子オブジェクトのアクティブ化ポリシーを指定します。デフォルトポリシーは，SHARED_SERVER です。**Shared**：指定された 1 つのオブジェクトの複数のクライアントが同一のインプリメンテーションを共有します。OAD でアクティブ化されるのは，一度に，1 つのサーバーだけです。**Unshared**：指定された 1 つのインプリメンテーションの 1 つのクライアントだけが，アクティブ化されたサーバーにバインドされます。複数のクライアントが同一のオブジェクトインプリメンテーションにバインドしようとすると，クライアントアプリケーションごとに別個のサーバーがアクティブ化されます。クライアントアプリケーションが接続を切断するか，終了すると，そのサーバーは終了します。

例：リポジトリ ID の指定

次のコマンドは，OAD に VisiBroker プログラム factory を登録します。リポジトリ ID が IDL:ehTest/Factory:1.0（インターフェース名 ehTest::Factory に対応）のオブジェクトに対する要求でこのプログラムは起動します。アクティブ化されるオブジェクトのインスタンス名は ReentrantServer であり，このインスタンス名は，コマンドライン引数として子の実行可能ファイルにも渡されます。このサーバーには，要求側のクライアントが子サーバーへの接続を切断すると終了する非共有ポリシーが設定されています。

```
prompt> oadutil reg -r IDL:ehTest/Factory:1.0 -o ReentrantServer ¥
        -cpp /home/developer/Project1/factory_r -a ReentrantServer ¥
        -p unshared
```

例：IDL インターフェース名の指定

次のコマンドは，OAD に VisiBroker Server クラスを登録します。この例で，指定されたクラスは，リポジトリ ID が IDL:Bank/AccountManager:1.0（インターフェース名の IDL 名 Bank::AccountManager に対応）でインスタンス名が CreditUnion のオブジェクトをアクティブ化します。サーバーは非共有ポリシーで起動され，要求したクライアントがサーバーへの接続を切断すると終了されます。クライアントによって最初に起動される際，サーバーには，環境変数 DEBUG=1 も渡されます。

```
prompt> oadutil reg -i Bank::AccountManager -o CreditUnion \
        -cpp Server -a CreditUnion -p unshared -e DEBUG=1
```

OAD にリモートで登録する

リモートホストの OAD にインプリメンテーションを登録するには，-h 引数を使って oadutil reg を実行します。

次の例は，UNIX シェルから Windows 上の OAD にリモート登録を実行する方法です。oadutil に引数を渡す前に UNIX シェルがバックスラッシュを解釈しないように，二重のバックスラッシュでエスケープする必要があります。

```
prompt> oadutil reg -r IDL:Library:1.0 Harvard ¥
-cpp c:¥¥vbroker¥¥examples¥¥library¥¥libsrv.exe -p shared -h 100.64.15.198
```

スマートエージェントを使用しない OAD の使用

スマートエージェントを使用しないで OAD を使ってサーバーにアクセスするには，vbroker.orb.activationIOR プロパティを使って OAD の IOR を oadutil とサーバーに示します。

たとえば，OAD の IOR が e:/adm ディレクトリ（Windows）にあり，製品に付属する bank_portable サンプル（examples/basic/bank_portable ディレクトリにある）を実行するとします。スマートエージェントを使用しないでこのサーバーにアクセスするには，次の手順にしたがいます。

1 **OAD を起動する：**OAD が参照するクラスパスにサーバーのクラスパスが含まれる必要があります。コマンドは次のとおりです。

```
prompt>start oad -VBJprop vbroker.agent.enableLocator=false -verbose
```

2 **oadutil を使ってサーバーを登録する：**コマンドは次のとおりです。

```
prompt> oadutil -VBJprop vbroker.orb.activationIOR=file:///e:/adm/oadj.ior -VBJprop
     vbroker.agent.enableLocator=false reg -i Bank::AccountManager
 -o BankManager -cpp Server
```

3 **サーバーの IOR を生成する：**サーバーは，起動時に IOR をファイルに書き出します。サーバーが実行中の場合はサーバーを終了し，OAD によるサーバーの起動を実際に確認できるようにします。コマンドは次のとおりです。

```
prompt> Server -Dvbroker.orb.activationIOR=file:///e:/adm/oadj.ior Server
```

4 **クライアントを実行する：**OAD が実行中であることを確認した後，次のコマンドを使用します。

```
prompt> Client -Dvbroker.agent.enableLocator=false
```

ネーミングサービスによる OAD の使用

OAD では，ブートストラップにネーミングサービスを使用できます。前節では，スマートエージェントを使用せず，クライアントはサーバーの IOR ファイルを取得する必要がありました。このブートストラップは，次の手順に示すように，ネーミングサービスをかわりに使って実行することもできます。

1 ネーミングサービスへのリファレンスを指定して，OAD を起動します。ネーミングサービスは，ホスト myhost のポート 1111 で実行しているものとします。

```
prompt>oad -verbose -VBJprop vbroker.orb.initRef=NameService=corbaloc::myhost:1111/
NameService
```

2 サーバーを OAD に登録します。-cos_name パラメータを使用して，このサーバーをネーミングサービスに自動的にバインドすることを OAD に示します。

```
prompt>oadutil -VBJprop vbroker.orb.activationIOR=file:///e:/adm/oadj.ior -VBJprop
  vbroker.agent.enableLocator=false reg -i Bank::AccountManager -o BankManager
  -cos_name simple_test -cpp Server/pre>
```

```
prompt>oadutil -VBJprop vbroker.orb.activationIOR=file:///e:/adm/oadj.ior -VBJprop
  vbroker.agent.enableLocator=false reg -i Bank::AccountManager -o BankManager
  -cos_name simple_test -java Server
```

3 これで，クライアントはネーミングサービスを使ってサーバーのリファレンスを解決および取得できます。Java クライアント用のコードを次に示します。

```
prompt>org.omg.CORBA.Object server=
      rootCtx.resolve(new NameComponent[] {new NameComponent("simple_test","")});
```

-cos_name パラメータを使用しているので，OAD はネーミングサービスでのサーバーのバインディングを自動的に行います。

オブジェクトの複数のインスタンスの区別

インプリメンテーションでは，ReferenceData により，同じオブジェクトの複数のインスタンスを区別できます。リファレンスデータの値は，オブジェクトの作成時にインプリメンテーションによって選択され，オブジェクトの存続期間中は一定です。ReferenceData typedef は，プラットフォーム間と VisiBroker ORB で移植可能です。

作成するオブジェクトのインターフェースを識別するために CORBA 仕様で定義された inf_ptr を，VisiBroker では使用しません。VisiBroker で作成したアプリケーションでは，このパラメータに必ず NULL 値を指定します。

CreationImplDef クラスによるアクティブ化プロパティの設定

CreationImplDef クラスは, OAD が VisiBroker ORB オブジェクトをアクティブ化するために必要なプロパティとして，path_name，activation_policy，args，および env を含んでいます。次のサンプルは，CreationImplDef 構造体を示します。

path_name プロパティは，このオブジェクトを実装する実行可能プログラムの正確なパス名を設定します。activation_policy プロパティは，サーバーのアクティブ化ポリシーを表します。このポリシーはオブジェクトの作成と登録に使用されます。args プロパティと env プロパティは，サーバーに渡すコマンドライン引数と環境設定を表します。

```
module extension {
...
   enum Policy {
      SHARED_SERVER,
      UNSHARED_SERVER
   };
   struct CreationImplDef {
      CORBA::RepositoryId repository_id;
      string                 object_name;
      CORBA::ReferenceData   id;
      string                 path_name;
      Policy                 activation_policy;
      CORBA::StringSequence  args;
      CORBA::StringSequence  env;
   };
...
};
```

ORB インプリメンテーションを動的に変更する

次のサンプルに，オブジェクトの登録を動的に変更するために使用する change_implementation() メソッドを示します。このメソッドを使用して，オブジェクトのアクティブ化ポリシー，パス名，引数，および環境変数を変更できます。

```
module Activation
{
...
   void change_implementation(in extension::CreationImplDef old_info,
         in extension::CreationImplDef new_info)
      raises ( NotRegistered, InvalidPath, IsActive );
...
};
```

注意 change_implementation() メソッドを使用すると，オブジェクトのインプリメンテーション名とオブジェクト名を変更できますが，細心の注意が必要です。理由は，このような変更を行うと，クライアントプログラムは古い名前でオブジェクトを探すことができなくなるからです。

OAD::reg_implementation による OAD 登録

VisiBroker のクライアントアプリケーションでは，手動またはスクリプトで oadutil reg コマンドを使用するかわりに OAD::reg_implementation オペレーションで，1 つ以上のオブジェクトをアクティベーションデーモンに登録できます。このオペレーションを使用すると，オブジェクトインプリメンテーションは OAD と osagent に登録されます。OAD はこの情報をインプリメンテーションリポジトリに保存します。これは，クライアントが対象のオブジェクトにバインドしようとすると，オブジェクトインプリメンテーションを検索してアクティブ化するためです。

```
module Activation {
...
   typedef sequence<ObjectStatus> ObjectStatus List;
...
   typedef sequence<ImplementationStatus> ImplStatusList;
...
   interface OAD {
      // インプリメンテーションを登録します。
      Object reg_implementation(in extension::CreationImplDef impl)
         raises (DuplicateEntry, InvalidPath);
   }
}
```

CreationImplDef 構造体には，OAD に必要なプロパティが格納されます。それらのプロパティは，repository_id，object_name，id，path_name，activation_policy，args，および env です。これらのプロパティの値を設定および照会するためのオペレーションも用意されています。これらの追加プロパティは，OAD が VisiBroker ORB オブジェクトをアクティブ化するためのプロパティです。

```
struct CreationImplDef {
   CORBA::RepositoryId repository_id;
   string object_name;
   CORBA::ReferenceData id;
   string path_name;
   Policy activation_policy;
   CORBA::StringSequence args;
   CORBA::StringSequence env;
};
```

path_name プロパティは，このオブジェクトを実装する実行可能プログラムの正確なパス名を設定します。activation_policy プロパティは，サーバーのアクティブ化ポリシーを表します。args プロパティと env プロパティは，サーバーに渡されるオプションの引数と環境設定を表します。

OAD で渡す引数

オブジェクトインプリメンテーションの起動時に，OAD は，そのインプリメンテーションが OAD に登録されたときに指定された引数をすべて渡します。

オブジェクトの登録解除

オブジェクトから提供されるサービスが使用できなくなるか，一時的に中断された場合は，そのオブジェクトは OAD から登録解除する必要があります。VisiBroker ORB オブジェクトは，登録を解除されると，インプリメンテーションリポジトリから削除されます。また，スマートエージェントのディレクトリからも削除されます。登録解除したオブジェクトは，クライアントプログラムによる検索や使用ができなくなります。また，OAD.change_implementation() メソッドでオブジェクトのインプリメンテーションも変更できなくなります。登録の場合と同様に，登録解除もコマンドラインとプログラムのどちらかで実行できます。

oadutil ツールによるオブジェクトの登録解除

oadutil unreg コマンドでは，OAD に登録した 1 つ以上のオブジェクトインプリメンテーションを登録解除できます。オブジェクトの登録が一度解除されると，クライアントがそのオブジェクトを要求しても，OAD が自動的にそれをアクティブ化することはできなくなります。oadutil unreg で登録解除できるのは，oadutil reg コマンドで登録しておいたオブジェクトだけです。

インターフェース名だけを指定した場合は，そのインターフェースに関連付けられているすべての VisiBroker ORB オブジェクトが登録を解除されます。インターフェース名とオブジェクト名を指定して，特定の VisiBroker ORB オブジェクトだけを登録解除することもできます。オブジェクトを登録解除すると，そのオブジェクトに関連付けられているすべてのプロセスが終了します。

メモ　oadutil リストコマンドを使用するには，ネットワーク上の少なくとも 1 つのホストで oad プロセス（oad）を実行しておく必要があります。

oadutil unreg コマンドの構文は次のとおりです。

```
oadutil unreg [options]
```

oadutil unreg コマンドのオプションは，次のコマンドライン引数を受け取ります。

オプション	必須	説明
-i <interface name>	はい	特定の IDL インターフェース名を指定します。-i，-r，-s，または -poa のうち，指定できるのは一度に 1 つです。リポジトリ ID の指定方法の詳細については，271 ページの「インターフェース名をリポジトリ ID に変換する」を参照してください。
-r <repository id>	はい	特定のリポジトリ ID を指定します。-i，-r，-s，または -poa のうち，指定できるのは一度に 1 つです。
-s <service name>	はい	特定のサービス名を指定します。-i，-r，-s，または -poa のうち，指定できるのは一度に 1 つです。
-o <object name>	はい	特定のオブジェクト名を指定します。コマンドステートメントでインターフェース名またはリポジトリ ID を指定した場合にだけ，このオプションを使用できます。-s 引数または -poa 引数を使用する場合，このオプションは使用できません。
-poa <POA_name>	はい	oadutil reg -poa <POA_name> を使って登録した POA を登録解除します。
-host <host name>	いいえ	OAD が実行されているホストの名前を指定します。
-verbose	いいえ	詳細モードを有効にし，メッセージを標準出力に出力します。
-version	いいえ	このツールのバージョンを表示します。

登録解除のサンプル

oadutil unreg ユーティリティは，次の 3 つの場所にある VisiBroker ORB オブジェクトを登録解除できます。

- オブジェクトアクティベーションデーモン
- インプリメンテーションリポジトリ
- スマートエージェント

次のサンプルは，oadutil unreg コマンドの使い方を示しています。このコマンドは，ローカル OAD から MyBank という名前の Bank::AccountManager のインプリメンテーションを登録解除します。

```
oadutil unreg -i Bank::AccountManager -o MyBank
```

OAD からの登録解除の操作

オブジェクトのインプリメンテーションでは，OAD インターフェースのオペレーションまたは属性の 1 つを使用して，VisiBroker ORB オブジェクトの登録を解除できます。

- unreg_implementation(in CORBA::RepositoryId repId, in string object_name)
- unreg_interface(in CORBA::RepositoryId repId)
- unregister_all()
- attribute boolean destroy_on_unregister()

オペレーション	説明
unreg_implementation()	特定のリポジトリ ID とオブジェクト名でインプリメンテーションの登録を解除するには，このオペレーションを使用します。このオペレーションで，指定されたリポジトリ ID とオブジェクト名を実装しているすべてのプロセスが終了します。
unreg_interface()	特定のリポジトリ ID だけでインプリメンテーションの登録を解除する場合は，このオペレーションを使用します。このオペレーションで，指定されたリポジトリ ID を実装しているすべてのプロセスが終了します。
unregister_all()	すべてのインプリメンテーションを登録解除するには，このオペレーションを使用します。destroyActive が true に設定されていない限り，すべてのアクティブなインプリメンテーションは実行を続けます。下位互換性を保つため，unregister_all() は破棄を行いません。つまり，これは unregister_all_destroy(false) を呼び出すのと同等です。
destroy_on_unregister	インプリメンテーションを登録解除するとき，関連するすべての子プロセスを破棄するには，この属性を使用します。デフォルト値は，false です。

次は OAD の登録解除操作の例です。

```
module Activation {
...
  interface OAD {
      ...
    void unreg_implementation(in CORBA::RepositoryId repId,
      in string object_name)
      raises(NotRegistered);
      ...
  }
}
```

インプリメンテーションリポジトリの内容の表示

oadutil ツールを使用すると，特定のインプリメンテーションリポジトリの内容を一覧表示できます。oadutil ツールは，インプリメンテーションリポジトリ内の各インプリメンテーションについて，すべてのオブジェクトのインスタンス名，実行可能プログラムのパス名，アクティブ化モード，およびリファレンスデータを一覧表示します。実行可能プログラムに渡される引数または環境変数があれば，それらもすべて一覧表示されます。

OAD の IDL インターフェース

OAD は，VisiBroker ORB オブジェクトとして実装されます。これは，OAD にバインドして，そのインターフェースで，登録済みのオブジェクトのステータスを照会するクライアントプログラムを作成するときに使用します。次のサンプルは，OAD の IDL インターフェース仕様です。

```
module Activation
{
   enum state {
      ACTIVE,
      INACTIVE,
      WAITING_FOR_ACTIVATION
   };
   struct ObjectStatus {
      long unique_id;
      State activation_state;
```

```
        Object objRef;
    };
    typedef sequence<ObjectStatus> ObjectStatusList;
    struct ImplementationStatus {
        extension::CreationImplDef impl;
        ObjectStatusList status;
    };
    typedef sequence<ImplementationStatus> ImplStatusList;
    exception DuplicateEntry {};
    exception InvalidPath {};
    exception NotRegistered {};
    exception FailedToExecute {};
    exception NotResponding {};
    exception IsActive {};
    exception Busy {};
    interface OAD {
        Object reg_implementation(   in extension::CreationImplDef impl)
                raises (DuplicateEntry, InvalidPath);
        extension::CreationImplDef get_implementation(
                in CORBA::RepositoryId repId,
                in string object_name)
                raises ( NotRegistered);
        void change_implementation(in extension::CreationImplDef old_info,
                in extension::CreationImplDef new_info)
                 raises (NotRegistered,InvalidPath,IsActive);
        attribute boolean destroy_on_unregister;
        void unreg_implementation(in CORBA::RepositoryId repId,
                in string object_name)
                raises ( NotRegistered );
        void unreg_interface(in CORBA::RepositoryId repId)
                raises ( NotRegistered );
        void unregister_all();
        ImplementationStatus get_status(in CORBA::RepositoryId repId,
                in string object_name)
                raises ( NotRegistered);
        ImplStatusList get_status_interface(in CORBA::RepositoryId repId)
                                        raises (NotRegistered);
        ImplStatusList get_status_all();
};
```

第 21 章

インターフェースリポジトリの使い方

インターフェースリポジトリ（IR）は，CORBA オブジェクトインターフェースの記述を保持します。IR 内のデータは，IDL ファイル内のデータ（モジュール，インターフェース，オペレーション，およびパラメータ）の記述と同じですが，クライアントが実行時にアクセスできるように構成されている点が異なります。クライアントは，開発者がオンラインリファレンスツールを使用したときなどに，このインターフェースリポジトリを参照します。また，クライアントは，動的起動インターフェース（DII）を持つオブジェクトへの呼び出しを準備するときなどに，参照先の任意のオブジェクトのインターフェースを検索します。

この節では，インターフェースリポジトリの作成や，VisiBroker のユーティリティや独自のコードを使ってインターフェースリポジトリにアクセスする方法について説明しています。

インターフェースリポジトリの概要

インターフェースリポジトリ（IR）は，CORBA オブジェクトのインターフェース情報を格納したデータベースと考えられます。クライアントは，実行時にこの情報を使ってインターフェースの記述を取得したり，更新することができます。第 15 章「ロケーションサービスの使い方」に説明する VisiBroker ロケーションサービスが，オブジェクト**インスタンス**を記述するデータを保持するのに対して，IR のデータは，**インターフェース**（型）を記述します。IR に保存されているインターフェースを備えたインスタンスが存在する場合も，存在しない場合もあります。IR 内の情報は 1 つ以上の IDL ファイルの情報と等価ですが，IR の方が，クライアントが実行時に使いやすい形式で情報を表現しています。

また，インターフェースリポジトリを使用するクライアントは，第 22 章「動的起動インターフェースの使い方」に説明する動的起動インターフェース（DII：Dynamic Invocation Interface）を使用する場合があります。このようなクライアントは，インターフェースリポジトリを使って未知のオブジェクトのインターフェース情報を取得し，DII を使用してそのオブジェクトのメソッドを呼び出します。ただし，IR と DII の間に決められた関係があるわけではありません。たとえば，IR を使用して，開発者用の「IDL ブラウザ」ツールを作成できます。そのようなツールで，メソッドの記述をブラウザからエディタにドラッグすることにより，開発者のソースコードにそのメソッドを呼び出すテンプレートを挿入することもできます。このサンプルでは，DII とは関係なく IR が使用されます。

インターフェースリポジトリを作成するには，VisiBroker `irep` プログラムを使用します。このプログラムは IR サーバー（インプリメンテーション）です。インターフェースリポジ

トリを更新したり，それに記入するには，VisiBroker idl2ir プログラムを使用します。また，ユーザー自身が IR クライアントを記述して，インターフェースリポジトリを参照したり，更新することもできます。

インターフェースリポジトリの内容

インターフェースリポジトリはオブジェクトの階層を保持し，それらのオブジェクトのメソッドにより，インターフェースに関する情報が公開されます。インターフェースは，一般にオブジェクトの記述と考えられますが，オブジェクトの集合を使ってインターフェースを記述することは，CORBA 環境では意味のあることです。それは，データベースなどの新しいメカニズムが必要ないためです。

IR が保持できるオブジェクトの種類のサンプルとして，IDL ファイルが IDL モジュール定義を保持し，モジュールがインターフェース定義を保持し，インターフェースがオペレーション（メソッド）定義を保持する場合を例に取ります。これに対応して，インターフェースリポジトリは ModuleDef オブジェクトを保持し，ModuleDef オブジェクトは InterfaceDef オブジェクトを保持し，InterfaceDef オブジェクトは OperationDef オブジェクトを保持します。このように，IR の ModuleDef から，保持している InterfaceDef の情報を取得できます。逆もまた同様で，既知の InterfaceDef から，それがどの ModuleDef に存在するかを知ることができます。例外，属性，valuetype など，その他の IDL 構造のすべてがインターフェースリポジトリ内で表現されます。

インターフェースリポジトリは，タイプコードも保持します。タイプコードは，IDL ファイル内に明示的には一覧表示されませんが，IDL ファイル内で定義または参照されているタイプ（long，string，struct）から自動的に派生します。タイプコードは，CORBA の any 型のインスタンスをエンコードまたはデコードするために使用されます。Any 型は任意のタイプを表し，動的起動インターフェースで使用される共通タイプです。

作成できるインターフェースリポジトリの数

インターフェースリポジトリは，ほかのオブジェクトと同様に，必要な数だけ作成できます。IR の作成または使用に対して，VisiBroker から課される条件はありません。ユーザーのサイトでどのようにインターフェースリポジトリを配布し，それに名前を付けるかは，ユーザーが決定します。たとえば，1 つの中央インターフェースリポジトリに「製品版」オブジェクトのインターフェースをすべて入れ，開発者はテスト用に自分の IR を作成する，という規則を設けることもできます。

メモ　インターフェースリポジトリは書き込み可能であり，アクセスコントロールによって保護されません。クライアントが誤って，または故意に，IR を破棄したり，IR から機密扱いの情報を取得する可能性があります。

すべてのオブジェクトに定義されている _get_interface_def メソッドを使用するには，少なくとも 1 つのインターフェースリポジトリサーバーが実行されている必要があります。そうでないと，ORB が IR 内でインターフェースを検索できません。使用可能なインターフェースリポジトリがない場合，または ORB のバインド先の IR がオブジェクトのインターフェース定義とともにロードされていない場合，_get_interface_def は NO_IMPLEMENT 例外を生成します。

irep を使ったインターフェースリポジトリの作成と表示

VisiBroker インターフェースリポジトリサーバーは，irep という名前で <install_dir>/bin ディレクトリ内に置かれています。irep プログラムは，デーモンとして実行されます。オブジェクトインプリメンテーションと同様に，irep をオブジェクトアクティベーションデーモン（OAD）に登録することができます。oadutil ツールでは，（CORBA::Repository などのインターフェース名ではなく）IDL:org.omg/CORBA/Repository:2.3 などのオブジェクト ID が必要です。

irep を使ったインターフェースリポジトリの作成

インターフェースリポジトリを作成したり，その内容を表示するには，irep プログラムを使用します。irep プログラムで使用する構文は次のとおりです。

```
irep <driver_options> <other_options> <IRepName> [file.idl]
```

次の表で，irep でインターフェースリポジトリを作成するための構文について説明します。

構文	説明
IRepName	インターフェースリポジトリのインスタンス名を指定します。この名前を指定することで，クライアントは，このインターフェースリポジトリのインスタンスにバインドできます。
file.idl	IDL ファイルを指定します。irep は，作成するインターフェースリポジトリにこの IDL ファイルの内容をロードし，終了時に IR の内容を IDL ファイルに保存します。ファイルが指定されていない場合，irep は空のインターフェースリポジトリを作成します。

次の表に，irep 引数の定義を示します。第 4 章「C++ 対応プログラマツール」で定義されているドライバオプションも使用できます。

引数	説明
-D, -define foo[=bar]	プリプロセッサマクロを定義します。値は省略可能です。
-I, -include <dir>	#include の検索対象のディレクトリを追加します。
-P, -no_line_directives	プリプロセッサから #line 指示文を発行しません。デフォルトは off です。
-H, -list_includes	#includ の対象ファイルが見つかると，その名前を表示します。デフォルトは off です。
-C, -retain_comments	プリプロセス後の出力内にコメントを保持します。デフォルトは off です。
-U, -undefine foo	プリプロセッサマクロの定義を解除します。
-[no_]idl_strict	IDL ソースの解釈を OMG 標準に制限します。デフォルトは off です。
-[no_]warn_unrecognized_pragmas	認識できない # プラグマがあった場合に警告を出します。デフォルトは on です。
-[no_]back_compat_mapping	VisiBroker 3.x と互換性があるマッピングを使用します。
-h, -help, -usage, -?	このコマンドの使い方を出力します。
-version	ソフトウェアのバージョン番号を表示します。
-install <service name>	NT サービスとしてインストールします。
-remove <service name>	この NT サービスをアンインストールします。

次のサンプルは，Bank.idl という名前のファイルから TestIR という名前のインターフェースリポジトリを作成する方法を示します。

```
irep TestIR Bank.idl
```

インターフェースリポジトリの内容の表示

インターフェースリポジトリの内容を表示するには，VisiBroker ir2idl ユーティリティまたは VisiBroker コンソールアプリケーションを使用します。ir2idl ユーティリティの構文は次のとおりです。

```
ir2idl [-irep <IRname>]
```

次の表で，irep でインターフェースリポジトリの内容を表示するための構文について説明します。

構文	説明
-irep <IRname>	IRname というインターフェースリポジトリインスタンスにバインドするようにプログラムに指示します。このオプションが指定されない場合は，スマートエージェントから返される任意のインターフェースリポジトリにバインドします。

idl2ir を使ったインターフェースリポジトリの更新

インターフェースリポジトリを更新するには，VisiBroker idl2ir ユーティリティを使用します。このプログラムは IR クライアントです。idl2ir ユーティリティの構文は次のとおりです。

```
idl2ir [arguments] <idl_file_list>
```

次のサンプルは，TestIR インターフェースリポジトリを Bank.idl ファイルからの定義を使って更新する方法です。

```
idl2ir -irep TestIR -replace Bank.idl
```

idl2ir または irep ユーティリティを使ってインターフェースリポジトリ内のエントリを除去することはできません。項目を除去するには，次の手順にしたがいます。

- irep プログラムを終了します。
- irep コマンドラインで指定した IDL ファイルを編集します。
- 更新したファイルで irep を再度開始します。

インターフェースリポジトリには，簡単なトランザクションサービスがあります。指定された IDL ファイルのロードに失敗した場合，インターフェースリポジトリは，自分の内容を元の状態にロールバックします。IDL をロードした後は，インターフェースリポジトリは，以降のトランザクションで使用するために，その状態をコミットします。どのリポジトリでも，ホームディレクトリに <IRname>.rollback ファイルがあり，まだコミットされていない最後のトランザクションの状態が保存されます。

メモ　インターフェースリポジトリ内のすべてのエントリを除去する場合は，その内容を新しい空の IDL ファイルで置き換えます。たとえば，Empty.idl という名前の IDL ファイルを使って次のコマンドを実行します。

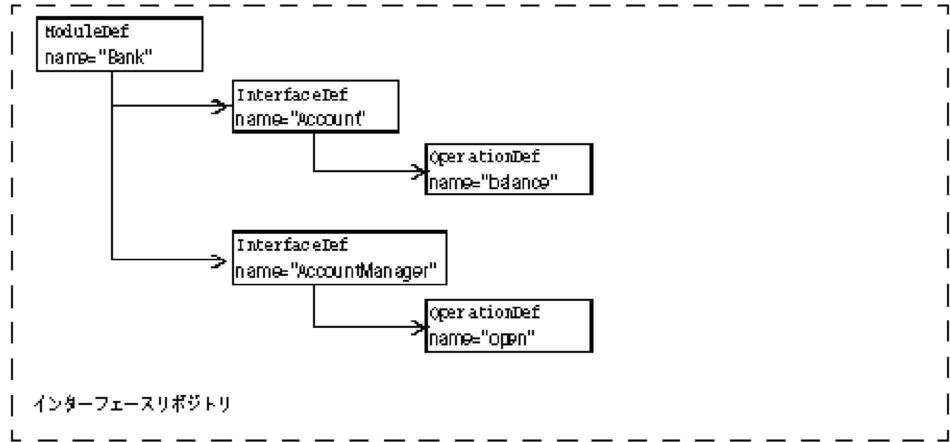
```
idl2ir -irep TestIR -replace Empty.idl
```

インターフェースリポジトリの構造体の概要

インターフェースリポジトリは，その中のオブジェクトを階層的に構成します。この階層は，IDL 仕様で定義されているインターフェースの構造に対応するものです。1 つの IDL モジュール定義内に複数のインターフェース定義があるのと同様に，インターフェースリポジトリ内の一部のオブジェクトは，ほかのオブジェクトを含んでいます。下に示すサンプル IDL ファイルがどのようにインターフェースリポジトリ内のオブジェクトの階層に変換されるかを示します。

```
// Bank.idl
module Bank {
   interface Account {
      float balance();
   };
   interface AccountManager {
      Account open(in string name);
   };
};
```

図 21.1 Bank.idl に対するインターフェースリポジトリのオブジェクト階層



OperationDef オブジェクトは，パラメータと戻り値の型を保持する追加のデータ構造体（インターフェースではない）へのリファレンスを保持します。

インターフェースリポジトリ内のオブジェクトの識別

次の表に，インターフェースリポジトリのオブジェクトを識別および分類するために提供されるオブジェクトを示します。

表 21.1　インターフェースリポジトリのオブジェクトを識別および分類するために使用されるオブジェクト

項目	説明
name	IDL 仕様の中でモジュール，インターフェース，オペレーションなどに割り当てられている識別子に対応する文字列。識別子は必ずしも一意ではありません。
id	IRObject を一意に識別する文字列。RepositoryID は 3 つのコンポーネントからなり，それぞれはコロン（:）デリミタで区切られています。最初のコンポーネントは「IDL:」で，最後のコンポーネントは「:1.0」などのバージョン番号です。2 番めのコンポーネントは，複数の識別子をスラッシュ（/）で区切って並べたものです。その最初の識別子は，通常，一意のプレフィクスです。
def_kind	インターフェースリポジトリオブジェクトのすべての有効なタイプを表す値を定義する列挙体。

インターフェースリポジトリに保存できるオブジェクトの型

下の表に，インターフェースリポジトリ内に保持できるオブジェクトをまとめます。これらのオブジェクトのほとんどは，IDL の構文要素に対応しています。たとえば，StructDef は，IDL の構造体宣言と同じ情報を保持し，InterfaceDef は，IDL のインターフェース宣言と同じ情報を保持します。同様に，IDL のプリミティブ（boolean，long など）宣言と同じ情報を保持する PrimitiveDef までさまざまなタイプがあります。

表 21.2　インターフェースリポジトリに保存できるオブジェクトの型

オブジェクトの型	説明
Repository	ほかのすべてのオブジェクトを保持する最上位レベルのモジュールを表します。
ModuleDef	IDL のモジュール宣言を表します。これは，ModuleDefs，InterfaceDefs，ConstantDefs，AliasDefs，および ExceptionDefs を保持します。また，IDL モジュールで定義できるその他の IDL 構造と等価な IR の構造を保持します。
InterfaceDef	IDL インターフェース宣言を表します。これは，OperationDefs，ExceptionDefs，AliasDefs，ConstantDefs，および AttributeDefs を保持します。
AttributeDef	IDL の属性宣言を表します。

表 21.2　インターフェースリポジトリに保存できるオブジェクトの型（続き）

オブジェクトの型	説明
OperationDef	IDL のオペレーション（メソッド）宣言を表します。これは，インターフェースの 1 つのオペレーションを定義します。この定義は，このオペレーションに必要なパラメータのリスト，戻り値，このオペレーションによって生成される例外のリスト，およびコンテキストのリストからなります。
ConstantDef	IDL の定数宣言を表します。
ExceptionDef	IDL の例外宣言を表します。
ValueDef	valuetype 定義を表します。これは，定数，タイプ，値，例外，オペレーション，および属性の各一覧を保持します。
ValueBoxDef	単純にボックス化された別の IDL 型の valuetype を表します。
ValueMemberDef	valuetype のメンバーを表します。
NativeDef	ネイティブ定義を表します。ユーザーが独自のネイティブを定義することはできません。
StructDef	IDL 構造体宣言を表します。
UnionDef	IDL 共用体宣言を表します。
EnumDef	IDL 列挙体宣言を表します。
AliasDef	IDL typedef 宣言を表します。IR の TypedefDef インターフェースは，StructDefs や UnionDefs に共通するオペレーションを定義するベースインターフェースであることに注意してください。
StringDef	IDL の固定長文字列宣言を表します。
SequenceDef	IDL のシーケンス宣言を表します。
ArrayDef	IDL の配列宣言を表します。
PrimitiveDef	IDL プリミティブ宣言を表します。null，void，long，ushort，ulong，float，double，boolean，char，octet，any，TypeCode，Principal，string，objref，longlong，ulonglong，longdouble，wchar，wstring があります。

継承元のインターフェース

インスタンス化できない（つまり抽象）3 つの IDL インターフェースに，よく使用されるメソッドが定義されています。これらのインターフェースは，IR 内の多くのオブジェクト（上の表を参照）に継承されます。次の表は，これらの広く継承されるインターフェースをまとめたものです。これらのインターフェースのその他のメソッドの詳細については，『*VisiBroker プログラマーズリファレンス*』を参照してください。

表 21.3　多くの IR オブジェクトに継承されるインターフェース

インターフェース	継承する側	主なクエリーメソッド
IRObject	Repository を初めとするすべての IR オブジェクト。	def_kind() は，モジュールやインターフェースなど，IR オブジェクトの定義種類を返します。 destroy() は，IR オブジェクトを破棄します。
Container	ほかの IR オブジェクトを保持できる IR オブジェクト（モジュールやインターフェースなど）。	lookup() は，保持するオブジェクトを名前で検索します。 contents() は，Container のオブジェクトをリストします。 describe_contents() は，Container のオブジェクトを記述します。
Contained	ほかのオブジェクト（Containers）に保持される IR オブジェクト。	name() は，このオブジェクトの名前です。 defined_in() は，オブジェクトを保持する Container です。 describe() は，オブジェクトを記述します。 move () は，別のコンテナにオブジェクトを移動します。

インターフェースリポジトリへのアクセス

クライアントプログラムは，インターフェースリポジトリの IDL インターフェースを使ってインターフェースリポジトリにあるオブジェクトに関する情報を取得できます。クライ

アントプログラムは，Repository にバインドし，次に下に示すメソッドを起動します。このインターフェースの詳細については，『*プログラマーズリファレンス*』を参照してください。

メモ　インターフェースリポジトリを使用するプログラムは，-D_VIS_INCLUDE_IR フラグを付けてコンパイルする必要があります。

```
class CORBA {
   class Repository : public Container {
      . . .
      CORBA::Contained_ptr lookup_id(const char * search_id);
      CORBA::PrimitiveDef_ptr get_primitive(CORBA::PrimitiveKind kind);
      CORBA::StringDef_ptr create_string(CORBA::ULong bound);
      CORBA::SequenceDef_ptr create_sequence(CORBA::ULong bound,
            CORBA::IDLType_ptr element_type);
      CORBA::ArrayDef_ptr create_array(CORBA::ULong length,
            CORBA::IDLType_ptr element_type);
      . . .
   };
   . . .
};
```

インターフェースリポジトリのサンプルプログラム

この節では，アカウントを作成および開く（開き直す）ための単純な AccountManager インターフェースを保持する単純なインターフェースリポジトリの例について説明します。このサンプルコードは，次のディレクトリ内に置かれています。

```
<install_dir>¥vbe¥examples¥ir
```

AccountManager インプリメンテーションは，初期化時に，管理される Account インターフェースのインターフェースリポジトリ定義をブートストラップします。これにより，この特定の Account インプリメンテーションにすでに実装済みの追加オペレーションがクライアントにエクスポーズされます。これで，クライアントは，（IDL に記述されている）既知のオペレーションすべてにアクセスできるようになります。また，インターフェースリポジトリにその他のオペレーションに対するサポートがあるかどうかを確認して，それらのオペレーションを呼び出すことができます。このサンプルでは，インターフェースリポジトリ定義オブジェクトを管理したり，インターフェースリポジトリを使ってリモートオブジェクトの詳細を調べる方法を具体的に示します。

このプログラムをテストするには，次の条件が必要です。

- OSAgent は実行中でなければなりません。詳細については，「スマートエージェントの使い方」の第 14 章「スマートエージェントの使い方」を参照してください。
- インターフェースリポジトリは irep を使って実行中でなければなりません。詳細については，282 ページの「irep を使ったインターフェースリポジトリの作成と表示」を参照してください。
- インターフェースリポジトリの起動時にコマンドラインで指定するか，idl2ir を使用するかのどちらかにより，インターフェースリポジトリに IDL ファイルがロードされていなければなりません。詳細については，284 ページの「idl2ir を使ったインターフェースリポジトリの更新」を参照してください。
- クライアントプログラムを起動します。

IR 内のインターフェースのオペレーションと属性の検索

```
/* PrintIR.C */
#ifndef _VIS_INCLUDE_IR
#define _VIS_INCLUDE_IR
#endif
#include "corba.h"
#include "strvar.h"
int main(int argc, char *argv[]) {
```

```
   try {
      if (argc != 2) {
         cout << "Usage: PrintIR idlName" << endl;
         exit(1);
      }
      CORBA::String_var idlName = (const char *)argv[1];
      CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
      CORBA::Repository_var rep = CORBA::Repository::_bind();
      CORBA::Contained_var contained = rep->lookup(idlName);
      CORBA::InterfaceDef_var intDef = CORBA::InterfaceDef::_narrow(contained);
      if (intDef != CORBA::InterfaceDef::_nil()) {
         CORBA::InterfaceDef::FullInterfaceDescription_var fullDesc = intDef-
>describe_interface();
         cout << "Operations:" << endl;
         for(CORBA::ULong i = 0; i < fullDesc->operations.length(); i++)
            cout << "  " << fullDesc->operations[i].name << endl;
            cout << "Attributes:" << endl;
            for(i = 0; i < fullDesc->attributes.length(); i++)
               cout << "  " << fullDesc->attributes[i].name << endl;
      } else
         cout << "idlName is not an interface: " << idlName << endl;
   } catch (const CORBA::Exception& excep) {
      cerr << "Exception occurred ..." << endl;
      cerr << excep << endl;
      exit(1);
   }
   return 0;
}
```

第 22 章

動的起動インターフェースの使い方

通常のクライアントプログラムの開発者は，自分のコードから起動する CORBA オブジェクトの型を知っており，コンパイラでそれらの型のスタブを生成してコードに挿入します。それに対して，汎用クライアントの開発者には，ユーザーが呼び出すオブジェクトの種類がわかりません。その場合は，動的起動インターフェース（DII : Dynamic Invocation Interface）を使用して，実行時に取得する情報から任意の CORBA オブジェクトの任意のメソッドを起動できるようにクライアントを作成します。

動的起動インターフェースの概要

クライアントプログラムで動的起動インターフェース（DII : Dynamic Invocation Interface）を使用すると，記述された時点では型がわからなかった CORBA オブジェクトのメソッドを起動できます。DII は，デフォルトの静的起動とは対照的です。静的起動では，クライアントが起動する各 CORBA オブジェクトのスタブをコンパイラで生成し，それをクライアントプログラムのソースコードに挿入する必要があります。つまり，静的起動を使用するクライアントは，起動するオブジェクトの型をあらかじめ宣言します。DII を使用するクライアントでは，どのような型のオブジェクトが起動されるかがプログラマにわからないため，そのような宣言はありません。DII の長所はその柔軟性です。DII を使用すると，コンパイルの時点ではインターフェースが存在しなかったオブジェクトを含め，任意のオブジェクトを起動できる汎用のクライアントを作成できます。DII には，次の 2 つの短所があります。

- スタブに相当する作業を行うコードが必要なので，プログラミングが難しくなる。
- 実行時に行う処理が増えるため，起動に時間がかかる。

DII は完全なクライアントインターフェースです。オブジェクトインプリメンテーションという観点から見れば，静的起動と動的起動は同じものです。

DII を使用して，次のようなクライアントを作成できます。

- **スクリプト環境と CORBA オブジェクト間のブリッジまたはアダプタ。**たとえば，スクリプトがブリッジを呼び出して，オブジェクトとメソッドの識別子，およびパラメータ値を渡します。ブリッジは，動的要求を作成して発行し，その結果を受け取り，それをスクリプト環境に返します。このようなブリッジでは，静的起動を使用できません。開発者は，スクリプト環境が起動するオブジェクトの型をあらかじめ知ることができないからです。

- **共通オブジェクトのテスタ。**たとえば，クライアントは，任意のオブジェクト識別子を受け取り，そのインターフェースをインターフェースリポジトリ（第 21 章「インターフェースリポジトリの使い方」を参照）内で検索し，そのメソッドをそれぞれ仮の引数値を使って起動します。やはり，静的起動を使用してこのような汎用のテスタを作成することはできません。

メモ　クライアントは，DII 要求で有効な引数を**渡す必要があります。**有効な引数を渡すことができなかった場合は，サーバーのクラッシュを含む予期しない結果が生じる可能性があります。インターフェースリポジトリを使用して，パラメータ値の型の検査を動的に行うことは可能ですが，その処理は複雑になります。最適なパフォーマンスを得るには，DII を使用するクライアントを呼び出すコード（スクリプトなど）の信頼性を高め，確実に有効な引数を渡すようにします。

重要な DII の概念

実際に動的起動インターフェースが配布される CORBA インターフェースは少数です。さらに，DII では，1 つのタスクを実行するのに複数の方法がある場合がほとんどです。つまり，それぞれの状況において，プログラミングの容易さとパフォーマンスのどちらを優先するかを選択できます。結果として，DII は，CORBA 機能の中でも難解な部類に入ります。この節では，出発点として，DII の主要な概念を説明します。

DII を使用するには，最も基本的なものから順に次の概念を理解する必要があります。

- Request オブジェクト
- Any オブジェクトと Typecode オブジェクト
- Request の送信オプション
- Reply の受信オプション

Request オブジェクトの使用

Request オブジェクトは，1 つの CORBA オブジェクトの 1 つのメソッドの 1 つの起動を表します。同じ CORBA オブジェクトの 2 つのメソッドを呼び出す場合，または 2 つの異なるオブジェクトの同じメソッドを呼び出す場合は, 2 つの Request オブジェクトが必要になります。あるメソッドを呼び出すには, 最初に, ターゲットリファレンス（目的の CORBA オブジェクトを表すオブジェクトリファレンス）が必要です。ターゲットリファレンスを使って Request を作成し，それを引数を使って記入し，Request を送信して返事を待ち，Request から結果を取得します。

Request を作成する方法は 2 つあります。簡単な方法は，ターゲットオブジェクトの _request メソッドを呼び出す方法です。このメソッドは，すべての CORBA オブジェクトによって継承されます。実際には，これはターゲットオブジェクトを呼び出しません。_request には，Request で呼び出すメソッドの IDL 名（「get_balance」など）を渡します。_request を使って作成された Request に引数を追加するには，呼び出すメソッドで必要な引数ごとに Request の add_value メソッドを呼び出します。1 つ以上の Context オブジェクトをターゲットに渡すには，その ctx メソッドを使用してそれらを Request に追加しなければなりません。

すぐには理解しにくいですが，Request の結果の型も result メソッドを使って指定しなければなりません。パフォーマンス上の理由から, VisiBroker ORB どうしで交換されるメッセージには，型情報が含まれません。Request の中でプレースホルダーの結果型を指定することにより，ターゲットオブジェクトによる応答メッセージから結果を正しく抽出するために必要な情報を VisiBroker ORB に提供します。同様に，呼び出すメソッドがユーザー例外を生成する可能性がある場合は，プレースホルダーの例外を Request に追加してから送信する必要があります。

Request オブジェクトを作成する複雑な方法には，ターゲットオブジェクトの _create_request メソッドを起動する方法です。これは再度すべての CORBA オブジェクトを継承する方法です。このメソッドには複数の引数があります。それらの引数は，新規の Request の引数に格納され，また結果の型を指定し，返されるユーザー例外があれば，そ

のユーザー例外を指定します。_create_request メソッドを使用するには，そのメソッドの引数となるコンポーネントをあらかじめ作成しておく必要があります。_create_request メソッドの潜在的な長所はパフォーマンスにあります。複数のターゲットオブジェクトにある同じメソッドを呼び出す場合は，複数回の _create_request の呼び出しの中で，そのメソッドの引数コンポーネントを再利用できます。

メモ　_create_request メソッドには，オーバーロードされた 2 つの形式があります。一方には ContextList パラメータと ExceptionList パラメータがあり，もう一方にはどちらもありません。1 つ以上の Context オブジェクトを渡してメソッドを呼び出すか，呼び出すメソッドが 1 つ以上のユーザー例外を生成する可能性がある場合，またはその両方の場合は，追加パラメータを持つ方の _create_request メソッドを使用する必要があります。

Any 型を使った引数のカプセル化

ターゲットメソッドの引数，結果，および例外は，Any という特殊なオブジェクトでそれぞれ指定されます。Any は，すべての型の引数をカプセル化する汎用オブジェクトです。Any は，IDL で記述できるどの型でも保持できます。引数を Request に Any として指定すると，コンパイラが型不一致を指摘することもなく，Request が任意の引数型と値を保持できます。結果と例外についても同様です。

Any は TypeCode と値で構成されます。値は値そのものであり，TypeCode は値の中のビット列を解釈する方法，つまりその値の型を記述するオブジェクトです。long と Object などの単純 IDL 型に対する単純な TypeCode 定数は，idl2cpp コンパイラによって生成されるヘッダーファイルに組み込まれます。structs，unions，typedefs などの IDL 構造型に対する TypeCode は構築する必要があります。これらが記述する型は再帰的に記述される場合があるので，そのような TypeCode も再帰的になる可能性があります。

ある struct が 1 つずつの long と string で構成されるとします。この struct の TypeCode には，long の TypeCode と string の TypeCode が含まれます。idl2cpp コンパイラは，-type_code_info オプション付きで呼び出されると，IDL ファイルの構造型に対する TypeCode を生成します。ただし，DII を使用している場合は，実行時に TypeCode を取得する必要があります。TypeCode は，実行時にインターフェースリポジトリから取得できます（第 21 章「インターフェースリポジトリの使い方」を参照）。または，ORB::create_struct_tc や ORB::create_exception_tc を呼び出して，VisiBroker ORB に TypeCode を作成するように要求することもできます。

_create_request メソッドを使用する場合，NVList と呼ばれる別の特殊なオブジェクトに，Any 型にカプセル化されたターゲットメソッドの引数を格納する必要があります。どのように Request を作成しても，その結果は NVList としてエンコードされます。この段落で引数について説明している内容は，結果にも同様に当てはまります。「NV」は名前付きの値（named value）という意味です。NVList は項目の数と複数の項目からなり，各項目は名前，値，およびフラグを持ちます。名前は引数の名前，値はその引数をカプセルしている Any，またフラグは引数の IDL モード（in または out など）を表します。Request の結果は，1 つの名前付き値として表されます。

要求の送信オプション

Request を作成し，引数，結果の型，および例外の型を設定したら，これをターゲットオブジェクトに送信します。Request を送信する方法はいくつかあります。

- 最も簡単な方法は，Request の invoke メソッドを呼び出すことです。このメソッドは，応答メッセージを受信するまでブロックします。
- ブロックを行わない方法として，少し複雑な Request の send_deferred メソッドがあります。これは，スレッドを使って並行処理を行う方法のかわりになります。多くのオペレーティングシステムでは，send_deferred メソッドの方がスレッドを生成するより経済的です。
- send_deferred メソッドを使用する目的が，複数のターゲットオブジェクトを並行して呼び出すことである場合，そのかわりとして ORB オブジェクトの send_multiple_requests_deferred メソッドを使用できます。このメソッドは，Request オブジェクトのシーケンスを受け取ります。

- ターゲットメソッドが IDL で oneway と定義されている場合にだけ，Request の send_oneway メソッドを使用します。
- VisiBroker ORB の send_multiple_requests_oneway メソッドを使用すると，複数の oneway メソッドを並行して呼び出すことができます。

応答の受信オプション

invoke メソッドを呼び出して Request を送信した場合，結果を取得する方法は 1 つだけです。Request オブジェクトの env メソッドを使用して，例外があるかどうかをテストします。例外がなければ，result メソッドを使用して，Request から NamedValue を抽出します。send_oneway メソッドを使用した場合，結果はありません。send_deferred メソッドを使用した場合は，そのオペレーションが完了したかどうかを定期的にチェックできます。それには，Request の poll_response メソッドを呼び出し，応答が受信されたかどうかを示すコードを取得します。しばらくポーリングした後，遅延送信の完了を待ちながらブロックする場合は，Request の get_response を使用します。

send_multiple_requests_deferred メソッドで Request を送信した場合，その Request の get_response メソッドを呼び出すことで，特定の Request が完了したかどうかわかります。未処理の Request が完了したかどうかを検出するには，VisiBroker ORB の get_next_response メソッドを使用します。ブロックすることなく，これと同じ処理を行うには，VisiBroker ORB の poll_next_response メソッドを使用します。

オブジェクトのオペレーションを動的に呼び出すための手順

クライアントが DII を使用する場合の手順をまとめると，次のようになります。

1. idl コンパイラに -type_code_info オプションを渡して，IDL のインターフェースと型に対するタイプコードが生成されるようにする。
2. 使用するターゲットオブジェクトへの共通リファレンスを取得する。
3. Request オブジェクトを作成する。
4. 要求のパラメータおよび返される結果を初期化する。
5. 要求を呼び出し，結果を待つ。
6. 結果を取得する。

DII を使用するサンプルプログラム

次のディレクトリに，DII の使い方を紹介するいくつかのサンプルプログラムが用意されています。

```
<install_dir>/examples/vbe/bank_dynamic
```

この節では，これらのサンプルプログラムを使用して，DII の概念を説明します。

これらのサンプルプログラムを VIS_INCLUDE_IR フラグ付きでコンパイルし，タイプコード生成オプションを追加します。

共通オブジェクトリファレンスの取得

DII を使用する場合，クライアントプログラムは，コンパイル時にターゲットオブジェクトのクラス定義がわからない場合があるので，従来のバインドメカニズムを使ってターゲットオブジェクトへのリファレンスを取得する必要はありません。

次のサンプルコードは，クライアントプログラムが VisiBroker ORB オブジェクトから提供される bind メソッドを使用し，オブジェクトの名前を指定してオブジェクトにバインドする方法を示します。このメソッドは共通 CORBA::Object を戻します。

```
. . .
CORBA::Object_var account;
try {
   // ORB を初期化します。
   CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
} catch (const CORBA::Exception& e)
   cout << "Failure during ORB_init" << endl;
   cout << e << endl;
}
. . .
try {
   // Account インターフェースをサポートするオブジェクトにバインドするように ORB に要求
します。
   account = orb->bind("IDL:Account:1.0");
} catch (const CORBA::Exception& excep)
   cout << "Error binding to account" << endl;
   cout << excep << endl;
}
cout << "Bound to account object" << endl;
. . .
```

要求の作成と初期化

クライアントプログラムがオブジェクトのメソッドを呼び出すと，そのメソッドの呼び出しを表す Request オブジェクトが作成されます。Request オブジェクトがバッファに書き込まれるか，または**マーシャリング**され，そのオブジェクトのインプリメンテーションに送信されます。クライアントプログラムがクライアントスタブを使用する場合，この処理は透過的に行われます。DII を使用する場合は，クライアントプログラムが Request オブジェクトを自分で作成して送信する必要があります。

メモ　このクラスにコンストラクタはありません。Object の _request メソッドまたは Object の _create_request メソッドを使用して，Request オブジェクトを作成します。

Request クラス

次のサンプルコードは Request クラスを示します。Request を作成するために使用されるオブジェクトリファレンスから，要求の target が暗黙的に設定されます。Request を作成する際は，operation の名前を指定する必要があります。

```
class Request {
   public:
      CORBA::Object_ptr target() const;
      const char* operation() const;
      CORBA::NVList_ptr arguments();
      CORBA::NamedValue_ptr result();
      CORBA::Environment_ptr env();
      void    ctx(CORBA::Context_ptr ctx);
      CORBA::Context_ptr ctx() const;
      CORBA::Status invoke();
      CORBA::Status send_oneway();
      CORBA::Status send_deferred();
      CORBA::Status get_response();
      CORBA::Status poll_response();
      . . .
   };
};
```

DII 要求を作成および初期化する方法

オブジェクトへのバインドを発行し，オブジェクトリファレンスを取得したら，2 つのメソッドのどちらかを使って Request オブジェクトを作成できます。

次のサンプルは，CORBA::Object クラスから提供されるメソッドを示します。

```
class Object {
   . . .
   CORBA::Request_ptr _request(Identifier operation);
   CORBA::Status _create_request(
      CORBA::Context_ptr ctx,
      const char *operation,
      CORBA::NVList_ptr arg_list,
      CORBA::NamedValue_ptr result,
      CORBA::Request_ptr request,
      CORBA::Flags req_flags);
   CORBA::Status _create_request(
      CORBA::Context_ptr ctx,
      const char *operation,
      CORBA::NVList_ptr arg_list,
      CORBA::NamedValue_ptr result,
      CORBA::ExceptionList_ptr eList,
      CORBA::ContextList_ptr ctxList,
      CORBA::Request_out request,
      CORBA::Flags req_flags);
   . . .
};
```

create_request メソッドの使い方

_create_request メソッドを使って Request オブジェクトを作成し，Context，オペレーション名，渡す引数リスト，および結果を初期化できます。オプションで，要求に ContextList を設定できます。これは，要求の IDL で定義される属性に対応します。このオペレーションのために作成された Request オブジェクトを返します。

_request メソッドの使い方

294 ページの「Request オブジェクトを作成するサンプルコード」は，_request メソッドを使用し，オペレーション名だけを指定して Request オブジェクトを作成する方法を示します。浮動要求を作成した後に，add_in_arg メソッドを呼び出して入力パラメータのアカウント名を追加します。この結果型は，set_return_type メソッドの呼び出しを介してオブジェクトリファレンス型として初期化されます。この呼び出しの後，result メソッドの呼び出しにより，戻り値が抽出されます。Account Manager インスタンスの別のメソッドを呼び出す場合も，パラメータと戻り値の型が異なるだけで，同じ手順が繰り返されます。

Any オブジェクトである req は，目的のアカウント name で初期化され，要求の引数リストに入力引数として追加されます。要求の初期化の最後の手順は，result 値を設定して float を受け取ることです。

Request オブジェクトを作成するサンプルコード

Request オブジェクトのオペレーション，引数，および結果に関連付けられたすべてのメモリは，そのオブジェクトが所有権を持ち続けるので，それらを解放しようとしてはなりません。次のサンプルコードは，Request オブジェクトを作成する例です。

```
. . .
CORBA::NamedValue_ptr result;
CORBA::Any_ptr resultAny;
CORBA::Request_var req;
```

```
CORBA::Any customer;
. . .
try {
   req = account->_request("balance");

   // 要求の引数を作成します。
   customer <<= (const char *) name;
   CORBA::NVList_ptr arguments = req->arguments();
   arguments->add_value("customer", customer, CORBA::ARG_IN);
   // 結果を設定します。
   result = req->result();
   resultAny = result->value();
   resultAny->replace(CORBA::_tc_float, &result);
} catch(CORBA::Exception& excep) {
. . .
```

要求のコンテキストの設定

サンプルプログラムでは使用されていませんが，Context オブジェクトを使用して，プロパティのリストを含めて NamedValue オブジェクトとして保存することができます。これはオブジェクトインプリメンテーションに Request の一部として渡されます。オブジェクトインプリメンテーションに自動的に伝えられる情報を表します。

```
class Context {
   public:

      const char *context_name() const;
      CORBA::Context_ptr         parent();
      CORBA::Status create_child(const char *name, CORBA::Context_ptr&);
      CORBA::Status set_one_value(const char *name, const CORBA::Any&);
      CORBA::Status set_values(CORBA::NVList_ptr);
      CORBA::Status delete_values(const char *name);
      CORBA::Status get_values(
            const char *start_scope,
            CORBA::Flags,
            const char *name,
            CORBA::NVList_ptr&) const;
};
```

要求に引数を設定する方法

Request の引数は NVList オブジェクトで表され，名前／値のペアを NamedValue オブジェクトとして保存します。arguments メソッドを使用すると，このリストへのポインタを取得できます。次に，このポインタを使って各引数の名前と値を設定します。

メモ　常に引数は Request を送信する前に初期化します。引数を初期化しないと，マーシャリングエラーが発生し，サーバーが停止してしまう可能性があります。

NVList クラスを使用して，引数リストを実装する

このクラスは，メソッド呼び出しの引数を表す NamedValue オブジェクトのリストを実装します。リスト内のオブジェクトを追加，削除，および照会するためのメソッドが用意されています。次のサンプルコードは，NVList クラスを示します。

```
class NVList {
   public:
      . . .
      CORBA::Long count() const;
      CORBA::NamedValue_ptr add(Flags);
      CORBA::NamedValue_ptr add_item(const char *name, CORBA::Flags flags);
      CORBA::NamedValue_ptr add_value(
            const char *name,
            const CORBA::Any *any,
            CORBA::Flags flags);
```

```
        CORBA::NamedValue_ptr add_item_consume(char *name, CORBA::Flags flags);
        CORBA::NamedValue_ptr add_value_consume(
              char *name,
              CORBA::Any *any,
              CORBA::Flags flags);
        CORBA::NamedValue_ptr item(CORBA::Long index);
        CORBA::Status remove(CORBA::Long index);
        . . .
};
```

NamedValue クラスを使用して，入力引数と出力引数を設定する

このクラスは，メソッド呼び出し要求用の入力と出力の引数の両方を表す名前／値ペアを実装します。NamedValue クラスは，クライアントプログラムに返される要求の結果を表すためにも使用されます。name プロパティは単純な文字列であり，value プロパティは Any クラスによって表されます。次のサンプルコードは，NamedValue クラスを示します。

```
class NamedValue {
   public:
      const char *name() const;
      CORBA::Any *value() const;
      CORBA::Flags flags() const;
};
```

次の表は，NamedValue クラスのメソッドの説明です。

表 22.1　NamedValue のメソッド

メソッド	説明
name	項目の名前へのポインタを返します。このポインタを使って名前を初期化できます。
value	項目の値を表す Any オブジェクトへのポインタを返します。このポインタを使用して，値を初期化できます。詳細については，296 ページの「Any クラスを使ってタイプセーフに引数を渡す」を参照してください。
flags	この項目が入力引数，出力引数，または入出力引数のいずれであるかを示します。この項目が入出力引数の場合は，フラグを指定することにより，VisiBroker ORB が引数のコピーを作成し，呼び出し元のメモリを書き換えないように指示できます。フラグは ARG_IN，ARG_OUT，ARG_INOUT です。

Any クラスを使ってタイプセーフに引数を渡す

このクラスは，IDL で指定された型を保持し，これをタイプセーフな方法で渡すために使用されます。

このクラスのオブジェクトには，それが保持するオブジェクトの型およびそのオブジェクトへのポインタを定義する TypeCode へのポインタがあります。オブジェクトを構築，コピー，および解放するメソッドのほか，そのオブジェクトの値と型を初期化および照会するためのメソッドが用意されています。また，ストリームからオブジェクトを読み込んだり，ストリームにオブジェクトを書き込むためのストリーム演算子メソッドも用意されています。次のサンプルコードは，このクラスの定義例です。

```
class Any {
   public:
      . . .
      CORBA_TypeCode_ptr type();
      void type(CORBA_TypeCode_ptr tc);
      const void *value() const;
      static CORBA::Any_ptr _nil();
      static CORBA::Any_ptr _duplicate(CORBA::Any *ptr);
      static void _release(CORBA::Any *ptr);
      . . .
}
```

TypeCode クラスを使用して，引数または属性の型を表す

このクラスは，引数や属性の型を表すために，インターフェースリポジトリおよび IDL コンパイラによって使用されます。Request オブジェクトで引数の型を指定するために，Any クラスとともに TypeCode オブジェクトも使用されます。

TypeCode オブジェクトには，kind とパラメータリストプロパティがあります。次のサンプルコードは，TypeCode クラスを示します。

次の表に，TypeCode オブジェクトの種類とパラメータを示します。

表 22.2　TypeCode の種類とパラメータ

種類	パラメータリスト
tk_abstract_interface	repository_id, interface_name
tk_alias	repository_id, alias_name, TypeCode
tk_any	なし
tk_array	length, TypeCode
tk_boolean	なし
tk_char	なし
tk_double	なし
tk_enum	repository_id, enum-name, enum-id^1, enum-id^2, ... enum-id^n
tk_except	repository_id, exception_name, StructMembers
tk_fixed	digits, scale
tk_float	なし
tk_long	なし
tk_longdouble	なし
tk_longlong	なし
tk_native	id, name
tk_null	なし
tk_objref	repository_id, interface_id
tk_octet	なし
tk_Principal	なし
tk_sequence	TypeCode, maxlen
tk_short	なし
tk_string	maxlen-integer
tk_struct	repository_id, struct-name, {member^1, TypeCode^1}, {member^n, TypeCode^n}
tk_TypeCode	なし
tk_ulong	なし
tk_ulonglong	なし
tk_union	repository_id, union-name, switch TypeCode, {label-value^1, member-name^1, TypeCode^1}, {label-value^n, member-name^n, TypeCode^n}
tk_ushort	なし
tk_value	repository_id, value_name, boxType
tk_value_box	repository_id, value_name, typeModifier, concreteBase, members
tk_void	なし
tk_wchar	なし
tk_wstring	なし

TypeCode クラス：

```
class _VISEXPORT CORBA_TypeCode {
  public:
      . . .
    // すべての CORBA_TypeCode の種類について
    CORBA::Boolean equal(CORBA_TypeCode_ptr tc) const;
    CORBA::Boolean equivalent(CORBA_TypeCode_ptr tc) const;
    CORBA_TypeCode_ptr get_compact_typecode() const;
```

```
        CORBA::TCKind kind() const // . . .
        // tk_objref, tk_struct, tk_union, tk_enum, tk_alias, および tk_except について
        virtual const char* id() const; // raises(BadKind);
        virtual const char *name() const; // raises(BadKind);
        // tk_struct, tk_union, tk_enum, および tk_except について
        virtual CORBA::ULong member_count() const;
              // raises((BadKind));
        virtual const char *member_name(CORBA::ULong index) const;
              // raises((BadKind, Bounds));
        // tk_struct, tk_union, および tk_except について
        virtual CORBA_TypeCode_ptr member_type(CORBA::ULong index) const;
              // raises((BadKind, Bounds));
        // tk_union について
        virtual CORBA::Any_ptr member_label(CORBA::ULong index) const;
              // raises((BadKind, Bounds));
        virtual CORBA_TypeCode_ptr discriminator_type() const;
              // raises((BadKind));
        virtual CORBA::Long default_index() const;
              // raises((BadKind));
        // tk_string, tk_sequence, および tk_array について
        virtual CORBA::ULong length() const;
              // raises((BadKind));
        // tk_sequence, tk_array, および tk_alias について
        virtual CORBA_TypeCode_ptr content_type() const;
              // raises((BadKind));
        // tk_fixed について
        virtual CORBA::UShort fixed_digits() const;
              // raises (BadKind)
        virtual CORBA::Short fixed_scale() const;
              // raises (BadKind)
        // tk_value について
        virtual CORBA::Visibility
           member_visibility(CORBA::ULong index) const;
              // raises(BadKind, Bounds);
        virtual CORBA::ValueModifier type_modifier() const;
              // raises(BadKind);
        virtual CORBA::TypeCode_ptr concrete_base_type() const;
              // raises(BadKind);
    };
```

DII 要求の送信と結果の受信

Request クラスは，293 ページの「要求の作成と初期化」で説明するように，正しく初期化されると複数の要求送信メソッドを提供します。

要求を呼び出す

要求を送信する最も簡単な方法は，要求の invoke メソッドを呼び出すことです。このメソッドは，要求を送信した後，応答を待ってからクライアントプログラムに戻ります。return_value メソッドは，戻り値を表す Any オブジェクトへのポインタを返します。次のサンプルコードは，invoke メソッドを使った要求の送信方法を示します。

```
try {
   . . .
   // Account オブジェクトに送信する要求を作成します。
   request = account->_request("balance");
   // 結果の型を設定します。
   request->set_return_type(CORBA::_tc_float);
   // Account オブジェクトへの要求を実行します。
 request->invoke();
   // 残高を取得します。
   CORBA::Float balance;
```

```
    CORBA::Any& balance_result = request->return_value();
    balance_result >>= balance;
    // 残高を出力します。
    cout << "The balance in " << name << "'s account is $" << balance << endl;
} catch(const CORBA::Exception& e) {
    cerr << e << endl;
    return 1;
}
return 0;
. . .
```

send_deferred メソッドを使用して，遅延 DII 要求を送信する

オペレーションリクエストを送信するメソッドには，ブロックを行わない send_deferred メソッドもあります。クライアントは，このメソッドを使って要求を送信した後，poll_response　メソッドを使用して，有効な応答があるかどうかを判定できます。get_response メソッドは，応答を受信するまでブロックします。次のコードに，これらのメソッドの使い方を示します。次のサンプルは，send_deferred メソッドと poll_response メソッドを使用して，遅延 DII 要求を送信する方法を示します。

```
. . .
try {
    // マネージャオブジェクトに送信する要求を作成します。
    CORBA::Request_var request = manager->_request("open");
    // 要求の引数を作成します。
    CORBA::Any customer;
    customer <<= (const char *) name;
    CORBA::NVList_ptr arguments = request->arguments();
    arguments->add_value( "name" , customer, CORBA::ARG_IN );
    // 結果の型を設定します。
    request->set_return_type(CORBA::_tc_Object);
    // 新しい口座の作成には多少時間がかかります。
    // マネージャオブジェクトへの遅延要求を実行します。
  request->send_deferred();
    VISPortable::vsleep(1);
    while (!request->poll_response()) {
        cout << "  Waiting for response..." << endl;
        VISPortable::vsleep(1);  // 1 秒間隔でポーリングします。
    }
  request->get_response();
    // 戻り値を取得します。
    CORBA::Object_var account;
    CORBA::Any& open_result = request->return_value();
    open_result >>= CORBA::Any::to_object(account.out());
. . .
}
```

send_oneway メソッドを使用して，非同期 DII 要求を送信する

非同期要求を送信するには，send_oneway メソッドを使用します。一方向要求には，オブジェクトインプリメンテーションからクライアントに返される応答がありません。

複数の要求を送信する

DII Request オブジェクトのシーケンスは，Request オブジェクトの配列を使って作成できます。要求シーケンスは，VisiBroker ORB メソッド，send_multiple_requests_oneway または send_multiple_requests_deferred を使って送信できます。要求のシーケンスを一方向要求として送信した場合，どの要求にもサーバーからの応答はありません。

次のサンプルコードは，2 つの要求を作成し，それらを使って要求のシーケンスを作成する方法を示します。シーケンスは次に，send_multiple_requests_deferred メソッドを使って送信されます。

```
. . .
// 残高への要求を作成します。
try {
   req1 = account->_request("balance");
   // 要求の引数を作成します。
   customer1 <<= (const char *) "Happy";
   CORBA::NVList_ptr arguments = req1->arguments();
   arguments->add_value("customer", customer1, CORBA::ARG_IN);
   // 結果を設定します。
   . . .
} catch(const CORBA::Exception& excep) {
   cout << "Error while creating request" << endl;
   cout << excep << endl;
}
// slowBalance への要求 2 を作成します。
try {
   req2 = account->_request("slowBalance");
   // 要求の引数を作成します。
   customer2 <<= (const char *) "Sleepy";
   CORBA::NVList_ptr arguments = req2->arguments();
   arguments->add_value("customer", customer2, CORBA::ARG_IN);
   // 結果を設定します。
   . . .
} catch(const CORBA::Exception& excep) {
   cout << "Error while creating request" << endl;
   cout << excep << endl;
}
// 要求シーケンスを作成します。
CORBA::Request_ptr reqs[2];
reqs[0] = (CORBA::Request*) req1;
reqs[1] = (CORBA::Request*) req2;
CORBA::RequestSeq reqseq((CORBA::ULong)2, 2, (CORBA::Request_ptr *) reqs);
// 要求を送信します。
try {
   orb->send_multiple_requests_deferred(reqseq);
   cout << "Send multiple deferred calls are made..." << endl;
} catch(const CORBA::Exception& excep) {
. . .
```

複数の要求を受信する

send_multiple_requests_deferred を使って要求のシーケンスが送信した場合，サーバーから各要求に送信される応答を受信するには，poll_next_response メソッドと get_next_response メソッドを使用します。

VisiBroker ORB メソッド poll_next_response は，サーバーから応答を受信したかどうかを判定するために使用されます。少なくとも 1 つの応答が使用可能である場合，このメソッドは true を返します。使用可能な応答がない場合，このメソッドは false を返します。

VisiBroker ORB の get_next_response メソッドは，応答の受信に使用されます。使用可能な応答がない場合，このメソッドは，応答を受信するまでブロックします。クライアントプログラムがブロックするのが不都合な場合は，まず poll_next_response メソッドを使って有効な応答があるかどうかを判定し，次に get_next_response メソッドを使って結果を受信します。次のサンプルコードは，複数の要求の受信方法を示します。

複数の要求を送信し，その結果を受信するための VisiBroker ORB メソッド

```
class CORBA {
   class ORB {
      . . .
      typedef sequence <Request_ptr> RequestSeq;
```

```
        void send_multiple_requests_oneway(const RequestSeq &);
        void send_multiple_requests_deferred(const RequestSeq &);
        Boolean poll_next_response();
        Status get_next_response();
        . . .
    };
};
```

DII によるインターフェースリポジトリの使い方

インターフェースリポジトリ（IR）内の情報を DII の Request オブジェクトに格納する場合もあります。第 21 章「インターフェースリポジトリの使い方」を参照してください。次の例では，インターフェースリポジトリを使ってオペレーションのパラメータを取得します。この例は，実際の DII アプリケーションの代表例とは異なり，リモートオブジェクトの型（Account）とリモートオブジェクトの 1 つのメソッドの名前（balance）をあらかじめ知っていることに注意してください。実際の DII アプリケーションでは，ユーザーなど，外部の情報源から情報が取得されます。

- any Account オブジェクトにバインドする。
- IR の Account の balance メソッドをチェックし，IR OperationDef からオペレーションリストを構築する。
- 引数と結果のコンポーネントを作成し，これらを _create_request メソッドに渡す。balance メソッドは例外を返さないことに注意してください。
- Request を呼び出し，結果を抽出して出力する。

```
// acctdii_ir.C
// このサンプルは，IR と DII の使い方を示します。
#include <iostream.h>
#include "corba.h"
int main(int argc, char* const* argv) {
   CORBA::ORB_ptr orb;
   CORBA::Object_var account;
   CORBA::NamedValue_var result;
   CORBA::Any_ptr resultAny;
   CORBA::Request_var req;
   CORBA::NVList_var operation_list;
   CORBA::Any customer;
   CORBA::Float acct_balance;
try {
    // 口座名として argv[1]，またはデフォルトを使用します。
    CORBA::String_var name;
    if (argc == 2)
       name = (const char *) argv[1];
    else
       name = (const char *) "Default Name";
   try {
      // ORB を初期化します。
      orb = CORBA::ORB_init(argc, argv);
   } catch(const CORBA::Exception& excep) {
      cout << "Failure during ORB_init" << endl;
      cout << excep << endl;
      exit(1);
   }
   cout << "ORB_init succeeded" << endl;
   // 従来のバインドとは異なり，このバインドは「orb」に対して呼び出され，
   // インターフェース名に基づく汎用のオブジェクトポインタを返します。
   try {
 account = orb->bind("IDL:Account:1.0");
   } catch(const CORBA::Exception& excep) {
      cout << "Error binding to account" << endl;
      cout << excep << endl;
```

```
      exit(2);
   }
   cout << "Bound to account object" << endl;
   // Account の「balance」メソッドに対するオペレーション記述を
   // 取得します。
   try {
      CORBA::InterfaceDef_var intf = account->_get_interface();
      if (intf == CORBA::InterfaceDef::_nil()) {
         cout << "Account returned a nil interface definition. " << endl;
         cout << "  Be sure an Interface Repository is running and" << endl;
         cout << "  properly loaded" << endl;
         exit(3);
      }
CORBA::Contained_var oper_container = intf->lookup("balance");
CORBA::OperationDef_var oper_def =
      CORBA::OperationDef::_narrow(oper_container);
      orb->create_operation_list(oper_def, operation_list.out());

   } catch(const CORBA::Exception& excep) {
      cout << "Error while obtaining operation list" << endl;
      cout << excep << endl;
      exit(4);
   }
   // Account オブジェクトに送信する要求を作成します。
   try {
      // 結果のプレースホルダーを作成します。
      orb->create_named_value(result.out());
      resultAny = result->value();
      resultAny->replace( CORBA::_tc_float, &result);
      // operation_list 内に引数の値を設定します。
CORBA::NamedValue_ptr arg = operation_list->item(0);
CORBA::Any_ptr anyArg = arg->value();
      *anyArg <<= (const char *) name;

      // 要求を作成します。
 account->_create_request(CORBA::Context::_nil(),

          "balance",
          operation_list,
          result,
          req.out(),
          0);

   } catch(const CORBA::Exception& excep) {
      cout << "Error while creating request" << endl;
      cout << excep << endl;
      exit(5);
   }
   // 要求を実行します。
   try {
      req->invoke();
      CORBA::Environment_ptr env = req->env();
      if ( env->exception() ) {
         cout << "Exception occurred" << endl;
         cout << *(env->exception()) << endl;
         acct_balance = 0;
      } else  {
         // 戻り値を取得します。
acct_balance = *(CORBA::Float *)resultAny->value();
      }
   } catch(const CORBA::Exception& excep) {
      cout << "Error while invoking request" << endl;
      cout << excep << endl;
      exit(6);
   }
   // 結果を出力します。
```

```
    cout << "The balance in " << name << "'s account is $";
    cout << acct_balance << "." << endl;
} catch ( const CORBA::Exception& excep ) {
    cout << "Error occurred" << endl;
    cout << excep << endl;
}
```

第 23 章

動的スケルトンインターフェースの使い方

この節では，クライアント要求に応答するためにオブジェクトサーバーが実行時にオブジェクトインプリメンテーションを動的に作成するしくみについて説明します。

動的スケルトンインターフェースの概要

動的スケルトンインターフェース（DSI：Dynamic Skeleton Interface）は，生成されるスケルトンインターフェースを継承しないオブジェクトインプリメンテーションを作成するメカニズムです。通常，オブジェクトインプリメンテーションは，idl2cpp コンパイラによって生成されるスケルトンクラスから派生されます。DSI を使用すると，オブジェクトは，idl2cpp コンパイラによって生成されるスケルトンクラスを継承しなくても，自分自身を ORB に登録し，クライアントからオペレーションリクエストを受信して処理し，その結果をクライアントに返すことができます。

メモ　クライアントプログラムから見ると，DSI で実装されたオブジェクトは，ほかの VisiBroker ORB オブジェクトとまったく同じように動作します。クライアントは，DSI を使用するオブジェクトインプリメンテーションと通信するために，特別なオペレーションを提供する必要はありません。

VisiBroker ORB は，オブジェクトの invoke メソッドを呼び出し，それを ServerRequest オブジェクトに渡して，クライアントオペレーションリクエストを DSI オブジェクトインプリメンテーションに提供します。オブジェクトインプリメンテーションの役割は，要求されたオペレーションの判定，その要求にバインドされた引数の解釈，要求に応答するために内部の 1 つ以上のメソッドを呼び出すこと，および適切な値を返すことです。

オブジェクトスケルトンの提供する通常の言語マッピングを使用するよりも，DSI を使ったオブジェクトインプリメンテーションの方が必要なプログラミング作業は増えてしまいますが，DSI を使って実装されたオブジェクトは，プロトコル間ブリッジを提供する場合にたいへん役立ちます。

オブジェクトインプリメンテーションを動的に作成するための手順

DSI を使って動的にオブジェクトインプリメンテーションを作成するには，次の操作を実行します。

1 IDL をコンパイルする場合は，`-type_code_inf` フラグを使用します。
2 スケルトンクラスから派生するのではなく，`PortableServer::DynamicImplementation` 抽象クラスから派生するように，オブジェクトインプリメンテーションを設計します。
3 `invoke` メソッドを宣言して実装する。VisiBroker ORB がオブジェクトにクライアント要求を送るためにこれを使用します。
4 オブジェクトインプリメンテーション（POA サーバント）をデフォルトのサーバントとして POA マネージャに登録します。

DSI を使用するサンプルプログラム

DSI の使い方を紹介するサンプルプログラムは，次のディレクトリにあります。

```
<install_dir>/examples/vbe/basic/bank_dynamic
```

この節では，このサンプルを使って DSI の概念を説明します。次に示す `Bank.idl` ファイルは，このサンプルで実装されているインターフェースを示します。

```
// Bank.idl
module Bank {
  interface Account {
    float balance();
  };
  interface AccountManager {
    Account open(in string name);
  };
};
```

DynamicImplementation クラスの拡張

DSI を使用するには，次の `DynamicImplementation` クラスを基底クラスとしてオブジェクトインプリメンテーションを派生させる必要があります。このクラスは，複数のコンストラクタと `invoke` メソッドを提供するので，それらを実装する必要があります。

```
class PortableServer::DynamicImplementation : public virtual PortableServer::ServantBase
{
   public:
      virtual void invoke(PortableServer::ServerRequest_ptr request) = 0;
. . .
};
```

動的要求のオブジェクトを設計するサンプル

次のサンプルコードは，DSI を使って実装される `AccountImpl` クラスの宣言です。これは，`DynamicImplementation` クラスから派生し，`invoke` メソッドを宣言します。VisiBroker ORB は `invoke` メソッドを呼び出し，クライアントのオペレーションリクエストを `ServerRequest` オブジェクトの形式でインプリメンテーションに渡します。

下のサンプルコードは，`Account` クラスコンストラクタと `_primary_interface` 関数です。

```
class AccountImpl : public PortableServer::DynamicImplementation {
   public:
      AccountImpl(PortableServer::Current_ptr current,
            PortableServer::POA_ptr poa)
            : _poa_current(PortableServer::Current::_duplicate(current)),
```

```
            _poa(poa)
        {}
        CORBA::Object_ptr get(const char *name) {
            CORBA::Float balance;
            // 口座が存在するかどうかをチェックします。
            if (!_registry.get(name, balance)) {
                // 新しい口座を作成するまでの間を設けます。
                VISPortable::vsleep(3);
                // 0 〜 1000 ドルの範囲で口座に残高を設定します。
                balance = abs(rand()) % 100000 / 100.0;
                // 新しい口座を出力します。
                cout << "Created " << name << "'s account: " << balance << endl;
                _registry.put(name, balance);
            }
            // オブジェクトリファレンスを返します。
            PortableServer::ObjectId_var accountId =
            PortableServer::string_to_ObjectId(name);
            return _poa->create_reference_with_id(accountId, "IDL:Bank/
                        Account:1.0");
        }
    private:
        AccountRegistry _registry;
        PortableServer::POA_ptr _poa;
        PortableServer::Current_var _poa_current;
        CORBA::RepositoryId _primary_interface(
                const PortableServer::ObjectId& oid, PortableServer::POA_ptr poa) {
            return CORBA::string_dup((const char *)"IDL:Bank/Account:1.0");
        };
        void invoke(CORBA::ServerRequest_ptr request) {
            // オブジェクト ID から口座名を取得します。
            PortableServer::ObjectId_var oid = _poa_current->get_object_id();
            CORBA::String_var name;
            try {
                name = PortableServer::ObjectId_to_string(oid);
            } catch (const CORBA::Exception& e) {
                throw CORBA::OBJECT_NOT_EXIST();
            }
            // オペレーション名が正しいことを確認します。
            if (strcmp(request->operation(), "balance") != 0) {
                throw CORBA::BAD_OPERATION();
            }
            // 残高を求め，結果を格納します。
            CORBA::NVList_ptr params = new CORBA::NVList(0);
            request->arguments(params);
            CORBA::Float balance;
            if (!_registry.get(name, balance))
                throw CORBA::OBJECT_NOT_EXIST();
            CORBA::Any result;
            result <<= balance;
            request->set_result(result);
            cout << "Checked " << name << "'s balance: " << balance << endl;
        }
};
```

次のサンプルコードは，DSI を使って実装される AccountManagerImpl クラスのインプリメンテーションです。これは，DynamicImplementation クラスから派生し，invoke メソッドを宣言します。VisiBroker ORB は invoke メソッドを呼び出し，クライアントのオペレーションリクエストを ServerRequest オブジェクトの形式でインプリメンテーションに渡します。

```
class AccountManagerImpl : public PortableServer::DynamicImplementation {
    public:
        AccountManagerImpl(AccountImpl* accounts) { _accounts = accounts; }
        CORBA::Object_ptr open(const char* name) {
            return _accounts->get(name);
        }
```

```
    private:
       AccountImpl* _accounts;
       CORBA::RepositoryId _primary_interface(
             const PortableServer::ObjectId& oid,
             PortableServer::POA_ptr poa) {
          return CORBA::string_dup((const char *)"IDL:Bank/AccountManager:1.0");
       };
       void invoke(CORBA::ServerRequest_ptr request) {
          // オペレーション名が正しいことを確認します。
          if (strcmp(request->operation(), "open") != 0)
             throw CORBA::BAD_OPERATION();
          // 入力パラメータを取得します。
          char *name = NULL;
          try {
             CORBA::NVList_ptr params = new CORBA::NVList(1);
             CORBA::Any any;
             any <<= (const char*) "";
             params->add_value("name", any, CORBA::ARG_IN);
             request->arguments(params);
             *(params->item(0)->value()) >>= name;
          } catch (const CORBA::Exception& e) {
             throw CORBA::BAD_PARAM();
          }
          // 実際のインプリメンテーションを呼び出して，結果を格納します。
          CORBA::Object_var account = open(name);
          CORBA::Any result;
          result <<= account;
          request->set_result(result);
       }
 };
```

リポジトリ ID の指定

_primary_interface メソッドは，サポートしているリポジトリ識別子を返すように実装する必要があります。リポジトリ識別子を正しく指定するには，オブジェクトの IDL インターフェース名から順に次の操作を実行します。

1. 先頭以外のデリミタスコープ解決演算子（::）をすべてスラッシュ（/）に置き換えます。
2. 文字列の先頭に「IDL:」を追加します。
3. 文字列の末尾に「:1.0」を追加します。

たとえば，このサンプルコードは IDL インターフェース名です。

```
Bank::AccountManager
```

その名前から得たリポジトリ識別子は次のようになっています。

```
IDL:Bank/AccountManager:1.0
```

ServerRequest クラスについて

ServerRequest オブジェクトは，オブジェクトインプリメンテーションの invoke メソッドに対するパラメータとして渡されます。ServerRequest オブジェクトはオペレーションリクエストを表し，要求されたオペレーションの名前，パラメータリスト，およびコンテキストを取得するためのメソッドを提供します。また，呼び出し元に返される結果を設定するメソッドと例外を反映するメソッドも提供します。

```
class CORBA::ServerRequest {
   public:
   const char* op_name() const { return _operation; }
   void params(CORBA::NVList_ptr);
   void result(CORBA::Any_ptr);
```

```
      void exception(CORBA::Any_ptr exception);
      ...
      CORBA::Context_ptr ctx() {
         ...
      }
      // POA 仕様のメソッド
      const char *operation() const { return _operation; }
      void arguments(CORBA::NVList_ptr param) { params(param); }
      void set_result(const CORBA::Any& a) { result(new CORBA::Any(a)); }
      void set_exception(const CORBA::Any& a) {
         exception(new CORBA::Any(a));
      }
   };
```

arguments，set_result，または set_exception メソッドに渡されたすべての引数は，その後 VisiBroker ORB によって所有されます。これらの引数のためのメモリは，VisiBroker ORB によって解放されるので，これを解放してはなりません。

メモ 次のメソッドは使用しなくなっています。

- op_name
- params
- result
- exception

Account オブジェクトの実装

Account インターフェースが宣言するメソッドは 1 つだけなので，AccountImpl クラスの invoke メソッドで行う処理は比較的簡単です。

このメソッドは，最初に要求されたオペレーション名が「balance」であるかどうかをチェックします。名前が一致しない場合は，BAD_OPERATION 例外が生成されます。Account オブジェクトが複数のメソッドを提供する場合，invoke メソッドはすべてのオペレーション名をチェックし，適切な内部メソッドを使って要求を処理する必要があります。

balance メソッドはパラメータを受け取らないので，オペレーションリクエストと関連するパラメータリストはありません。balance メソッドが呼び出されるだけで，その結果が Any オブジェクトにパッケージされて，呼び出し元に返されます。これには，ServerRequest オブジェクトの set_result メソッドが使用されます。

AccountManager オブジェクトの実装

Account オブジェクト同様，AccountManager インターフェースも 1 つのメソッドを宣言します。しかし，AccountManagerImpl オブジェクトの open メソッドはアカウント名パラメータを受け取りません。これにより，invoke メソッドによる処理は少し複雑なものになります。

このメソッドは，最初に要求されたオペレーション名が「open」であるかどうかをチェックします。名前が一致しない場合は，BAD_OPERATION 例外が生成されます。AccountManager オブジェクトが複数のメソッドを提供する場合，invoke メソッドはすべてのオペレーション名をチェックし，適切な内部メソッドを使って要求を処理する必要があります。

入力パラメータの処理

AccountManagerImpl オブジェクトの invoke メソッドは，次の順序でオペレーションリクエストの入力パラメータを処理します。

1 オペレーションのパラメータリストを保持する NVList を作成する。

2 所定のパラメータごとに Any オブジェクトを作成し，TypeCode とパラメータの種類（ARG_IN，ARG_OUT，ARG_INOUT）を設定して，NVList に追加する。

3 ServerRequest オブジェクトの arguments メソッドを呼び出し，NVList を渡して，リストにあるすべてのパラメータを更新する。

open メソッドは口座名のパラメータを 1 つ受け取るので，ServerRequest 内でパラメータを保持するために NVList オブジェクトが 1 つ作成されます。NVList クラスは，1 つ以上の NamedValue オブジェクトを含むパラメータリストを実装します。NVList クラスと NamedValue クラスについては，第 22 章「動的起動インターフェースの使い方」を参照してください。

アカウント名を保持するために Any オブジェクトが作成されます。次に，引数の名前を「name」に，パラメータの種類を ARG_IN に設定して，この Any が NVList に追加されます。

NVList が初期化されると，ServerRequest オブジェクトの arguments メソッドが起動され，リストにあるすべてのパラメータの値が取得されます。

メモ　arguments メソッドを呼び出した後，NVList は VisiBroker ORB に所有されます。つまり，オブジェクトインプリメンテーションが NVList 内の ARG_INOUT パラメータを変更した場合，ORB は自動的にそれを認識します。この NVList を呼び出し元で解放することはできません。

入力引数の NVList を生成するかわりとして，VisiBroker ORB オブジェクトの create_operation_list メソッドを使用することもできます。このメソッドは OperationDef を受け取り，必要なすべての Any オブジェクトを完全に初期化して NVList オブジェクトを返します。適切な OperationDef オブジェクトは，インターフェースリポジトリから取得できます。281 ページの「インターフェースリポジトリの使い方」を参照してください。

戻り値の設定

ServerRequest オブジェクトの arguments メソッドを呼び出したら，name パラメータの値を抽出して，新しい Account オブジェクトを作成するために使用できます。新しく作成された Account オブジェクトを保持するために，Any オブジェクトが作成されます。ServerRequest オブジェクトの set_result メソッドを呼び出すと，これが呼び出し元に返されます。

サーバーインプリメンテーション

次のサンプルコードで示す main ルーチンのインプリメンテーションは，15 ページの「VisiBroker を使ったサンプルアプリケーションの開発」で紹介した元のサンプルとほとんど同じです。

```
int main(int argc, char* const* argv) {
   try {
      // ORB を初期化します。
      CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
      // ルート POA へのリファレンスを取得します。
      CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
      PortableServer::POA_var rootPOA = PortableServer::POA::_narrow(obj);
      // POA マネージャを取得します。
      PortableServer::POAManager_var poaManager = rootPOA->the_POAManager();
      // 適切なポリシーで accountPOA を作成します。
      CORBA::PolicyList accountPolicies;
      accountPolicies.length(3);
      accountPolicies[(CORBA::ULong)0] =
      rootPOA->create_servant_retention_policy(PortableServer::NON_RETAIN);
      accountPolicies[(CORBA::ULong)1] =
      rootPOA->create_request_processing_policy(
      PortableServer::USE_DEFAULT_SERVANT);
      accountPolicies[(CORBA::ULong)2] =
      rootPOA->create_id_uniqueness_policy(
```

```
      PortableServer::MULTIPLE_ID);
      PortableServer::POA_var accountPOA =
            rootPOA->create_POA("bank_account_poa",
            poaManager,
            accountPolicies);
      // accountPOA のデフォルトのサーバントを作成します。
      PortableServer::Current_var current = PortableServer::Current::_instance();
      AccountImpl accountServant(current, accountPOA);
      accountPOA->set_servant(&accountServant);
      // 適切なポリシーで managerPOA を作成します。
      CORBA::PolicyList managerPolicies;
      managerPolicies.length(3);
      managerPolicies[(CORBA::ULong)0] = rootPOA->create_lifespan_policy(
      PortableServer::PERSISTENT);
      managerPolicies[(CORBA::ULong)1] =
      rootPOA->create_request_processing_policy(
      PortableServer::USE_DEFAULT_SERVANT);
      managerPolicies[(CORBA::ULong)2] =
      rootPOA->create_id_uniqueness_policy(
      PortableServer::MULTIPLE_ID);
      PortableServer::POA_var managerPOA = rootPOA->create_POA("bank_agent_poa",
      poaManager,
      managerPolicies);
      // managerPOA のデフォルトのサーバントを作成します。
      AccountManagerImpl managerServant(&accountServant);
      managerPOA->set_servant(&managerServant);
      // POA マネージャをアクティブ化します。
      poaManager->activate();
      cout << "AccountManager is ready" << endl;
      // 着信要求を待機します。
      orb->run();
   } catch(const CORBA::Exception& e) {
      cerr << e << endl;
      return 1;
   }
   return 0;
}
```

DSI インプリメンテーションはデフォルトサーバントとしてインスタンス化されます。POA の作成には，対応するポリシーが必要です。詳細については，第 9 章「POA の使い方」を参照してください。

第 24 章

ポータブルインターセプタの使い方

ここでは，ポータブルインターセプタの概要について説明します。また，いくつかのポータブルインターセプタのサンプルを紹介し，ポータブルインターセプタファクトリなどの高度な機能についても説明します。

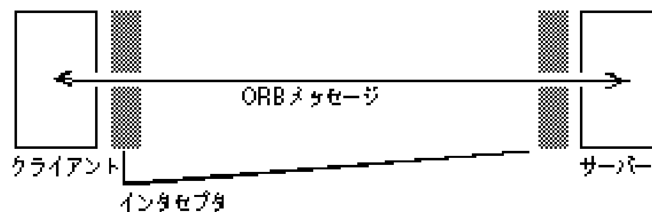
ポータブルインターセプタの詳細については，OMG Final Adopted Specification（最終採用仕様）ptc/2001-04-03 の「Portable Interceptors」を参照してください。

ポータブルインターセプタの概要

VisiBroker ORB は，「インターセプタ」と呼ばれるインターフェースのセットを提供します。インターセプタは，セキュリティ，トランザクション，ログなどの ORB の追加機能を組み込むためのフレームワークを提供します。これらのインターセプタインターフェースは，「コールバック」メカニズムに基づいています。たとえば，インターセプタを使用すると，クライアントとサーバー間の通信を検知し，必要な場合にこれらの通信を変更して VisiBroker Edition ORB の動作を効率的に変更できます。

最も簡単な使い方として，インターセプタはコードのトレースに役立ちます。クライアントとサーバー間で交わされるメッセージを監視できるので，ORB がどのように要求を処理しているかを正確に判定できます。

図 24.1　インターセプタのしくみ



監視ツールやセキュリティ層などのさらに高度なアプリケーションを作成する場合，このようなより低いレベルのアプリケーションの動作に必要な情報と制御は，インターセプタから提供されます。たとえば，数多くのサーバーのアクティビティを監視して，負荷分散を実行するアプリケーションを開発できます。

インターセプタの種類

VisiBroker ORB では，2 種類のインターセプタがサポートされています。

表 24.1　VisiBroker ORB でサポートされているインターセプタの種類

ポータブルインターセプタ	VisiBroker インターセプタ
ポータブルコードのインターセプタを記述できる OMG の標準機能で，さまざまなベンダーの ORB と一緒に使用できます。	VisiBroker 固有のインターセプタです。詳細については，第 25 章「VisiBroker インターセプタの使い方」を参照してください。

ポータブルインターセプタの種類

OMG 仕様で定義されているポータブルインターセプタには，*リクエストインターセプタ*と *IOR インターセプタ*の 2 種類があります。

表 24.2　ポータブルインターセプタの種類

リクエストインターセプタ	IOR インターセプタ
VisiBroker ORB サービスを有効にし，クライアントとサーバーの間でコンテキスト情報を転送できます。リクエストインターセプタには，*クライアントリクエストインターセプタ*と *サーバーリクエストインターセプタ*があります。	VisiBroker ORB サービスがサーバーやオブジェクトの ORB サービス関連機能について記述された情報を IOR に追加できるようにします。たとえば，SSL などのセキュリティサービスでは，タグ付きコンポーネントを IOR に追加することで，そのコンポーネントを認識するクライアントがコンポーネント内の情報に基づいてサーバーとの接続を確立できます。

ポータブルインターセプタと VisiBroker インターセプタの詳細については，第 25 章「VisiBroker インターセプタの使い方」を参照してください。

「VisiBroker for Java APIs」と「ポータブルインターセプタのインターフェースとクラス」も参照してください。

ポータブルインターセプタと情報インターフェース

すべてのポータブルインターセプタは，次のベースインターセプタ API クラスの 1 つを実装します。これらの API クラスは，VisiBroker Edition ORB によって定義および実装されています。

- リクエストインターセプタ
  - `ClientRequestInterceptor`
  - `ServerRequestInterceptor`
- `IORInterceptor`

Interceptor クラス

上記のインターセプタはすべて，よく使用するクラス `Interceptor` から継承されています。`Interceptor` クラスでは，継承されたクラスでよく使用されるメソッドが定義されています。

Interceptor クラス：

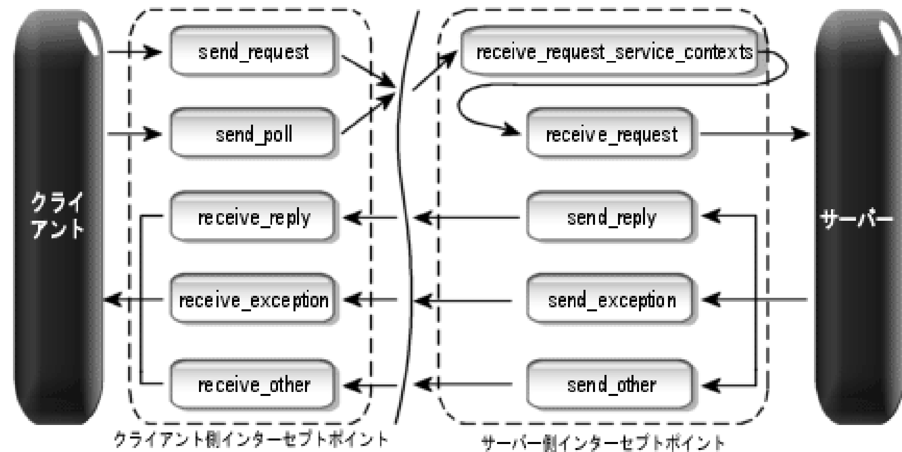
```
class PortableInterceptor::Interceptor
{
   virtual char* name() = 0;
   virtual void destroy() = 0;
}
```

リクエストインターセプタ

「リクエストインターセプタ」を使用して，特定のインターセプトポイントで要求／応答のシーケンスの流れをインターセプトします。これにより，サービスはクライアントとサーバー間でコンテキスト情報を送受信できます。各インターセプトポイントでは, VisiBroker Edition ORB がオブジェクトを与え，このオブジェクトによってインターセプタは要求情報にアクセスできます。リクエストインターセプタには次の 2 種類あり，それぞれ対応する要求情報インターフェースがあります。

- `ClientRequestInterceptor` と `ClientRequestInfo`
- `ServerRequestInterceptor` と `ServerRequestInfo`

図 24.2　要求インターセプトポイント



リクエストインターセプタの詳細については,「VisiBroker for Java APIs」と「ポータブルインターセプタのインターフェースとクラス」を参照してください。

ClientRequestInterceptor

`ClientRequestInterceptor` には，クライアント側で実装されるインターセプトポイントがあります。次の表に示すように，OMG によって `ClientRequestInterceptor` で定義されているインターセプトポイントは 5 つあります。

表 24.3　ClientRequestInterceptor インターセプトポイント

インターセプトポイント	説明
`send_request`	クライアント側のインターセプタは，要求がサーバーに送信される前に要求を照会し，サービスコンテキストを変更できます。
`send_poll`	TII（Time-Independent Invocation）[1] ポーリングが応答シーケンスを取得する間に，クライアント側のインターセプタで要求を照会できます。
`receive_reply`	応答情報がサーバーから戻されてクライアントが制御する前に，クライアント側のインターセプタで応答情報を照会できます。
`receive_exception`	例外が発生してクライアントに送信される前に，クライアント側のインターセプタがその例外の情報を照会できます。
`receive_other`	通常の応答以外の要求結果や例外を受け取ったときに，クライアント側のインターセプタで使用可能な情報を照会できます。

[1]TII は VisiBroker Edition ORB では実装されません。その結果，`send_poll( )` インターセプトポイントは起動されません。

各インターセプトポイントの詳細については,「VisiBroker for Java APIs」と「ポータブルインターセプタのインターフェースとクラス」を参照してください。

```
class _VISEXPORT ClientRequestInterceptor: public virtual Interceptor
{
   public:
      virtual void send_request(ClientRequestInfo_ptr _ri) = 0;
      virtual void send_poll(ClientRequestInfo_ptr _ri) = 0;
      virtual void receive_reply(ClientRequestInfo_ptr _ri) = 0;
      virtual void receive_exception(ClientRequestInfo_ptr _ri) = 0;
      virtual void receive_other(ClientRequestInfo_ptr _ri) = 0;
};
```

クライアント側の規則

次にクライアント側の規則を示します。

- 開始インターセプトポイントは send_request と send_poll です。指定した要求／応答のシーケンスで呼び出されるインターセプトポイントは，このうちの 1 つだけです。
- 終了インターセプトポイントは，receive_reply，receive_exception，および receive_other です。
- 中間インターセプトポイントはありません。
- 終了インターセプトポイントは，send_request または send_poll の実行が成功した場合にだけ呼び出されます。
- receive_exception は，ORB がシャットダウンしたことで要求がキャンセルされると，システム例外 BAD_INV_ORDER（マイナーコード 4）とともに呼び出されます。マイナーコード 4 は ORB がシャットダウンしたことを表しています。
- 要求が何らかの理由でキャンセルされると，receive_exception がマイナーコード 3 のシステム例外 TRANSIENT とともに呼び出されます。

正常な呼び出し	send_request の後に receive_reply が続きます。開始ポイントの後に終了ポイントが続きます。
再試行	send_request の後に receive_other が続きます。開始ポイントの後に終了ポイントが続きます。

ServerRequestInterceptor

ServerRequestInterceptor には，サーバー側で実装されるインターセプトポイントがあります。ServerRequestInterceptor で定義されているインターセプトポイントは，5 つあります。次の表に，ServerRequestInterceptor のインターセプトポイントを示します。

表 24.4　ServerRequestInterceptor インターセプトポイント

インターセプトポイント	説明
receive_request_service_contexts	サーバー側インターセプタは，着信した要求からサービスのコンテキスト情報を取得し，PortableInterceptor::Current のスロットに転送できます。
receive_request	サーバー側インターセプタは，オペレーションパラメータを含むすべての情報が使用可能になってから要求情報を照会できます。
send_reply	ターゲットオペレーションが呼び出されて応答がサーバーに戻される前に，サーバー側インターセプタは応答情報を照会して応答のサービスコンテキストを変更できます。
send_exception	例外が発生してクライアントに送信される前に，サーバー側インターセプタは例外の情報を照会して応答のサービスコンテキストを変更できます。
send_other	通常の応答以外の要求結果や例外を受け取ったときに，サーバー側のインターセプタで使用可能な情報を照会できます。

各インターセプトポイントの詳細については，「VisiBroker for Java APIs」と「ポータブルインターセプタのインターフェースとクラス」を参照してください。

ServerRequestInterceptor クラス：

```
class _VISEXPORT ServerRequestInterceptor: public virtual Interceptor
{
   public:
      virtual void receive_request_service_contexts(ServerRequestInfo_ptr _ri) = 0;
      virtual void receive_request(ServerRequestInfo_ptr _ri) = 0;
      virtual void send_reply(ServerRequestInfo_ptr _ri) = 0;
      virtual void send_exception(ServerRequestInfo_ptr _ri) = 0;
      virtual void send_other(ServerRequestInfo_ptr _ri) = 0;
};
```

サーバー側の規則

次にサーバー側の規則を示します。

- 開始インターセプトポイントは receive_request_service_contexts です。このインターセプトポイントは，指定したすべての要求／応答シーケンス上で呼び出されます。
- 終了インターセプトポイントは，send_reply，send_exception，および send_other です。指定した要求／応答のシーケンスで呼び出されるインターセプトポイントは，このうちの 1 つだけです。
- 中間インターセプトポイントは receive_request です。これは receive_request_service_contexts の後と終了インターセプトポイントの前で呼び出されます。
- 例外では receive_request は呼び出されません。
- 終了インターセプトポイントは，send_request または send_poll の実行が成功した場合にだけ呼び出されます。
- send_exception は，ORB がシャットダウンしたことで要求がキャンセルされると，システム例外 BAD_INV_ORDER（マイナーコード 4）とともに呼び出されます。マイナーコード 4 は ORB がシャットダウンしたことを表しています。
- 要求が何らかの理由でキャンセルされると，send_exception がマイナーコード 3 のシステム例外 TRANSIENT とともに呼び出されます。

正常な呼び出し　インターセプトポイントの順序は，receive_request_service_contexts，receive_request，send_reply です。開始ポイント，中間ポイント，終了ポイントの順に続きます。

IOR インターセプタ

アプリケーションで IORInterceptor を使用すると，サーバーまたはオブジェクトの ORB サービス関連の機能に関する情報をオブジェクトリファレンスに追加することができます。これにより，クライアントの VisiBroker Edition ORB サービスインプリメンテーションが適切に機能するようになります。これを実行するには，インターセプトポイント establish_components を呼び出します。IORInfo のインスタンスが，インターセプトポイントに渡されます。IORInfo の詳細については，「VisiBroker for Java APIs」と「ポータブルインターセプタのインターフェースとクラス」を参照してください。

```
class _VISEXPORT IORInterceptor: public virtual Interceptor
{
   public:
      virtual void establish_components(IORInfo_ptr _info) = 0;
      virtual void components_established(IORInfo_ptr _info) = 0;
      virtual void adapter_manager_state_changed(
         CORBA::Long _id, CORBA::Short _state) = 0;
      virtual void adapter_state_changed(
         const ObjectReferenceTemplateSeq& _templates,
         CORBA::Short _state) = 0;
};
```

ポータブルインターセプタ（PI）Current

PortableInterceptor::Current（以降 PICurrent）は，ポータブルインターセプタが使用できるスロットのテーブルで，スレッドのコンテキスト情報を要求コンテキストに転送することができます。PICurrent は不要な場合もあります。ただし，インターセプトポイントでクライアントのスレッドコンテキスト情報が必要な場合は，PICurrent を使用してこの情報を転送します。

PICurrent は次の呼び出しを介して取得します。

```
ORB->resolve_initial_references("PICurrent");
```

PortableInterceptor::Current クラス：

```
class _VISEXPORT Current: public virtual CORBA::Current, public virtual CORBA_Object
{
   public:
      virtual CORBA::Any* get_slot(CORBA::ULong _id);
      virtual void set_slot(CORBA::ULong _id, const CORBA::Any& _data);
};
```

Codec

Codec には，インターセプタに対して，これらのコンポーネントを対応する IDL データ型形式と CDR カプセル化形式の間で転送する機構があります。Codec は CodecFactory から取得できます。詳細については，318 ページの「CodecFactory」を参照してください。

Codec クラス：

```
class _VISEXPORT Codec
{
   public:
      virtual CORBA::OctetSequence* encode(const CORBA::Any& _data) = 0;
      virtual CORBA::Any* decode(const CORBA::OctetSequence& _data) = 0;
      virtual CORBA::OctetSequence* encode_value(const CORBA::Any& _data) = 0;
      virtual CORBA::Any* decode_value(const CORBA::OctetSequence& _data,
         CORBA::TypeCode_ptr _tc) = 0;
};
```

CodecFactory

このクラスを使用して，エンコード形式のメジャーバージョンとマイナーバージョンを指定して，Codec オブジェクトを作成します。CodecFactory は次の呼び出しを介して取得します。

```
ORB->resolve_initial_references("CodecFactory")
```

CodecFactory クラス：

```
class _VISEXPORT CodecFactory
{
   public:
      virtual Codec_ptr create_codec(const Encoding& _enc) = 0;
};
```

ポータブルインターセプタの作成

ポータブルインターセプタを作成する一般的な方法は次のとおりです。

1 インターセプタは，次のインターセプタインターフェースのいずれか 1 つから継承される必要があります。
   - ClientRequestInterceptor

- ServerRequestInterceptor
- IORInterceptor

2 インターセプタは，インターセプタで使用できる 1 つ，または複数のインターセプトポイントを実装します。

3 インターセプタには，名前を付けることも匿名にすることもできます。ただし，同じ型のインターセプタに同じ名前を付けることはできません。しかし，VisiBroker Edition ORB には匿名のインターセプタをいくつでも登録できます。

例：PortableInterceptor の作成

```
#include "PortableInterceptor_c.hh"

class SampleClientRequestInterceptor: public
PortableInterceptor::ClientRequestInterceptor
{
   char * name() {
      return "SampleClientRequestInterceptor";
   }

   void send_request(ClientRequestInfo_ptr _ri) {
      ....... // 実際のインターセプタコード
   }

   void send_request(ClientRequestInfo_ptr _ri) {
      ....... // 実際のインターセプタコード
   }

   void receive_reply(ClientRequestInfo_ptr _ri) {
      ....... // 実際のインターセプタコード
   }

   void receive_exception(ClientRequestInfo_ptr _ri) {
      ....... // 実際のインターセプタコード
   }

   void receive_other(ClientRequestInfo_ptr _ri) {
      ....... // 実際のインターセプタコード
   }
};
```

ポータブルインターセプタの登録

ポータブルインターセプタは，使用する前にまず VisiBroker Edition ORB に登録する必要があります。ポータブルインターセプタを登録するには，ORBInitializer オブジェクトが実装されて登録されていなければなりません。ポータブルインターセプタは，その pre_init( ) メソッドまたは post_init( ) メソッド，あるいは両方を実装する関連 ORBInitializer オブジェクトを登録することで, ORB 初期化中にインスタンス化および登録できます。VisiBroker Edition ORB は初期化中に ORBInitInfo オブジェクトを使用して，登録された各 ORBInitializer を呼び出します。

ORBInitializer クラス：

```
class _VISEXPORT ORBInitializer
{
   public:

      virtual void pre_init(ORBInitInfo_ptr _info) = 0;
      virtual void post_init(ORBInitInfo_ptr _info) = 0;
};
```

ORBInitInfo　クラス：

```
class _VISEXPORT ORBInitInfo
{
   public:
      virtual CORBA::StringSequence* arguments() = 0;
      virtual char* orb_id() = 0;
      virtual IOP::CodecFactory_ptr codec_factory() = 0;
      virtual void register_initial_reference(const char* _id, CORBA::Object_ptr _obj) =
0;
      virtual CORBA::Object_ptr resolve_initial_references(const char* _id) = 0;
      virtual void add_client_request_interceptor(
         ClientRequestInterceptor_ptr _interceptor) = 0;
      virtual void add_server_request_interceptor(
         ServerRequestInterceptor_ptr _interceptor) = 0;
      virtual void add_ior_interceptor(IORInterceptor_ptr _interceptor) = 0;
      virtual CORBA::ULong allocate_slot_id() = 0;
      virtual void register_policy_factory(CORBA::ULong _type,
         PolicyFactory_ptr _policy_factory) = 0;
};
```

ORBInitializer の登録

ORBInitializer を登録するために，グローバルメソッド register_orb_initializer が提供されています。インターセプタを実装する各サービスは，ORBInitializer のインスタンスを提供します。サービスを使用するには，アプリケーションで次の操作を実行します。

1. サービスの ORBInitializer を使って register_orb_initializer( ) を呼び出します。
2. 新規の ORB 識別子を使用して, 新規の ORB を生成するインスタンス化 ORB_Init( ) 呼び出しを作成します。

ORB.init( ) で次の処理が行われます。

1. org.omg.PortableInterceptor.ORBInitializerClass で始まるこれらの ORB プロパティを回収します。
2. 各プロパティの <Service> 部分を回収します。
3. クラス名として <Service> 文字列を使用して，オブジェクトをインスタンス化します。
4. そのオブジェクトの pre_init( ) メソッドと post_init( ) メソッドを呼び出します。
5. 例外があっても，ORB は無視して続行します。

メモ 名前の競合を防ぐために，逆の DNS 命名規則をお勧めします。たとえば，ABC 社に 2 つの初期化子がある場合，次のプロパティを定義できます。

```
org.omg.PortableInterceptor.ORBInitializerClass.com.abc.ORBInit1
```

```
org.omg.PortableInterceptor.ORBInitializerClass.com.abc.ORBInit2
```

register_orb_initializer メソッドは PortableInterceptor モジュールで次のように定義されます。

```
class _VISEXPORT PortableInterceptor {
   static void register_orb_initializer(ORBInitializer *init);
}
```

例：ORBInitializer の登録

ABC 社が記述したクライアント側の監視ツールには，次の ORBInitializer が実装されています。

```
#include "PortableInterceptor_c.hh"

class MonitoringService: public PortableInterceptor::ORBInitializer
{
   void pre_init(ORBInitInfo_ptr _info)
   {
```

```
         // サービスのインターセプタをインスタンス化します。
         Interceptor* interceptor = new MonitoringInterceptor();

         // 監視のインターセプタを登録します。
         _info->add_client_request_interceptor(interceptor);
      }

      void post_init(ORBInitInfo_ptr _info)
      {
         // この init ポイントは必要ではありません。
      }
   };

   MonitoringService * monitoring_service = new MonitoringService();
   PortableInterceptor::register_orb_initializer(monitoring_service);
```

次のコマンドは，この監視サービスを使って MyApp と呼ばれるプログラムを実行します。

```
java -
Dorg.omg.PortableInterceptor.ORBInitializerClass.com.abc.Monitoring.MonitoringService
MyApp
```

VisiBroker によるポータブルインターセプタの拡張機能

POA スコープ付きサーバーリクエストインターセプタ

OMG の指定したポータブルインターセプタは，グローバルにスコープされます。VisiBroker は，新しいモジュール呼び出し PortableInterceptorExt を追加して，ポータブルインターセプタの public 拡張機能である「POA スコープ付きサーバーリクエストインターセプタ」を定義しています。この新しいモジュールには PortableInterceptor::IORInfo から継承されたローカルインターフェース IORInfoExt が保持されており，POA スコープ付きサーバーリクエストインターセプタをインストールするためのメソッドが追加されています。

IORInfoExt クラス：

```
#include "PortableInterceptorExt_c.hh"

class IORInfoExt: public PortableInterceptor::IORInfo
{
   public:
      virtual void add_server_request_interceptor(
         ServerRequestInterceptor_ptr _interceptor) = 0;
      virtual char* full_poa_name();
};
```

VisiBroker ポータブルインターセプタのインプリメンテーションの制限

次に，VisiBroker におけるポータブルインターセプタのインプリメンテーションの制限を示します。

ClientRequestInfo の制限

- arguments()，result()，exceptions()，contexts()，operation_contexts()：DII 呼び出しでのみ使用できます。
- operation_context()：使用できません。CORBA::NO_RESOURCES が生成されます。
- received_exception()：タイプコード情報が利用できる場合にだけ使用できます（たとえば，-typecode_info を使って IDL をコンパイルし，プログラムにリンクした場合）。タイプコード情報がない場合は，常に CORBA::UNKNOWN が返されます。

ServerRequestInfo の制限

- arguments() と result()：DSI 呼び出しでのみ使用できます。詳細については，第 23 章「動的スケルトンインターフェースの使い方」を参照してください。
- exceptions()，contexts()，operation_context()：使用できません。CORBA::NO_RESOURCES が生成されます。
- sending_exception()：タイプコード情報が利用できる場合にだけ使用できます（たとえば，-typecode_info を使って IDL をコンパイルし，プログラムにリンクした場合)。タイプコード情報がない場合は，常に CORBA::UNKNOWN が返されます。

ポータブルインターセプタの概要

この節では，ポータブルインターセプタを利用するためにアプリケーションが実際に記述される方法や，各リクエストインターセプタを実装する方法について説明します。各サンプルは，クライアントとサーバーのアプリケーションと，Java と C++ で記述されたクライアントとサーバーのインターセプタから構成されています。各インターフェースの定義の詳細については，「VisiBroker for Java APIs」と「ポータブルインターセプタのインターフェースとクラス」を参照してください。ポータブルインターセプタを利用する開発者は，最新の CORBA 仕様対応のポータブルインターセプタに関する章をお読みになることをお勧めします。

ポータブルインターセプタのサンプルは次のディレクトリに置かれています。

```
<install_dir>/examples/vbe/pi
```

各サンプルは，サンプルの目的をより明確にするために，次のディレクトリ名の 1 つと関連付けられています。

- client_server
- chaining

例：client_server

この節では，client_server のサンプルの目的，説明，コンパイル手順，実行方法，および配布手順について説明します。

サンプルの目的

このサンプルでは，コードを変更しないで，既存の CORBA アプリケーションにポータブルインターセプタを簡単に追加できる方法について説明します。ポータブルインターセプタは，クライアント側とサーバー側のあらゆるアプリケーションに追加できます。追加するには，実行時に設定できる指定オプションやプロパティを使用して，関連アプリケーションをもう一度実行します。

使用されるクライアントおよびサーバーアプリケーションは，次の場所にあるアプリケーションと同様のアプリケーションです。

```
<install_dir>/examples/vbe/basic/bank_agent
```

ポータブルインターセプタは，実行時設定でサンプル全体に追加されています。これで，VisiBroker インターセプタに慣れている場合も，VisiBroker インターセプタから OMB 固有のポータブルインターセプタへとすばやくコーディングを行うことができます。

必要なパッケージのインポート

ポータブルインターセプタインターフェースを使用するには，関連するパッケージまたはヘッダーファイルをインクルードする必要があります。

メモ　DuplicateName または InvalidName などのポータブルインターセプタ例外を使用する場合，ORBInitInfoPackage は省略可能です。

ポータブルインターセプタを使用するために必要なヘッダーファイル

```
#include "PortableInterceptor_c.hh"
#include "IOP_c.hh"
```

クライアント側のリクエストインターセプタをロードするには，ORBInitializer インターフェースを使用するクラスを実装する必要があります。これは，初期化に関する限り，サーバー側のリクエストインターセプタについても同じです。次は，これを実行するコードです。

サーバーリクエストインターセプタをロードするために必要な ORBInitializer の適切な継承

```
class SampleServerLoader : public PortableInterceptor::ORBInitializer
```

メモ　インターフェース ORBInitializer を実装する各オブジェクトも，LocalObject オブジェクトからそれぞれ継承する必要があります。これは ORBInitializer の IDL 定義がキーワード local を使用するためです。

IDL キーワード local の詳細については，第 29 章「valuetype の使い方」を参照してください。

ORB の初期化中，各リクエストインターセプタは pre_init() インターフェースのインプリメンテーションを介して追加されます。このインターフェースの中では，メソッド add_client_request_interceptor() を介してクライアントリクエストインターセプタが追加されています。関連するクライアントリクエストインターセプタは，インスタンス化してから ORB に追加する必要があります。

クライアント側リクエストインターセプタの初期化と ORB への登録

```
public: void pre_init(PortableInterceptor::ORBInitInfo_ptr _info) {
            SampleClientInterceptor *interceptor = new SampleClientInterceptor;
               try {
                  _info->add_client_request_interceptor(interceptor);
                  ...
```

OMG 仕様によると，必要なアプリケーションは register_orb_initializer メソッドを介して各インターセプタを登録します。詳細については，334 ページの「クライアントおよびサーバーアプリケーションの開発」を参照してください。

VisiBroker では，ダイナミックリンクライブラリ（Dynamic Link Library : DLL）などのオプションの方法を使用して，各インターセプタを登録できます。この登録方法を使用するメリットは，アプリケーションでコードを変更する必要がなく，実行方法だけを変更すればよいことです。実行中にほかのオプションを使用して，インターセプタを登録して実行できます。このオプションは，4.x インターセプタと同じです。

```
vbroker.orb.dynamicLibs=<DLL filename>
```

<DLL filename> は，ダイナミックリンクライブラリ（UNIX の場合は .SO，Windows の場合は .DLL 拡張子）のファイル名です。複数の DLL ファイルをロードするには，各ファイル名をカンマ（,）で区切ります。

Windows :

```
vbroker.orb.dynamicLibs=a.dll,b.dll,c.dll
```

UNIX :

```
vbroker.orb.dynamicLibs=a.so,b.so,c.so
```

インターセプタを動的にロードするには，VISInit インターフェースを使用します。これは，VisiBroker のインターセプタで使用されるインターフェースに似ています。詳細については，第 25 章「VisiBroker インターセプタの使い方」を参照してください。ORB_init のインプリメンテーション内にある各インターセプタローダーの登録は，よく似ています。

DLL ロードによるクライアント側 ORBInitializer の登録

```
void ORB_init(int& argc, char* const* argv, CORBA::ORB_ptr orb)
{
   if( _bind_interceptors_installed) return;

   SampleClientLoader *client = new SampleClientLoader();
   PortableInterceptor::register_orb_initializer(client);

   ...
```

クライアント側インターセプタローダーの完全なインプリメンテーション

```
// SampleClientLoader.C

#include "PortableInterceptor_c.hh"
#include "IOP_c.hh"

#include "SampleClientInterceptor.h"

#if !defined( DLL_COMPILE )
#include "vinit.h"
#include "corba.h"
#endif

// USE_STD_NS は，std 名前空間 USE_STD_NS を使用するために VisiBroker によって設定され
る定義です

class SampleClientLoader :
   public PortableInterceptor::ORBInitializer
{
private:
   short int _interceptors_installed;

   #if defined( DLL_COMPILE )
   static SampleClientLoader _instance;
   #endif

public:
   SampleClientLoader() {
         _interceptors_installed = 0;
   }

   void pre_init(PortableInterceptor::ORBInitInfo_ptr _info) {
         if(_interceptors_installed) return;

         cout << "=====>SampleClientLoader: Installing ..." << endl;

         SampleClientInterceptor *interceptor = new SampleClientInterceptor;

         try {
               _info->add_client_request_interceptor(interceptor);

               _interceptors_installed = 1;
                 cout << "=====>SampleClientLoader: Interceptors loaded."
                      << endl;
         }
         catch(PortableInterceptor::ORBInitInfo::DuplicateName &e) {
            cout << "=====>SampleClientLoader: "
                << e.name << " already installed!" << endl;
         }
         catch(...) {
            cout << "=====>SampleClientLoader: other exception occurred!"
```

```
                << endl;
            }
        }

        void post_init(PortableInterceptor::ORBInitInfo_ptr _info) {
    }
};

#if defined( DLL_COMPILE )

class VisiClientLoader : VISInit
{
private:
      static VisiClientLoader _instance;
      short int _bind_interceptors_installed;

public:
   VisiClientLoader() : VISInit(1) {
   _bind_interceptors_installed = 0;
   }

   void ORB_init(int& argc, char* const* argv, CORBA::ORB_ptr orb) {
         if( _bind_interceptors_installed) return;

         try {
               SampleClientLoader *client = new SampleClientLoader();
               PortableInterceptor::register_orb_initializer(client);

               _bind_interceptors_installed = 1;
         }
         catch(const CORBA::Exception& e)
         {
            cerr << e << endl;
         }
   }
};

// 静的インスタンス
VisiClientLoader VisiClientLoader::_instance;

#endif
```

サーバー側インターセプタの ORBInitializer の実装

この段階では，クライアントリクエストインターセプタが適切にインスタンス化され，追加されている必要があります。その後に続くコードは，例外の処理と結果の表示だけを行います。同様に，サーバー側では，サーバーリクエストインターセプタも同じ方法で実行されますが，ORB の関連サーバーリクエストインターセプタを追加する際にメソッド add_server_request_interceptor() を使用します。

サーバー側リクエストインターセプタの初期化と ORB への登録

```
public void pre_init(PortableInterceptor::ORBInitInfo_ptr _info) {
             SampleServerInterceptor *interceptor = new SampleServerInterceptor;
             try {
                 _info->add_server_request_interceptor(interceptor);
                 ...
```

これは, DLL インプリメンテーションを介したサーバー側の ORBInitializer クラスをロードする場合にも同じように適用されます。

DLL を介したサーバー側要求 ORB 初期化子のロード

```
void ORB_init(int& argc, char* const* argv, CORBA::ORB_ptr orb)
{
   if( _poa_interceptors_installed) return;

   SampleServerLoader *server = new SampleServerLoader();
   PortableInterceptor::register_orb_initializer(server);
   ...
```

サーバー側インターセプタローダーの完全なインプリメンテーション

```
// SampleServerLoader.C

#include "PortableInterceptor_c.hh"
#include "IOP_c.hh"
#if defined( DLL_COMPILE )
#include "vinit.h"
#include "corba.h"
#endif
#include "SampleServerInterceptor.h"
 // USE_STD_NS は，std 名前空間 USE_STD_NS を使用するために VisiBroker によって
 // 設定される定義です

class SampleServerLoader :

   public PortableInterceptor::ORBInitializer
{
private:
   short int _interceptors_installed;

public:
   SampleServerLoader() {
      _interceptors_installed = 0;
   }
 void pre_init(PortableInterceptor::ORBInitInfo_ptr _info) {

   if(_interceptors_installed) return;

   cout << "=====>SampleServerLoader: Installing ..." << endl;

   SampleServerInterceptor *interceptor = new SampleServerInterceptor();

   try {
         _info->add_server_request_interceptor(interceptor);

 _interceptors_installed = 1;

         cout << "=====>SampleServerLoader: Interceptors loaded."
             << endl;
   }
   catch(PortableInterceptor::ORBInitInfo::DuplicateName &e) {
      cout << "=====>SampleServerLoader: "
          << e.name << " already installed!" << endl;
   }
   catch(...) {
      cout << "=====>SampleServerLoader: other exception occurred!"
          << endl;
      }
   }
 void post_init(PortableInterceptor::ORBInitInfo_ptr _info) {}
};
```

```
 #if defined( DLL_COMPILE )  class VisiServerLoader : VISInit

{
private:
   static VisiServerLoader _instance;
   short int _poa_interceptors_installed;

public:
   VisiServerLoader() : VISInit(1) {
   _poa_interceptors_installed = 0;
}
 void ORB_init(int& argc, char* const* argv, CORBA::ORB_ptr orb) {
      if( _poa_interceptors_installed) return;
 try {
           SampleServerLoader *server = new SampleServerLoader();
           PortableInterceptor::register_orb_initializer(server);
 _poa_interceptors_installed = 1;
      }
   catch(const CORBA::Exception& e)
      {
      cerr << e << endl;
      }


 } };  // 静的インスタンス

VisiServerLoader VisiServerLoader::_instance;

 #endif
```

クライアント側またはサーバー側のリクエストインターセプタの RequestInterceptor の実装

クライアント側かサーバー側のリクエストインターセプタのどちらかを実装する場合は，その他に 2 つのインターフェースを実装する必要があります。それは name() と destroy() です。

name() は，正しいインターセプタを識別するために ORB に名前を提供するために必要です。このメソッドは，あらゆる要求や応答中にロードされて呼び出されます。CORBA 仕様によると，インターセプタは匿名でもかまいません。その場合，name 属性に空の文字列を指定します。このサンプルでは，「SampleClientInterceptor」という名前がクライアント側のインターセプタに，「SampleServerInterceptor」という名前がサーバー側のインターセプタに割り当てられています。

インターフェース属性，読み取り専用属性名のインプリメンテーション

```
public: char *name(void) {
            return _name;
         }
```

クライアントの ClientRequestInterceptor 実装

リクエストインターセプタが適切に動作するように，クライアントリクエストインターセプタに ClientRequestInterceptor インターフェースを実装する必要があります。

クラスでインターフェースを実装する場合は，インプリメンテーションのタイプに関係なく，次の 5 つのリクエストインターセプタのメソッドを実装する必要があります。

- send_request()

- send_poll()
- receive_reply()
- receive_exception()
- receive_other()

また，リクエストインターセプタのインターフェースについては，事前に実装しておく必要があります。クライアント側のインターセプタでは，次に示す要求インターセプトポイントがそのイベントに関連してトリガーされます。

send_request：要求がサーバーに送信される前にインターセプタが要求情報を照会し，サービスコンテキストを変更できるインターセプトポイントを提供します。

public void send_request(ClientRequestInfo ri) インターフェースのインプリメンテーション

```
void send_request(PortableInterceptor::ClientRequestInfo_ptr ri) {
        ...
```

void send_poll(ClientRequestInfo ri) インターフェースのインプリメンテーション

send_poll：TII（Time-Independent Invocation）ポーリングが応答シーケンスを取得する間にインターセプタが要求情報を照会するためのインターセプトポイントです。

```
void send_poll(PortableInterceptor::ClientRequestInfo_ptr ri) {
         ...
```

void receive_reply(ClientRequestInfo ri) インターフェースのインプリメンテーション

receive_reply：サーバーから応答が返され，制御がクライアントに戻るまでの間にインターセプタが情報を照会するポイントです。

```
void receive_reply(PortableInterceptor::ClientRequestInfo_ptr ri) {
        ...
```

void receive_exception(ClientRequestInfo ri) インターフェースのインプリメンテーション

receive_exception：例外の情報がクライアントで生成される前に，インターセプタがその情報を照会できるインターセプトポイントを提供します。

```
void receive_exception(PortableInterceptor::ClientRequestInfo_ptr ri) {
        ...
```

receive_other：インターセプタは通常の応答や例外以外の要求の結果に関する情報を照会します。たとえば，LOCATION_FORWARD ステータスの GIOP Reply を受信した場合は要求が再試行されます。また，非同期の呼び出しでは，要求の後すぐに応答が返されるとは限りません。制御はクライアントに戻り，終了インターセプトポイントが呼び出されます。

```
void receive_other(PortableInterceptor::ClientRequestInfo_ptr ri) {
       ...
```

クライアント側リクエストインターセプタの完全なインプリメンテーションは次のとおりです。

```
// SampleClientInterceptor.h

#include "PortableInterceptor_c.hh"
#include "IOP_c.hh"

// USE_STD_NS は，std 名前空間 USE_STD_NS を使用するために VisiBroker によって設定され
る定義です
```

```
class SampleClientInterceptor :
   public PortableInterceptor::ClientRequestInterceptor
{
private:
   char *_name;

   void init(char *name) {
      _name = new char[strlen(name)+1];
      strcpy(_name, name);
   }

public:
   SampleClientInterceptor(char *name) {
      init(name);
         }

   SampleClientInterceptor() {
      init("SampleClientInterceptor");
   }

char *name(void) {
   return _name;
}

void destroy(void) {
   // ここでは何もしません
   cout << "=====>SampleServerLoader: Interceptors unloaded" << endl;
}

/**
* これは，VisiBroker 4.x ClientRequestInterceptor の次のメソッドに似ています。
*
* void preinvoke_premarshal(CORBA::Object_ptr target,
*                          const char* operation,
*                          IOP::ServiceContextList&servicecontexts,
*                          VISClosure& closure) = 0;
*/
void send_request(PortableInterceptor::ClientRequestInfo_ptr ri) {
   cout << "=====> SampleClientInterceptor id " << ri->request_id()
       << " send_request => " << ri->operation()
       << ": Target = " << ri->target()
       << endl;
}

/**
* VisiBroker 4.x ClientRequestInterceptor に
* 対応するインターフェースはありません。
*/

void send_poll(PortableInterceptor::ClientRequestInfo_ptr ri) {
   cout << "=====> SampleClientInterceptor id " << ri->request_id()
       << " send_poll => " << ri->operation()
       << ": Target = " << ri->target()
        << endl;
}

/**
* これは，VisiBroker 4.x ClientRequestInterceptor の次のメソッドに似ています。
```

```
*
* void postinvoke(CORBA::Object_ptr target,
*                const IOP::ServiceContextList& service_contexts,
*                CORBA_MarshalInBuffer& payload,
*                CORBA::Environment_ptr env,
*                VISClosure& closure) = 0;
*
* with env not holding any exception value.
*/
void receive_reply(PortableInterceptor::ClientRequestInfo_ptr ri){
   cout << "=====> SampleClientInterceptor id " << ri->request_id()
       << " receive_reply => " << ri->operation()
       << endl;
}

/**
* これは，VisiBroker 4.x ClientRequestInterceptor の次のメソッドに似ています。
*
* void postinvoke(CORBA::Object_ptr target,
*                const IOP::ServiceContextList& service_contexts,
*                CORBA_MarshalInBuffer& payload,
*                CORBA::Environment_ptr env,
*                VISClosure& closure) = 0;
*
* with env holding the exception value.
*/
void receive_exception(PortableInterceptor::ClientRequestInfo_ptr ri) {
   cout << "=====> SampleClientInterceptor id " << ri->request_id()
       << " receive_exception => " << ri->operation()
        << ": Exception = " << ri->received_exception()
       << endl;
}

/**
* これは，VisiBroker 4.x ClientRequestInterceptor の次のメソッドに似ています。
*
* void postinvoke(CORBA::Object_ptr target,
*                const IOP::ServiceContextList& service_contexts,
*                CORBA_MarshalInBuffer& payload,
*                CORBA::Environment_ptr env,
*                VISClosure& closure) = 0;
*
* with env holding the exception value.
*/
void receive_other(PortableInterceptor::ClientRequestInfo_ptr ri) {
   cout << "=====> SampleClientInterceptor id " << ri->request_id()
       << " receive_other => " << ri->operation()
       << ": Exception = " << ri->received_exception()
       << ", Reply Status = " <<
     getReplyStatus(ri->reply_status())
      << endl;
   }

protected:
   char *getReplyStatus(CORBA::Short status) {
      if(status == PortableInterceptor::SUCCESSFUL)
            return "SUCCESSFUL";
      else if(status == PortableInterceptor::SYSTEM_EXCEPTION)
            return "SYSTEM_EXCEPTION";
      else if(status == PortableInterceptor::USER_EXCEPTION)
```

```
                return "USER_EXCEPTION";
            else if(status == PortableInterceptor::LOCATION_FORWARD)
                return "LOCATION_FORWARD";
            else if(status == PortableInterceptor::TRANSPORT_RETRY)
                return "TRANSPORT_RETRY";
            else
                return "invalid reply status id";
            }
    };
```

サーバー側のインターセプタでは，次のような要求インターセプトポイントが，そのイベントに関連してトリガーされます。

receive_request_service_contexts：インターセプタは着信要求からサービスのコンテキスト情報を取得し，PortableInterceptor::Current のスロットに転送できるインターセプトポイントを提供します。このインターセプトポイントは，サーバントマネージャの前に呼び出されます。詳細については，第 9 章「POA の使い方」の「サーバントとサーバントマネージャの使い方」を参照してください。

void receive_request_service_contexts (ServerRequestInfo ri) インターフェースのインプリメンテーション

```
void receive_request_service_contexts(PortableInterceptor::ServerRequestInfo_ptr ri) {
        ...
```

receive_request：インターセプタはオペレーションのパラメータなど，すべての有効な情報を照会できるインターセプトポイントを提供します。

void receive_request (ServerRequestInfo ri) インターフェースのインプリメンテーション

```
void receive_request(PortableInterceptor::ServerRequestInfo_ptr ri) {
        ...
```

send_reply：ターゲットオペレーションを呼び出してその応答がサーバーに戻される前に，インターセプタは応答情報を照会して応答のサービスコンテキストを変更できるインターセプトポイントを提供します。

void receive_reply (ServerRequestInfo ri) インターフェースのインプリメンテーション

```
void send_reply(PortableInterceptor::ServerRequestInfo_ptr ri) {
        ...
```

send_exception：クライアントで例外が生成される前にインターセプタが例外情報を照会し，応答のサービスコンテキストを変更できるインターセプトポイントを提供します。

void receive_exception (ServerRequestInfo ri) インターフェースのインプリメンテーション

```
void send_exception(PortableInterceptor::ServerRequestInfo_ptr ri) {
        ...
```

send_other：インターセプタは通常の応答や例外以外の要求の結果に関する情報を照会します。たとえば，LOCATION_FORWARD ステータスの GIOP Reply を受信した場合は要求が再試行されます。また，非同期の呼び出しでは，要求の後すぐに応答が返されるとは限りません。制御はクライアントに戻り，終了インターセプトポイントが呼び出されます。

void receive_other (ServerRequestInfo ri) インターフェースのインプリメンテーション

```
void send_other(PortableInterceptor::ServerRequestInfo_ptr ri) {
          ...
```

どのインターセプトポイントを使用しても，クライアントとサーバーの両方で異なる呼び出しポイントにある異なるタイプの情報を取得できます。次のサンプルでは，これらの情報がデバッグとして画面に表示されています。

サーバー側リクエストインターセプタの完全なインプリメンテーションは次のとおりです。

```
// SampleServerInterceptor.h

#include "PortableInterceptor_c.hh"
#include "IOP_c.hh"

// USE_STD_NS は，std 名前空間 USE_STD_NS を使用するために VisiBroker によって設定される定義です

class SampleServerInterceptor :
   public PortableInterceptor::ServerRequestInterceptor
{
private:
   char *_name;

   void init(char *name) {
      _name = new char[strlen(name)+1];
      strcpy(_name, name);
   }

public:
   SampleServerInterceptor(char *name) {
      init(name);
   }

   SampleServerInterceptor() {
      init("SampleServerInterceptor");
}

   char *name(void) {
      return _name;
}

   void destroy(void) {
      // ここでは何もしません
      cout << "=====>SampleServerLoader: Interceptors unloaded" << endl;
   }
/**
* これは，VisiBroker 4.x ClientRequestInterceptor の次のメソッドに似ています。
*
* void preinvoke_premarshal(CORBA::Object_ptr target,
*                           const char* operation,
*                           IOP::ServiceContextList& servicecontexts,
*                           VISClosure& closure) = 0;
*/
void
receive_request_service_contexts(PortableInterceptor::ServerRequestInfo_ptr
ri) {
         cout << "=====> SampleServerInterceptor id " << ri->request_id()
```

```
            << " receive_request_service_contexts => " << ri->operation()
            << endl;
   }

/**
* VisiBroker 4.x SeverRequestInterceptor に
* 対応するインターフェースはありません。
*/
void receive_request(PortableInterceptor::ServerRequestInfo_ptr ri)
{
   cout << "=====> SampleServerInterceptor id " << ri->request_id()
       << " receive_request => " << ri->operation()
       << ": Object ID = " << ri->object_id()
       << ", Adapter ID = " << ri->adapter_id()
       << endl;
}

/**
* VisiBroker 4.x SeverRequestInterceptor に
* 対応するインターフェースはありません。
*/
void send_reply(PortableInterceptor::ServerRequestInfo_ptr ri) {
   cout << "=====> SampleServerInterceptor id " << ri->request_id()
       << " send_reply => " << ri->operation()
      << endl;
}

/**
* これは, VisiBroker 4.x ServerRequestInterceptor の次のメソッドに似ています。
*
* virtual void postinvoke_premarshal(CORBA::Object_ptr _target,
*                                   IOP::ServiceContextList&_service_contexts,
*                                   CORBA::Environment_ptr _env,
*                                   VISClosure& _closure) = 0;
*
* with env holding the exception value.
*/
void send_exception(PortableInterceptor::ServerRequestInfo_ptr ri) {
   cout << "=====> SampleServerInterceptor id " << ri->request_id()
       << " send_exception => " << ri->operation()
       << ": Exception = " << ri->sending_exception()
       << ", Reply status = " << getReplyStatus(ri->reply_status())
       << endl;
}

/**
* これは, VisiBroker 4.x ServerRequestInterceptor の次のメソッドに似ています。
*
* virtual void postinvoke_premarshal(CORBA::Object_ptr _target,
*                                   IOP::ServiceContextList& _service_contexts,
*                                   CORBA::Environment_ptr _env,
*                                   VISClosure& _closure) = 0;
*
* with env holding the exception value.
*/
void send_other(PortableInterceptor::ServerRequestInfo_ptr ri) {
   cout << "=====> SampleServerInterceptor id " << ri->request_id()
       << " send_other => " << ri->operation()
       << ": Exception = " << ri->sending_exception()
```

```
            << ", Reply Status = " << getReplyStatus(ri->reply_status())
            << endl;
      }

   protected:
      char *getReplyStatus(CORBA::Short status) {
         if(status == PortableInterceptor::SUCCESSFUL)
            return "SUCCESSFUL";
               else if(status == PortableInterceptor::SYSTEM_EXCEPTION)
            return "SYSTEM_EXCEPTION";
               else if(status == PortableInterceptor::USER_EXCEPTION)
            return "USER_EXCEPTION";
               else if(status == PortableInterceptor::LOCATION_FORWARD)
            return "LOCATION_FORWARD";
               else if(status == PortableInterceptor::TRANSPORT_RETRY)
            return "TRANSPORT_RETRY";
               else
            return "invalid reply status id";
         }
   };
```

クライアントおよびサーバーアプリケーションの開発

インターセプタのクラスを記述したら，それぞれのクライアントアプリケーションとサーバーアプリケーションに登録します。

C++ の場合，クライアントとサーバーは，PortableInterceptor::register_orb_initalizer(<class_name>) メソッドを使用して，それぞれの ORBInitializer クラスを登録します。ここで，<class_name> は登録されるクラス名です。

サンプルでは，別の方法を使ってインターセプタクラスをダイナミックリンクライブラリ（DLL : Dynamic Link Library）として登録する方法についても説明します。この登録方法を使用するメリットは，実行時の変更が必要だが，アプリケーションで何もコードを変更する必要がないことです。

メモ これは VisiBroker 独自の登録方法です。OMG に完全に準拠するには，次の方法は使用しないでください。

インターセプタクラスを DLL としてロードする VisiBroker 独自の方法を使用する場合は，クライアントアプリケーションとサーバーアプリケーションにコードを追加する必要はありません。サンプルでは，DLL のコンパイルとリンクが指定されていない場合，マクロを介してコードの一部を無効にできます。

クライアントアプリケーションのインプリメンテーション

```
// Client.C

#include "Bank_c.hh"

// USE_STD_NS は，std 名前空間 USE_STD_NS を使用するために VisiBroker によって設定され
る定義です

#if !defined( DLL_COMPILE )
#include "SampleClientLoader.C"
#endif

int main(int argc, char* const* argv)
{
    try {
        // ロードの DLL メソッドを使用しない場合は，
        // ORB を初期化する「前に」ローダーをインスタンス化します
        #if !defined( DLL_COMPILE )
```

```
        SampleClientLoader* loader = new SampleClientLoader;
        PortableInterceptor::register_orb_initializer(loader);
        #endif

        // ORB を初期化します。
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

        // マネージャの ID を取得します。
        PortableServer::ObjectId_var managerId =
                        PortableServer::string_to_ObjectId("BankManager");

        // AccountManager を検索します。完全な POA 名とサーバント ID を指定します。
        Bank::AccountManager_var manager =
                  Bank::AccountManager::_bind("/bank_agent_poa", managerId);

        // 口座名として argv[1], またはデフォルトを使用します。
        const char* name = argc > 1 ? argv[1] : "Jack B. Quick";

        // アカウントマネージャに指定した口座を開くように要求します。
        Bank::Account_var account = manager->open(name);

        // 口座の残高を取得します。
        CORBA::Float balance = account->balance();

        // 残高を印刷します。
        cout << "The balance in " << name << "'s account is $" << balance
             << endl;
    }
    catch(const CORBA::Exception& e) {
        cerr << e << endl;
        return 1;
    }

    return 0;
}
```

サーバーアプリケーションのインプリメンテーション

```
// Server.C

#include "BankImpl.h"

// USE_STD_NS は, std 名前空間 USE_STD_NS を使用するために VisiBroker によって設定され
る定義です

#if !defined( DLL_COMPILE )
#include "SampleServerLoader.C"
#endif

int main(int argc, char* const* argv)
{
    try {
        // ORB を初期化する前にインターセプタローダーをインスタンス化します。
        #if !defined( DLL_COMPILE )
        SampleServerLoader* loader = new SampleServerLoader();
        PortableInterceptor::register_orb_initializer(loader);
        #endif

        // ORB を初期化します。
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

        // ルート POA へのリファレンスを取得します。
        CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
```

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

        CORBA::PolicyList policies;
        policies.length(1);
        policies[(CORBA::ULong)0] = rootPOA->create_lifespan_policy(
                                        PortableServer::PERSISTENT);

        // POA マネージャを取得します。
        PortableServer::POAManager_var poa_manager = rootPOA->the_POAManager();

        // 適切なポリシーで myPOA を作成します。
        PortableServer::POA_var myPOA = rootPOA->create_POA("bank_agent_poa",
                                                    poa_manager,
                                                    policies);
        // サーバントを作成します。
        AccountManagerImpl managerServant;

        // サーバントの ID を決定します。
        PortableServer::ObjectId_var managerId =
                    PortableServer::string_to_ObjectId("BankManager");

        // その ID を使って myPOA でサーバントをアクティブ化します。
        myPOA->activate_object_with_id(managerId, &managerServant);

        // POA マネージャをアクティブ化します。
        poa_manager->activate();

        CORBA::Object_var reference =
myPOA->servant_to_reference(&managerServant);
        cout << reference << " is ready" << endl;

        // 着信要求を待機します。
        orb->run();
    }
    catch(const CORBA::Exception& e) {
        cerr << e << endl;
        return 1;
    }
    return 0;
}
```

コンパイルの手順

VisiBroker 独自のロード方法を使用せずに C++ のサンプルをコンパイルするには，次のコマンドを実行します。

Windows :

```
<install_dir>¥examples¥vbe¥pi¥client_server> nmake -f Makefile.cpp
```

UNIX :

```
<install_dir>/examples/vbe/pi/client_server> make -f Makefile.cpp
```

VisiBroker 独自のロード方法を使って C++ のサンプルをコンパイルするには，次のコマンドを実行します。

Windows :

```
<install_dir>¥examples¥vbe¥pi¥client_server>
 nmake -f Makefile.cpp dll
```

UNIX :

```
<install_dir>/examples/vbe/pi/client_server>
 make -f Makefile.cpp dll
```

クライアントアプリケーションとサーバーアプリケーションの実行と配布

VisiBroker 独自のロード方法を使用せずに C++ のサンプルを実行するには，次の手順にしたがってサーバーとクライアントを起動します。

Windows：

```
<install_dir>¥examples¥vbe¥pi¥client_server> start Server (running under a new command
prompt window)
<install_dir>¥examples¥vbe¥pi¥client_server> Client John (using a given name)
```

または

```
<install_dir>¥examples¥vbe¥pi¥client_server> Client (using a default name)
```

UNIX： 2 つのコンソールシェルを開きます。

```
<install_dir>/examples/vbe/pi/client_server> Server(in the first window)
<install_dir>/examples/vbe/pi/client_server> Client John (in the second window, using a
given name)
```

または

```
<install_dir>/examples/vbe/pi/client_server> Client (in the second window, using the
default name)
```

VisiBroker 独自のロード方法を使って C++ のサンプルを実行するには，次の手順にしたがってサーバーとクライアントを起動します。

Windows：

```
<install_dir>¥examples¥vbe¥pi¥client_server> start Server -Dvbroker.orb.dynamicLibs=
SampleServerLoader.dll(running under a new command prompt window)
<install_dir>¥examples¥vbe¥pi¥client_server> Client John -Dvbroker.orb.dynamicLibs=
SampleClientLoader.dll(using a given name)
```

または

```
<install_dir>¥examples¥vbe¥pi¥client_server> Client -Dvbroker.orb.dynamicLibs=
SampleClientLoader.dll (using a default name)
```

UNIX： 2 つのコンソールシェルを開きます。

```
<install_dir>/examples/vbe/pi/client_server> Server
-Dvbroker.orb.dynamicLibs=./SampleServerLoader.so (in the first window)
<install_dir>/examples/vbe/pi/client_server> Client
-Dvbroker.orb.dynamicLibs=./SampleClientLoader.so John (in the second window, using a
given name)
```

第 25 章

VisiBroker インターセプタの使い方

ここでは，VisiBroker インターセプタ（インターセプタ）というフレームワークの概要について説明します。インターセプタのサンプルを紹介し，インターセプタファクトリやインターセプタのチェイン化など，高度な機能についても解説します。また，ポータブルインターセプタと VisiBroker インターセプタの両方を同じサービスで使用した場合に予想される動作についても取り上げます。

インターセプタの概要

ポータブルインターセプタと同様に, VisiBroker インターセプタも VisiBroker ORB サービスに ORB の通常の実行の流れをインターセプトするメカニズムを提供します。VisiBroker インターセプタには，次の 2 種類があります。

- *クライアントインターセプタ*はシステムレベルのインターセプタで，クライアントオブジェクトのメソッドを起動すると呼び出されます。
- *サーバーインターセプタ*はシステムレベルのインターセプタで，サーバーオブジェクトのメソッドを起動すると呼び出されます。

VisiBroker インターセプタを使用するには，インターセプタインターフェースの 1 つを実装するクラスを宣言します。インターセプタオブジェクトをインスタンス化してから，対応するインターセプタマネージャにインスタンスを登録します。これにより，インターセプタオブジェクトはオブジェクトのメソッドの 1 つが起動された場合や，パラメータがマーシャリング／アンマーシャリングされた場合などに，インターセプタマネージャから通知を受けます。

VisiBroker インターセプタとポータブルインターセプタの重要な違いは，VisiBroker インターセプタが共用呼び出しのために起動できないことです。したがって，使用するインターセプタを選択する際は，注意が必要です。

メモ　クライアント側でマーシャリングされる前に，またはサーバー側で処理される前にオペレーションリクエストを捕捉するには，オブジェクトラッパーを使用します。オブジェクトラッパーの使用については，第 26 章「オブジェクトラッパーの使い方」を参照してください。

インターセプタインターフェースとインターセプタマネージャ

インターセプタの開発者は，次の 1 つまたは複数の基準インターセプタ API クラスからクラスを派生させます。これらの API クラスは，VisiBroker で定義し，実装します。

- クライアントインターセプタ：
  - BindInterceptor
  - ClientRequestInterceptor
- サーバーインターセプタ：
  - POALifeCycleInterceptor
  - ActiveObjectLifeCycleInterceptor
  - ServerRequestInterceptor
  - IORCreationInterceptor
- サービスリゾルバインターセプタ

クライアントインターセプタ

現在，次の 2 種類のクライアントインターセプタと，それぞれに対応したマネージャがあります。

- BindInterceptor と BindInterceptorManager
- ClientRequestInterceptor と ClientRequestInterceptorManager

クライアントインターセプタの詳細については，第 24 章「ポータブルインターセプタの使い方」を参照してください。

BindInterceptor

BindInterceptor オブジェクトは，グローバルなインターセプタであり，バインドの前と後にクライアント側で呼び出されます。

```
class _VISEXPORT BindInterceptor : public virtual VISPseudoInterface {
   public:
      virtual IOP::IORValue_ptr bind(IOP::IORValue_ptr ior,
            CORBA_Object_ptr obj,
            CORBA::Boolean rebind,
            VISClosure& closure) = 0;
      virtual IOP::IORValue_ptr bind_failed(IOP::IORValue_ptr ior,
            CORBA_Object_ptr object,
            VISClosure& closure) = 0;
      virtual void bind_succeeded(IOP::IORValue_ptr ior,
            CORBA_Object_ptr object,
            CORBA::Long profile_index,
            interceptor::InterceptorManagerControl_ptr control,
            VISClosure& closure) = 0;
      virtual void exception_occurred(IOP::IORValue_ptr ior,
            CORBA_Object_ptr object,
            CORBA_Environment_ptr env,
            VISClosure& closure) = 0;
};
```

ClientRequestInterceptor

ClientRequestInterceptor オブジェクトは，BindInterceptor オブジェクトの bind_succeeded 呼び出し中に登録され，接続が終了するまでアクティブな状態を維持します。そのメソッドのうちの 2 つは，クライアントオブジェクト側での実行前に呼び出されます。そのうちの 1 つ，preinvoke_premarshal はパラメータがマーシャリングされる前に呼び出され，もう 1 つのメソッド，preinvoke_postmarshal はマーシャリング後に呼び出されます。3 番めのメソッド，postinvoke は要求完了後に呼び出されます。

```
class _VISEXPORT ClientRequestInterceptor : public virtual VISPseudoInterface {
   public:
      virtual void preinvoke_premarshal(CORBA::Object_ptr target,
            const char* operation,
            IOP::ServiceContextList& servicecontexts,
            VISClosure& closure) = 0;
      virtual void preinvoke_postmarshal(CORBA::Object_ptr target,
            CORBA_MarshalInBuffer& payload,
            VISClosure& closure) = 0;
      virtual void postinvoke(CORBA::Object_ptr target,
            const IOP::ServiceContextList& service_contexts,
            CORBA_MarshalInBuffer& payload,
            CORBA::Environment_ptr env,
            VISClosure& closure) = 0;
      virtual void exception_occurred(CORBA::Object_ptr target,
            CORBA::Environment_ptr env,
            VISClosure& closure) = 0;
};
```

サーバーインターセプタ

次の種類のサーバーインターセプタがあります。

- POALifeCycleInterceptor と POALifeCycleInterceptorManager
- ActiveObjectLifeCycleInterceptor と ActiveObjectLifeCycleInterceptorManager
- ServerRequestInterceptor と ServerRequestInterceptorManager
- IORCreationInterceptor と IORCreationInterceptorManager

サーバーインターセプタの詳細については，第 24 章「ポータブルインターセプタの使い方」を参照してください。

POALifeCycleInterceptor

POALifeCycleInterceptor オブジェクトはグローバルなインターセプタであり，POA が（create メソッドを使って）作成されたり，（destroy メソッドを使って）破棄されるたびに呼び出されます。

```
class _VISEXPORT POALifeCycleInterceptor : public virtual VISPseudoInterface {
   public:
      virtual void create(PortableServer::POA_ptr _poa,
            CORBA::PolicyList& _policies,
            IOP::IORValue*& _iorTemplate,
            interceptor::InterceptorManagerControl_ptr _poaAdmin) = 0;
      virtual void destroy(PortableServer::POA_ptr _poa) = 0;
};
```

ActiveObjectLifeCycleInterceptor

ActiveObjectLifeCycleInterceptor オブジェクトは，create メソッドでアクティブオブジェクトマップにオブジェクトを追加するとき，またはオブジェクトが非アクティブ化され，destroy メソッドで霊化された後に呼び出されます。POALifeCycleInterceptor が POA の作成時に，このインターセプタを POA 単位で登録します。このインターセプタは，POA に RETAIN ポリシーがある場合にだけ登録されます。

```
class _VISEXPORT ActiveObjectLifeCycleInterceptor : public virtual VISPseudoInterface {
   public:
      virtual void create(const PortableServer::ObjectId& _oid,
            PortableServer_ServantBase* _servant,
            PortableServer::POA_ptr _adapter) = 0;
      virtual void destroy(const PortableServer::ObjectId& _oid,
            PortableServer_ServantBase* _servant,
            PortableServer::POA_ptr _adapter) = 0;
};
```

ServerRequestInterceptor

ServerRequestInterceptor オブジェクトはリモートオブジェクトのサーバーインプリメンテーション呼び出しのさまざまな段階で呼び出されます。たとえば，呼び出しの前（preinvoke メソッドによる），および応答のマーシャリングの前後の呼び出しの後（それぞれ postinvoke_premarshal メソッドと postinvoke_postmarshal メソッドによる）などがあります。このインターセプタは，POA 作成時に POALifeCycleInterceptor オブジェクトで POA 単位の登録を行うことができます。

```
class _VISEXPORT ServerRequestInterceptor : public virtual VISPseudoInterface {
   public:
      virtual void preinvoke(CORBA::Object_ptr _target, const char* _operation,
            const IOP::ServiceContextList& _service_contexts,
            CORBA_MarshalInBuffer& _payload,
            VISClosure& _closure) = 0;
      virtual void postinvoke_premarshal(CORBA::Object_ptr _target,
            IOP::ServiceContextList& _service_contexts,
            CORBA::Environment_ptr _env,
            VISClosure& _closure) = 0;
      virtual void postinvoke_postmarshal(CORBA::Object_ptr _target,
            CORBA_MarshalOutBuffer& _payload,
            VISClosure& _closure) = 0;
      virtual void exception_occurred(CORBA::Object_ptr _target,
            CORBA::Environment_ptr _env,
            VISClosure& _closure) = 0;
};
```

メモ CORBA::SystemException または任意のサブクラス（CORBA::NO_PERMISSION など）がサーバー側で発生した場合，例外を暗号化することはできません。これは，ORB が一部の例外を内部的に使用するためです（自動リバインドを実行するための TRANSIENT など）。

IORCreationInterceptor

IORCreationInterceptor オブジェクトは，POA によって create でオブジェクトリファレンスが作成されるたびに呼び出されます。このインターセプタは，POA 作成時に POALifeCycleInterceptor で POA 単位の登録を行うことができます。

```
class _VISEXPORT IORCreationInterceptor : public virtual VISPseudoInterface {
   public:
      virtual void create(PortableServer::POA_ptr _poa,
            IOP::IORValue*& _ior) = 0;
};
```

サービスリゾルバインターセプタ

後で動的にロードできるユーザーサービスをインストールするために，このインターセプタを使用します。

```
class _VISEXPORT interceptor::ServiceResolverInterceptor :public virtual
VISPsuedoInterface {
   public:
      virtual CORBA::Object_ptr resolve(const char* _name) = 0;
};
class _VISEXPORT ServiceResolverInterceptorManager :public virtual InterceptorManager,
   public virtual VISPseudoInterface {
      public:
         virtual void add(const char* _name, ServiceResolverInterceptor_ptr
_interceptor) =
               0;
         virtual void remove(const char* _name) = 0;
};
```

`resolve_initial_references` を呼び出すと，インストールされたすべてのサービス上のリゾルバが呼び出されます。これで，resolve は適切なオブジェクトを返すことができます。

サービスイニシャライザを記述する場合は，サービスを追加できるように，`InterceptorManagerControl` の取得後に `ServiceResolver` を取得する必要があります。

VisiBroker ORB によるインターセプタの登録

各インターセプタインターフェースには，対応するインターセプタマネージャインターフェースがあります。これでインターセプタオブジェクトを VisiBroker ORB に登録します。インターセプタを登録するには，次のような操作を実行します。

1. パラメータ `VisiBrokerInterceptorControl` を指定し，`resolve_initial_references` メソッドを ORB オブジェクトで呼び出して，`InterceptorManagerControl` オブジェクトへのリファレンスを取得します。
2. InterceptorManagerControl オブジェクトの `get_manager` メソッドに渡す文字列値を表す文字列値の 1 つを次の表から選択し，`InterceptorManagerControl` オブジェクトの `get_manager` メソッドを呼び出します。このオブジェクトリファレンスは，対応するインターフェースに必ずキャストしてください。

表 25.1　InterceptorManagerControl オブジェクトの文字列値

値	対応するインターセプタインターフェース
ClientRequest	ClientRequestInterceptor
Bind	BindInterceptor
POALifeCycle	POALifeCycleInterceptor
ActiveObjectLifeCycle	ActiveObjectLifeCycleInterceptor
ServerRequest	ServerRequestInterceptor
IORCreation	IORCreationInterceptor
ServiceResolver	ServiceResolverInterceptor

3. インターセプタのインスタンスを作成します。
4. `add` メソッドを呼び出して，インターセプタオブジェクトをマネージャオブジェクトに登録します。
5. クライアントプログラムとサーバープログラムの実行中に，インターセプタオブジェクトをロードします。

インターセプタオブジェクトの作成

最後に，ファクトリクラスを実装する必要があります。ファクトリクラスは，インターセプタのインスタンスを作成し，それを VisiBroker ORB に登録します。ファクトリクラスは，VISInit クラスを継承する必要があります。

```
// vinit.h ファイル
class _VISEXPORT VISInit {
   public:
      VISInit();
      VISInit(CORBA::Long init_priority);
      virtual ~VISInit();
      // ORB_init は，CORBA::ORB_init() の最初の方で呼び出されます。
      virtual void ORB_init(int& /*argc*/,
            char* const* /*argv*/,
            CORBA_ORB* /*orb*/)
         {}
      // ORB_initialized は，CORBA::ORB_init() の最後で呼び出されます。
      virtual void ORB_initialized(CORBA_ORB* /*orb*/) {}
      // shutdown は，CORBA::ORB::shutdown() が呼び出されたとき，または
      // プロセスのシャットダウンが検出されたときに呼び出されます。
      virtual void ORB_shutdown() {}
      . . .
};
```

メモ　344 ページの「サンプルインターセプタ」のサンプルのように，ほかのインターセプタ内から，インターセプタの新しいインスタンスを作成し，それを VisiBroker ORB に登録することもできます。

インターセプタのロード

インターセプタをロードするには，アプリケーションの中で CORBA::ORB_init を呼び出す前にファクトリをインスタンス化するだけです。

サンプルインターセプタ

このサンプルインターセプタでは，インターセプタ API のすべてのメソッド（第 24 章「ポータブルインターセプタの使い方」を参照）を使用します。これにより，これらのメソッドの使い方や呼び出しのタイミングを理解できます。

サンプルコード

346 ページの「コードリスト」の各インターセプタ API メソッドは，標準出力に情報メッセージを出力する簡単な実装になっています。

以下のサンプルアプリケーションは，次のディレクトリにあります。

```
<install_dir>¥examples¥vbe¥interceptors¥
```

- active_object_lifecycle
- client_server
- ior_creation

クライアント／サーバーインターセプタのサンプル

サンプルプログラムを実行するには，通常どおりにファイルをコンパイルします。その後，次のようにサーバーとクライアントを起動します。

```
prompt>Server
prompt>Client John
```

メモ　VisiBroker 3.x の ServiceInit クラスは，ServiceLoader と ServiceResolverInterceptor の 2 つのインターフェースの実装に置き換えられています。その実行例については，「ServiceResolverInterceptor の例」を参照してください。

サンプルインターセプタの実行結果を次の表に示します。クライアントとサーバーによる実行結果が順に並んでいます。

表 25.2　サンプルインターセプタの実行結果

クライアント	サーバー
	============>SampleServerLoader: Interceptors loaded============> In POA /.Nothing to do.============> In POA bank_agent_poa, 1 ServerRequest interceptor installedStub [repository_id=IDL:Bank/AccountManager:1.0,key=ServiceId[service=/bank_agent_poa,id={11 bytes: [B][a][n][k][M][a][n][a][g][e][r]}]] is ready.
Bind Interceptors loaded==========> SampleBindInterceptor bind==========> SampleBindInterceptor bind_succeeded==========> SampleClientInterceptor id MyClientInterceptor preinvoke_premarshal=> open==========> SampleClientInterceptor id MyClientInterceptor preinvoke_postmarshal	
	============> SampleServerInterceptor id MyServerInterceptor preinvoke => openCreated john's account: Stub[repository_id=IDL:Bank/Account:1.0, key=TransientId[poaName=/,id={4 bytes: (0)(0)(0)(0)}, sec=0,usec=0]]
============> SampleClientInterceptor id MyClientInterceptor postinvoke============> SampleBindInterceptor bind============> SampleBindInterceptor bind_succeeded============> SampleClientInterceptor id MyClientInterceptor preinvoke_premarshal => balance ============> SampleClientInterceptor id MyClientInterceptor preinvoke_postmarshal	
	============> SampleServerInterceptor id MyServerInterceptor postinvoke_premarshal============> SampleServerInterceptor id MyServerInterceptor postinvoke_postmarshal
============> SampleClientInterceptor id MyClientInterceptor postinvoke The balance in john's account is $245.64	

OAD は実行していないので，bind 呼び出しは失敗し，サーバーは実行を続けます。クライアントは Account オブジェクトにバインドし，次に balance メソッドを呼び出します。この要求は，サーバーによって受信および処理され，その結果がクライアントに返されます。クライアントはその結果を出力します。

サンプルコードとその結果が示しているように，クライアントとサーバーのインターセプタはどちらも各プロセス開始時にインストールされます。インターセプタの登録については，346 ページの「SampleServerLoader」を参照してください。

コードリスト

SampleServerLoader

SampleServerLoader オブジェクトは，POALifeCycleInterceptor クラスのロードとオブジェクトのインスタンス化を担当します。このクラスは vbroker.orb.dynamicLibs によって VisiBroker ORB に動的にリンクされます。SampleServerLoader クラスには，init メソッドがあり，初期化時に VisiBroker ORB によって呼び出されます。このメソッドは，POALifeCycleInterceptor オブジェクトを作成して InterceptorManager に登録し，結果的にインストールする専用のメソッドです。

```
#include <iostream.h>
#include "vinit.h"
#include "SamplePOALifeCycleInterceptor.h"

class POAInterceptorLoader : VISInit {
   private:
      short int _poa_interceptors_installed;
   public:
      POAInterceptorLoader(){
         _poa_interceptors_installed = 0;
      }
      void ORB_init(int& argc, char* const* argv, CORBA::ORB_ptr orb) {
         if( _poa_interceptors_installed) return;
         cout << "Installing POA interceptors" << endl;
         SamplePOALifeCycleInterceptor *interceptor = new SamplePOALifeCycleInterceptor;
         // インターセプタマネージャの制御を取得します。
         CORBA::Object *object =
               orb->resolve_initial_references("VisiBrokerInterceptorControl");
         interceptor::InterceptorManagerControl_var control =
               interceptor::InterceptorManagerControl::_narrow(object);
         // POA マネージャを取得します。
         interceptor::InterceptorManager_var manager =
               control->get_manager("POALifeCycle");
         PortableServerExt::POALifeCycleInterceptorManager_var poa_mgr =
               PortableServerExt::POALifeCycleInterceptorManager::_narrow(manager);
         // POA インターセプタをリストに追加します。
         poa_mgr->add( (PortableServerExt::POALifeCycleInterceptor*)interceptor);
         cout << "POA interceptors installed" << endl;
         _poa_interceptors_installed = 1;
      }
};
```

SamplePOALifeCycleInterceptor

SamplePOALifeCycleInterceptor オブジェクトは，POA の作成や破棄のたびに呼び出されます。client_server のサンプルでは 2 つの POA を使用しているため，このインターセプタは，最初に rootPOA の作成時に，次に myPOA の作成時に合わせて 2 回呼び出されます。SampleServerInterceptor は，myPOA の作成時にだけインストールします。

```
#include "interceptor_c.hh"
#include "PortableServerExt_c.hh"
#include "IOP_c.hh"
#include "SampleServerInterceptor.h"

class SamplePOALifeCycleInterceptor : PortableServerExt::POALifeCycleInterceptor {
   public:
      void create( PortableServer::POA_ptr poa,
            CORBA_PolicyList& policies,
            IOP::IORValue_ptr& iorTemplate,
            interceptor::InterceptorManagerControl_ptr control) {
```

```
      if( strcmp( poa->the_name(),"bank_agent_poa") == 0 ) {
         // 要求レベルのインターセプタを追加します。
         SampleServerInterceptor* interceptor =
               new SampleServerInterceptor("MyServerInterceptor");
         // ServerRequest インターセプタマネージャを取得します。
         interceptor::InterceptorManager_var generic_manager =
               control->get_manager("ServerRequest");
         interceptor::ServerRequestInterceptorManager_var manager =
               interceptor::ServerRequestInterceptorManager::_narrow(
                     generic_manager);
         // インターセプタを追加します。
         manager->add( (interceptor::ServerRequestInterceptor*)interceptor);
         cout <<"============>In POA " << poa->the_name() <<
               ", 1 ServerRequest interceptor installed"<< endl;
      } else
         cout << "============>In POA " << poa->the_name() <<
               ". Nothing to do." << endl;
   }
   void destroy( PortableServer::POA_ptr poa) {
      // トレース用
      cout << "============> SamplePOALifeCycleInterceptor destroy" <<
            poa->the_name() << endl;
   }
};
```

SampleServerInterceptor

SampleServerInterceptor オブジェクトは，サーバーが要求を受信するか，それに応答するたびに起動されます。

```
#include <iostream.h>
#include "vclosure.h"
#include "interceptor_c.hh"
#include "IOP_c.hh"
// USE_STD_NS は，std 名前空間 USE_STD_NS を使用するために VisiBroker によって設定され
る定義です

class SampleServerInterceptor : interceptor::ServerRequestInterceptor {
   private:
      char * _id;
   public:
      SampleServerInterceptor( const char* id) {
         _id = new char[ strlen(id)];
         strcpy( _id,id);
      }
      ~SampleServerInterceptor() { _id = NULL;}
      void preinvoke( CORBA_Object* target,
            const char* operation,
            const IOP::ServiceContextList& service_contexts,
            CORBA_MarshalInBuffer& payload,
            VISClosure& closure) {
         closure.data = new char[ strlen(_id)];
         strcpy( (char*)(closure.data), _id);
         cout << "============> SampleServerInterceptor id " <<
               (char*)(closure.data) <<
               " preinvoke => " << operation << endl;
      }
      void postinvoke_premarshal( CORBA_Object* target,
            IOP::ServiceContextList& service_contexts,
            CORBA::Environment_ptr env,
            VISClosure& closure) {
```

```
            cout << "============> SampleServerInterceptor id " <<
                  (char*)(closure.data) <<
                  " postinvoke_premarshal " << endl;
         }
         void postinvoke_postmarshal(CORBA_Object* target,
               CORBA_MarshalOutBuffer& payload,
               VISClosure& closure) {
            cout << "============> SampleServerInterceptor id " <<
               (char*)(closure.data) <<
               " postinvoke_postmarshal " << endl;
         }
         void exception_occurred( CORBA_Object* target,
               CORBA::Environment_ptr env,
               VISClosure& closure) {
            cout << "============> SampleServerInterceptor id " <<
                  (char*)(closure.data) <<
                  " exception_occurred" << endl;
         }
};
```

SampleClientInterceptor

SampleClientInterceptor オブジェクトは，サーバーが要求を受信するか，それに応答するたびに起動されます。

```
#include <iostream.h>
#include "interceptor_c.hh"
#include "IOP_c.hh"
#include "vclosure.h"

class SampleClientInterceptor : public interceptor::ClientRequestInterceptor {
   private:
      char * _id;
   public:
      SampleClientInterceptor( char * id) {
         _id = new char[ strlen(id)+1];
         strcpy(_id,id);
      }
      void preinvoke_premarshal(CORBA::Object_ptr target,
            const char* operation,
            IOP::ServiceContextList& servicecontexts,
            VISClosure& closure) {
         closure.data = new char[ strlen(_id)];
         strcpy( (char*)(closure.data), _id);
         cout << "SampleClientInterceptor id " << closure.data
               << "=================> preinvoke_premarshal "
               << operation << endl;
      }
      void preinvoke_postmarshal(CORBA::Object_ptr target,
            CORBA_MarshalInBuffer& payload,
            VISClosure& closure) {
         cout << "SampleClientInterceptor id " << closure.data
               << "=================> preinvoke_postmarshal "
               << endl;
      }
      void postinvoke(CORBA::Object_ptr target,
            const IOP::ServiceContextList& service_contexts,
            CORBA_MarshalInBuffer& payload,
            CORBA::Environment_ptr env,
            VISClosure& closure) {
         cout << "SampleClientInterceptor id " << closure.data
```

```
                << "================> postinvoke "
                << endl;
        }
        void exception_occurred(CORBA::Object_ptr target,
              CORBA::Environment_ptr env,
              VISClosure& closure) {
              cout << "SampleClientInterceptor id " << closure.data
              << "================> exception_occurred "
              << endl;
        }
};
```

SampleClientLoader

SampleClientLoader は，BindInterceptor オブジェクトのロードを担当します。このクラスは vbroker.orb.dynamicLibs によって VisiBroker ORB に動的にリンクされます。SampleClientLoader クラスは，bind メソッドと bind_succeeded メソッドを含みます。これらのメソッドは，オブジェクトのバインド中に ORB によって呼び出されます。バインドが成功すると，ORB が bind_succeeded を呼び出し，BindInterceptor オブジェクトが作成され InterceptorManager に登録されて結果的にインストールされます。

```
#include <iostream.h>
#include "vinit.h"
#include "SampleBindInterceptor.h"

class BindInterceptorLoader : VISInit {
   private:
      short int _bind_interceptors_installed;
   public:
      BindInterceptorLoader() {
         _bind_interceptors_installed = 0;
      }
      void ORB_init(int& argc, char* const* argv, CORBA::ORB_ptr orb) {
         if( _bind_interceptors_installed) return;
         cout << "Installing Bind interceptors" << endl;
         SampleBindInterceptor *interceptor =
               new SampleBindInterceptor;
         // インターセプタマネージャの制御を取得します。
         CORBA::Object *object =
               orb->resolve_initial_references("VisiBrokerInterceptorControl");
         interceptor::InterceptorManagerControl_var control =
               interceptor::InterceptorManagerControl::_narrow(object);
         // Bind マネージャを取得します。
         interceptor::InterceptorManager_var manager =
               control->get_manager("Bind");
         interceptor::BindInterceptorManager_var bind_mgr =
               interceptor::BindInterceptorManager::_narrow(manager);
         // POA インターセプタをリストに追加します。
         bind_mgr->add( (interceptor::BindInterceptor*)interceptor);
         cout << "Bind interceptors installed" << endl;
         _bind_interceptors_installed = 1;
      }
};
```

SampleBindInterceptor

SampleBindInterceptor は，クライアントがオブジェクトをバインドしようとするときに起動されます。ORB を初期化した後のクライアント側で実行する最初の手順は，AccountManager オブジェクトへのバインドです。このバインドで，SampleBindInterceptor が呼び出され，バインドが成功すると SampleClientInterceptor がインストールされます。

```
#include <iostream.h>
#include "interceptor_c.hh"
#include "IOP_c.hh"
#include "vclosure.h"
#include "SampleClientInterceptor.h"

class SampleBindInterceptor : public interceptor::BindInterceptor {
   public:
      IOP::IORValue_ptr bind(IOP::IORValue_ptr ior,
            CORBA_Object_ptr obj,
            CORBA::Boolean rebind,
            VISClosure& closure) {
         cout << "SampleBindInterceptor-----> bind" << endl;
         return NULL;
      }
      IOP::IORValue_ptr bind_failed(IOP::IORValue_ptr ior,
            CORBA_Object_ptr object,
            VISClosure& closure) {
         cout << "SampleBindInterceptor-----> bind_failed" << endl;
         return NULL;
      }
      void bind_succeeded(IOP::IORValue_ptr ior,
            CORBA_Object_ptr object,
            CORBA::Long profile_index,
            interceptor::InterceptorManagerControl_ptr control,
            VISClosure& closure) {
         cout << "SampleBindInterceptor-----> bind_succeeded"
               << endl;
         // 要求レベルのインターセプタを追加します。
         interceptor::ClientRequestInterceptor_var interceptor =
               new SampleClientInterceptor((char*)"MyClientInterceptor");
         // ClientRequest インターセプタマネージャを取得します。
         interceptor::InterceptorManager_var generic_manager =
               control->get_manager("ClientRequest");
         interceptor::ClientRequestInterceptorManager_var manager =
               interceptor::ClientRequestInterceptorManager::_narrow(
                     generic_manager);
         // インターセプタを追加します。
         manager->add( (interceptor::ClientRequestInterceptor*)interceptor);
         cout <<"============>In bind_succeeded, 1 "
               <<"ClientRequest interceptor installed"<< endl;
      }
      void exception_occurred(IOP::IORValue_ptr ior,
            CORBA_Object_ptr object,
            CORBA_Environment_ptr env,
            VISClosure& closure) {
         cout << "SampleBindInterceptor-----> exception_occured"
               << endl;
      }
};
```

インターセプタ間の情報の受け渡し

インターセプタによる呼び出しのうち，あるシーケンスの開始時に，ORB によって Closure オブジェクトが作成されます。この特定のシーケンス内のすべての呼び出しに対して，同じ Closure オブジェクトが使用されます。Closure オブジェクトには，java.lang.Object 型の 1 つの public データフィールド object があります。このデータフィールドは，状態情報を保存するためにインターセプタによって設定されます。Closure オブジェクトが作成されるシーケンスは，インターセプタのタイプによって異なります。

ClientRequestInterceptor では，preinvoke_premarshal を呼び出す前に新しい Closure が作成され，成功かどうかに関係なく要求が完了するまで，同じ Closure がその要求で使用されます。同様に，ServerInterceptor では，preinvoke を呼び出す前に新しい Closure が作成され，その特定の要求の処理に関連するすべてのインターセプタ呼び出しで同じ Closure が使用されます。

Closure の使い方のサンプルについては，次のディレクトリを参照してください。

```
<install_dir>/examples/vbe/interceptors/client_server
```

Closure オブジェクトを ExtendedClosure にキャストして，次のように response_expected と request_id を取得します。

```
CORBA::Boolean my_response_expected =
    ((ExtendedClosure)closure).reqInfo.response_expected;
CORBA::ULong my_request_id =
    ((ExtendedClosure)closure).reqInfo.request_id;
```

ポータブルインターセプタと VisiBroker インターセプタの同時使用

VisiBroker ORB を使用して，ポータブルインターセプタと VisiBroker インターセプタの両方を同時にインストールできます。ただし，インプリメンテーションが異なるため，両方のインターセプタを使用する開発者は，操作全体の流れについて規則と制約を理解しておく必要があります。

インターセプタの呼び出しポイントの順序

インターセプトポイントの呼び出し順序は，インターセプタのバージョンごとにインターセプトポイントを呼び出す順序の規則にしたがいます。これは，開発者が実際に複数のバージョンをインストールするかどうかに関係なく，したがう必要があります。

クライアント側インターセプタ

インターセプタが例外を生成しないと仮定して，ポータブルと VisiBroker の両方のクライアント側インターセプタがインストールされている場合のイベントの順序は次のようになります。

1. send_request（ポータブルインターセプタ），次に preinvoke_premarshal（インターセプタ）
2. 要求メッセージの構築
3. preinvoke_postmarshal（インターセプタ）
4. 要求メッセージを送信して，応答を待機
5. postinvoke（インターセプタ），次に received_reply ／ receive_exception ／ receive_other（ポータブルインターセプタ）（応答のタイプによる）

サーバー側インターセプタ

インターセプタが例外を生成しないと仮定して，ポータブルと VisiBroker の両方のサーバー側インターセプタがインストールされている場合の，受信するイベントの順序は次のようになります（VisiBroker の場合と同じように検索要求が送られてもインターセプタは起動されません)。

1. received_request_service_contexts（ポータブルインターセプタ)，次に preinvoke（インターセプタ）
2. servantLocator.preinvoke（サーバントロケータを使用する場合）
3. receive_request（ポータブルインターセプタ）
4. サーバント上でオペレーションを呼び出す
5. postinvoke_premarshal（インターセプタ）
6. servantLocator.postinvoke（サーバントロケータを使用する場合）
7. send_reply ／ send_exception ／ send_other（要求の結果による）
8. postinvoke_postmarshal（インターセプタ）

POA 作成中の ORB イベントの順序

POA 作成中の ORB イベントの順序は次のとおりです。

1. POA を処理するサーバーエンジンのプロファイルを基に，IOR テンプレートを作成します。
2. インターセプタの POA ライフサイクルインターセプタの create() メソッドを呼び出します。このメソッドでは，新しいポリシーを追加したり，前の手順で作成した IOR テンプレートを変更できます。
3. ポータブルインターセプタの IORInfo オブジェクトが作成され，IORInterceptor の establish_components() メソッドが呼び出されます。このインターセプトポイントでは，インターセプタは，create_POA() に送られたポリシーと前の手順で追加されたポリシーを照会し，これらのポリシーに基づいて IOR テンプレートにコンポーネントを追加できます。
4. POA のオブジェクトリファレンスファクトリとリファレンステンプレートが作成され，ポータブルインターセプタ IORInterceptor の components_established() メソッドが呼び出されます。このインターセプトポイントでは，インターセプタはオブジェクトリファレンスの作成に使用する POA のオブジェクトリファレンスファクトリを変更できます。

POA リファレンス作成中の ORB イベントの順序

create_reference()，create_reference_with_id() などのオブジェクトリファレンスを作成する POA の呼び出し中に発生するイベントは次のとおりです。

1. オブジェクトリファレンスファクトリの make_object() メソッドを呼び出して，オブジェクトリファレンスを作成します。これにより，IOR 作成メソッドが呼び出されることはありません。ユーザーがファクトリを作成することもできます。VisiBroker IOR 作成インターセプタがインストールされていない場合，これはアプリケーションに返されるオブジェクトリファレンスになります。それ以外の場合はステップ 2 に進んでください。
2. 返されたオブジェクトリファレンスのデリゲートから IOR を抽出し，VisiBroker IOR 作成インターセプタの create() メソッドを呼び出します。

3 `create_reference()`, `create_reference_with_id()` の呼び出し元に, オブジェクトリファレンスとしてステップ 2 の IOR が返されます。

第 26 章

オブジェクトラッパーの使い方

ここでは，VisiBroker のオブジェクトラッパー機能について説明します。オブジェクトラッパーを使用して，アプリケーションはオブジェクトオペレーションリクエストの通知を受けたり，その要求をトラップすることができます。

オブジェクトラッパーの概要

VisiBroker のオブジェクトラッパー機能を使用すると，バインドしたオブジェクトのメソッドをクライアントアプリケーションが呼び出す際，またはサーバーアプリケーションがオペレーションリクエストを受け取る際に呼び出すメソッドを定義できます。VisiBroker ORB レベルで呼び出されるインターセプタ機能とは異なり，オブジェクトラッパーはオペレーションリクエストがマーシャリングされる前に呼び出されます。このためオブジェクトラッパーは，オペレーションリクエストをマーシャリングしたり，ネットワークを介して送信したり，あるいはオブジェクトインプリメンテーションに対して実際に送信されなくても結果を返すように設計できます。VisiBroker インターセプタの詳細については，第 25 章「VisiBroker インターセプタの使い方」を参照してください。

オブジェクトラッパーは，単一のアプリケーションのクライアント側とサーバー側の両方，またはどちらか一方にインストールします。

次に，アプリケーションにおけるオブジェクトラッパーの使用例を示します。

- クライアントが発行したオペレーションリクエスト，またはサーバーが受信したオペレーションリクエストに関する情報をログに記録します。
- オペレーションリクエストが完了するまでに要した時間を計測します。
- 毎回オブジェクトインプリメンテーションにコンタクトしなくてもすぐに結果を返せるように，頻繁に発行されるオペレーションリクエストの結果をキャッシュします。

メモ　VisiBroker ORB オブジェクトの `object_to_string` メソッドを使用して，オブジェクトラッパーがインストールされているオブジェクトへのリファレンスを外部化しても，受信側が別のプロセスの場合ラッパーは文字列化されたリファレンスの受信側には伝達されません。

型付きと型なしのオブジェクトラッパー

VisiBroker には，「*型付き*」と「*型なし*」の 2 種類のオブジェクトラッパーがあります。これらのオブジェクトラッパーは，同じアプリケーション内で併用できます。型付きラッパーの詳細については，361 ページの「型付きオブジェクトラッパー」を参照してください。型なしラッパーの詳細については，356 ページの「型なしオブジェクトラッパー」を参照してください。次の表は，この 2 種類のオブジェクトラッパーの主な相違点をまとめたものです。

表 26.1　型付きおよび型なしオブジェクトラッパーの機能の比較

機能	型付き	型なし
スタブに渡されるすべての引数を受け取る	はい	いいえ
実際に次のラッパー，スタブ，またはオブジェクトインプリメンテーションを呼び出さずに，呼び出し元に制御を返すことができる	はい	いいえ
すべてのオブジェクトのすべてのオペレーションリクエストで呼び出される	いいえ	はい

idl2cpp の特殊な要件

型付きまたは型なしのオブジェクトラッパーを使用する場合は，アプリケーションのコードを生成する際に -obj_wrapper オプションを付けて idl2cpp コンパイラを実行する必要があります。この結果，各インターフェース用のオブジェクトラッパーの基底クラスが生成されます。

Object ラッパーのサンプルアプリケーション

この章で，型付きおよび型なしオブジェクトラッパーの概念の説明に使用されるクライアント／サーバーサンプルアプリケーションは，次のディレクトリにあります。

```
<install_dir>¥examples¥vbe¥interceptors¥objectWrappers¥
```

型なしオブジェクトラッパー

型なしのオブジェクトラッパーを使用すると，オペレーションリクエストの処理前，処理後，またはその両方で呼び出すメソッドを定義できます。型なしラッパーは，クライアントとサーバーのどちらのアプリケーションにもインストールできます。また，複数のラッパーをインストールすることも可能です。

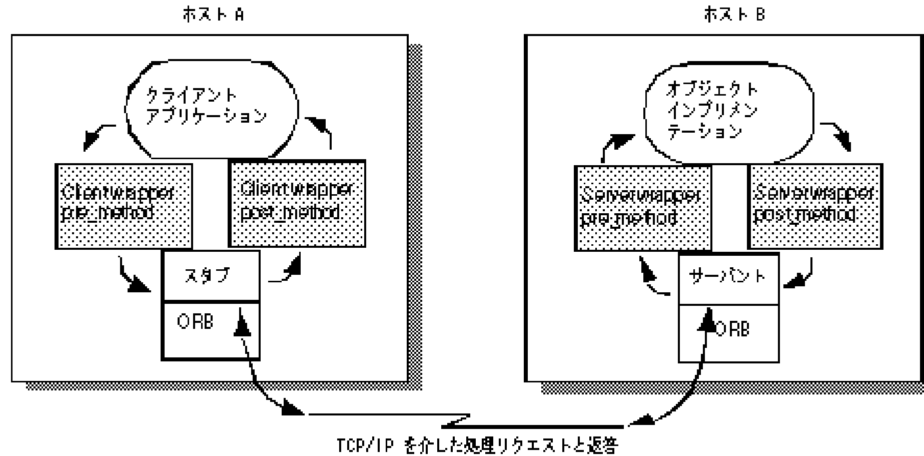
同じクライアントアプリケーションやサーバーアプリケーション内で，型付きと型なしのオブジェクトラッパーを併用することもできます。

デフォルトでは，型なしオブジェクトラッパーはグローバルスコープを持ち，あらゆるオペレーションリクエストで呼び出されます。型なしラッパーは，操作対象外のオブジェクトに対するオペレーションリクエストには影響しないように設計することもできます。

メモ　型付きオブジェクトラッパーとは異なり，型なしオブジェクトラッパーのメソッドは，スタブやオブジェクトインプリメンテーションが受け取る引数を受け取りません。また，スタブやオブジェクトインプリメンテーションの呼び出しも回避できません。

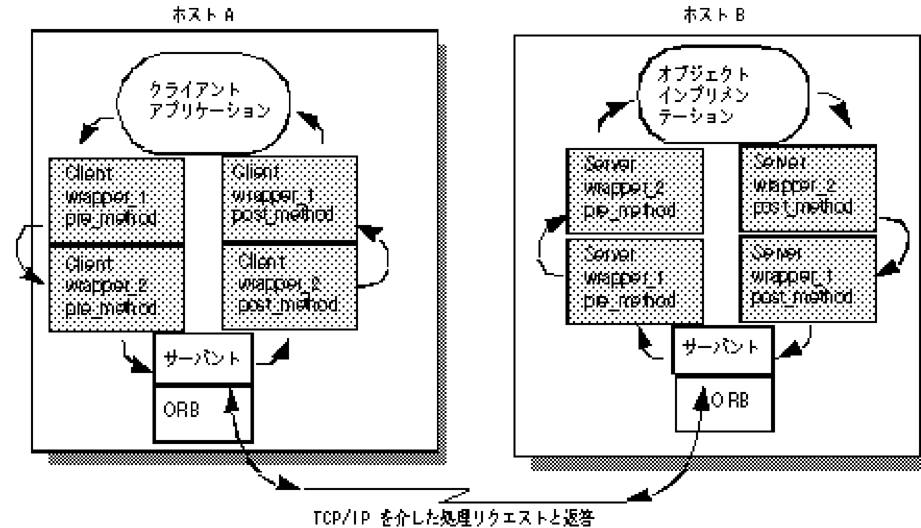
下図では，クライアントスタブメソッドの前に型なしオブジェクトラッパーの pre_method を呼び出してから，その後で post_method が呼び出される流れを示しています。また，オブジェクトインプリメンテーションについても，サーバー側の呼び出し経路を示します。

図 26.1　単一の型なしオブジェクトラッパー



複数の型なしオブジェクトラッパーの使い方

図 26.2　複数の型なしオブジェクトラッパー



pre_method 呼び出しの順序

クライアントでバインドしたオブジェクトのメソッドを呼び出すと，クライアントのスタブルーチンを呼び出す前に，型なしオブジェクトラッパー pre_method はそれぞれ制御を受け取ります。また，サーバーがオペレーションリクエストを受け取ると，そのオブジェクトインプリメンテーションが制御を受け取る前に型なしオブジェクトラッパー pre_method がそれぞれ呼び出されます。どちらの場合も，**最初に登録された**オブジェクトラッパーに属する pre_method メソッドから順に制御を受け取ります。

post_method 呼び出しの順序

サーバーのオブジェクトインプリメンテーションが処理を完了すると，応答がクライアントに送信される前に各 post_method が呼び出されます。クライアントがオペレーションリクエストへの応答を受け取ると，クライアントに制御が戻る前に各 post_method が呼び出されます。どちらの場合も，**最後に登録された**オブジェクトラッパーに属する post_method メソッドから順に制御を受け取ります。

メモ　型付きと型なしのオブジェクトラッパーを併用する場合の呼び出しの順序については，366 ページの「型なしラッパーと型付きラッパーの複合的な使い方」を参照してください。

型なしオブジェクトラッパーの使い方

型なしオブジェクトラッパーの使用に必要な手順は次のとおりです。各手順の詳細については，後で説明します。

1 型なしオブジェクトラッパーを作成するいくつかのインターフェースを決めます。
2 idl2cpp コンパイラで -obj_wrapper オプションを指定して IDL 仕様からコードを生成します。
3 型なしオブジェクトラッパーファクトリのインプリメンテーションを作成します。これは，VISObjectWrapper::UntypedObjectWrapperFactory クラスから派生します。
4 型なしオブジェクトラッパーのインプリメンテーションを作成します。これは，VISObjectWrapper::UntypedObjectWrapper クラスから派生します。
5 型なしオブジェクトラッパーファクトリを作成するように，アプリケーションを変更します。

型なしオブジェクトラッパーファクトリの実装

ObjectWrappers サンプルアプリケーションに含まれている TimeWrap.h ファイルでは，VISObjectWrapper::UntypedObjectWrapperFactory から型なしオブジェクトラッパーファクトリを派生させて定義する方法を示しています。

クライアントがオブジェクトにバインドする際，またはサーバーがオブジェクトインプリメンテーションのメソッドを呼び出す際に，型なしオブジェクトラッパーを作成するためにファクトリの create メソッドを呼び出します。create メソッドはターゲットオブジェクトを受け取ります。これにより，無視するオブジェクト型には型なしオブジェクトラッパーを作成しないようにファクトリを設計できます。また，このメソッドはサーバー側のオブジェクトインプリメンテーション用とクライアント側オブジェクト用のどちらのオブジェクトラッパーを作成するかを示す列挙値も受け取ります。

次のサンプルコードでは，メソッド呼び出しのタイミング情報を表示する型なしオブジェクトラッパーの作成に使用するパラメータ，TimingObjectWrapperFactory を示しています。TimingObjectWrapperFactory のコンストラクタに key パラメータが追加されていることに注意してください。サービスイニシャライザでこのラッパーを識別するためにも，このパラメータを使用します。

```
class TimingObjectWrapperFactory
  : public VISObjectWrapper::UntypedObjectWrapperFactory
{
   public:
      TimingObjectWrapperFactory(VISObjectWrapper::Location loc,
            const char* key)
         : VISObjectWrapper::UntypedObjectWrapperFactory(loc),
            _key(key) {}

      // ObjectWrapperFactory のオペレーション
      VISObjectWrapper::UntypedObjectWrapper_ptr create(
               CORBA::Object_ptr target,
               VISObjectWrapper::Location loc) {
         if (_owrap == NULL) {
               _owrap = new TimingObjectWrapper(_key);
         }
         return VISObjectWrapper::UntypedObjectWrapper::_duplicate(_owrap);
      }
   private:
      CORBA::String_var _key;
```

```
        VISObjectWrapper::UntypedObjectWrapper_var _owrap;
};
```

型なしオブジェクトラッパーのインプリメンテーション

次のサンプルコードは，TimeWrap.h ファイルで定義されている TimingObjectWrapper のインプリメンテーションを示します。形なしラッパーは VISObjectWrapper::UntypedObjectWrapper クラスから派生するものとし，型なしオブジェクトラッパーにある pre_method メソッドと post_method メソッドのいずれにもインプリメンテーションを提供できます。

ファクトリのコンストラクタで自動で，または VISObjectWrapper::ChainUntypedObjectWrapper::add メソッドを呼び出して手動でファクトリをインストールしたら，クライアントをオブジェクトにバインドする際，またはサーバーがオブジェクトインプリメンテーションのメソッドを呼び出す際に，型なしオブジェクトラッパーオブジェクトが自動的に作成されます。

次のサンプルコードにある pre_method は，TimeWrap.C で定義されている TimerBegin メソッドを呼び出し，このメソッドは Closure オブジェクトで現在時刻を保存します。同様に，post_method は TimerDelta メソッドを呼び出し，pre_method の呼び出しからの経過時間を出力します。

```
class TimingObjectWrapper : public VISObjectWrapper::UntypedObjectWrapper {
   public:
      TimingObjectWrapper(const char* key=NULL) : _key(key) {}
      void pre_method(const char* operation,
            CORBA::Object_ptr target,
            VISClosure& closure) {
         cout << "*Timing: [" << flush;
         if ((char *)_key)
            cout << _key << flush;
         else
            cout << "<no key>" << flush;
         cout << "] pre_method¥t" << operation << "¥t->" << endl;
         TimerBegin(closure, operation);
      }
      void post_method(const char* operation,
            CORBA::Object_ptr target,
            CORBA::Environment& env,
            VISClosure& closure) {
         cout << "*Timing: [" << flush;
         if ((char *)_key)
            cout << _key << flush;
         else
            cout << "<no key>" << flush;
         cout << "] post_method¥t" ;
         TimerDelta(closure, operation);
      }
   private:
      CORBA::String_var _key;
};
```

pre_method メソッドと post_method メソッドに共通の引数

pre_method と post_method はどちらも，次の表に示されるパラメータを受け取ります。

表 26.2　pre_method メソッドと post_method メソッドに共通の引数

パラメータ	説明
操作	ターゲットオブジェクトで要求されたオペレーションの名前。
target	ターゲットオブジェクト。
closure	このラッパーの複数のメソッドの呼び出しを介してデータを保存できる領域。
環境	メソッド呼び出しの前の手順で発生した例外をユーザーに通知するために使用される post_method のみのパラメータ。

型なしオブジェクトラッパーファクトリの作成と登録

ロケーションを受け取る基底クラスのコンストラクタで作成した型なしオブジェクトラッパーファクトリは，型なしラッパーのチェインに自動的に追加されます。

クライアント側でオブジェクトがラップされるのは，オブジェクトがバインドされる前に，型なしオブジェクトラッパーファクトリが作成および登録された場合だけです。サーバー側の型なしオブジェクトラッパーファクトリの作成と登録は，オブジェクトインプリメンテーションの呼び出し前に行われます。

次のサンプルコードは，サンプルファイル UntypedClient.C の一部です。クライアントに 2 つの型なしオブジェクトラッパーファクトリを作成して自動登録します。ファクトリは，VisiBroker ORB の初期化されてクライアントがオブジェクトにバインドされるまでに作成されます。

```
int main(int argc, char* const* argv) {
   try {
         // ORB を初期化します。
         CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
         // 型なしオブジェクトラッパーをインストールします。
         TimingObjectWrapperFactory timingfact(VISObjectWrapper::Client,
                  "timeclient");
         TraceObjectWrapperFactory tracingfact(VISObjectWrapper::Client,
                  "traceclient");
         // 口座マネージャを検索します。
   . . .]
```

次のサンプルコードは，サンプルファイル UntypedServer.C です。このサンプルは，サーバーに型なしオブジェクトラッパーファクトリを作成して登録します。ファクトリは，VisiBroker ORB が初期化されてからオブジェクトインプリメンテーションが作成されるまでに作成されます。

```
// UntypedServer.C
#include "Bank_s.hh"
#include "BankImpl.h"
#include "TimeWrap.h"
#include "TraceWrap.h"
USE_STD_NS
int main(int argc, char* const* argv) {
   try {
      // ORB を初期化します。
      CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
      // POA を初期化します。
      CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
      PortableServer::POA_var rootPoa = PortableServer::POA::_narrow(obj);
      CORBA::PolicyList policies;
      policies.length(1);
      policies[(CORBA::ULong)0] = rootPoa->create_lifespan_policy(
            PortableServer::PERSISTENT);
      // POA マネージャを取得します。
      PortableServer::POAManager_var poa_manager = rootPoa->the_POAManager();
      // 適切なポリシーで myPOA を作成します。
      PortableServer::POA_var myPOA = rootPoa->create_POA("bank_ow_poa",
            poa_manager,
            policies);
      // 口座マネージャに対する型なしオブジェクトラッパーをインストールします。
 TimingObjectWrapperFactory timingfact(VISObjectWrapper::Server,
                  "timingserver");
      TraceObjectWrapperFactory tracingfact(VISObjectWrapper::Server,
                  "traceserver");
      // 口座マネージャサーバントを作成します。
      AccountManagerImpl managerServant;
      // サーバントの ID を判定します。
      PortableServer::ObjectId_var managerId =
      PortableServer::string_to_ObjectId("BankManager");
```

```
      // その ID を使って myPOA でサーバントをアクティブ化します。
      myPOA->activate_object_with_id(managerId, &managerServant);
      // POA マネージャをアクティブ化します。
      rootPoa->the_POAManager()->activate();
      cout << "Manager is ready." << endl;
      // 要求の着信を待ちます。
      orb->run();
   } catch(const CORBA::Exception& e) {
      cerr << e << endl;
      return 1;
   }
   return 0;
}
```

型なしオブジェクトラッパーの削除

VISObjectWrapper::ChainUntypedObjectWrapperFactory クラスの remove メソッドで，クライアントアプリケーションまたはサーバーアプリケーションから型なしオブジェクトラッパーファクトリを削除できます。ファクトリを削除するときは，場所を指定する必要があります。つまり，VISObjectWrapper::Both にファクトリを追加した場合は，Client，Server，または Both のいずれかを選択してファクトリを削除できます。

メモ　クライアントから 1 つ以上のオブジェクトラッパーファクトリを削除しても，クライアントがすでにバインドしていたクラスのオブジェクトには影響しません。影響を受けるのは，それ以後にバインドされたオブジェクトだけです。サーバーからオブジェクトラッパーファクトリを削除しても，すでに作成されていたオブジェクトインプリメンテーションには影響しません。影響を受けるのは，それ以後に作成されたオブジェクトインプリメンテーションだけです。

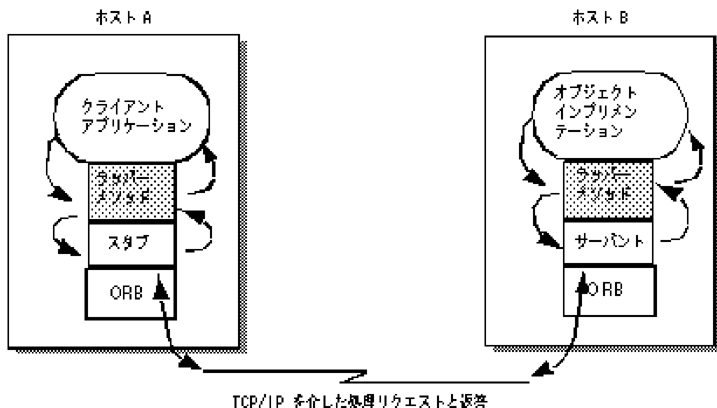
型付きオブジェクトラッパー

特定のクラスの型付きオブジェクトラッパーを実装する場合は，バインドされたオブジェクトのメソッドが呼び出されたときに行う処理を定義します。次の図では，クライアントスタブクラスのメソッドの前にクライアントでオブジェクトラッパーメソッドが呼び出される流れと，サーバーのインプリメンテーションメソッドの前にサーバー側のオブジェクトラッパーが呼び出される流れを示しています。

メモ　型付きオブジェクトラッパーのインプリメンテーションでは，ラップするオブジェクトから提供されるすべてのメソッドを実装する必要はありません。

同じクライアントアプリケーションやサーバーアプリケーション内で，型付きと型なしのオブジェクトラッパーを併用することもできます。詳細については，366 ページの「型なしラッパーと型付きラッパーの複合的な使い方」を参照してください。

図 26.3　登録された単一の型付きオブジェクトラッパー



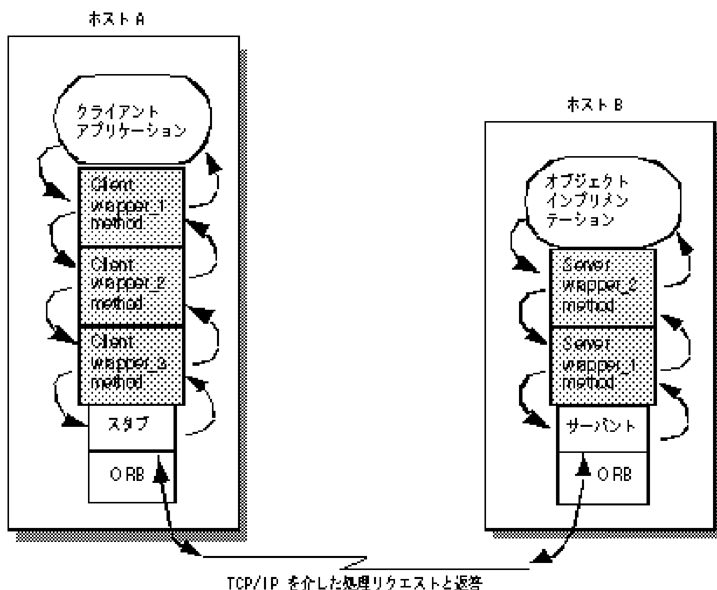
複数の型付きオブジェクトラッパーの使い方

特定クラスの 1 つのオブジェクトに対して，1 つまたは複数の型付きオブジェクトラッパーを実装して登録することができます。次の図を参照してください。

クライアント側で最初に登録されたオブジェクトラッパーは client_wrapper_1 なので，このラッパーのメソッドが最初に制御を受け取ります。処理が完了すると，client_wrapper_1 メソッドはチェイン内の次のオブジェクトのメソッドに制御を渡すか，クライアントに制御を戻します。

サーバー側で最初に登録されたオブジェクトラッパーは server_wrapper_1 なので，このラッパーのメソッドが最初に制御を受け取ります。処理が完了すると，server_wrapper_1 メソッドはチェイン内の次のオブジェクトのメソッドに制御を渡すか，サーバントに制御を戻します。

図 26.4　登録された複数の型付きオブジェクトラッパー



呼び出しの順序

特定のクラス用に登録された型付きオブジェクトラッパーのメソッドは，通常，クライアント側のスタブメソッドまたはサーバー側のスケルトンに渡される引数をすべて受け取ります。オブジェクトラッパーの各メソッドは，親クラスのメソッド

<interface_name>ObjectWrapper::<method_name> を呼び出してチェイン内の次のラッパーメソッドに制御を渡します。チェイン内の次のラッパーメソッドを呼び出さずに制御を返す場合は，オブジェクトラッパーで適切な戻り値を付けて制御を返す（return）ことができます。

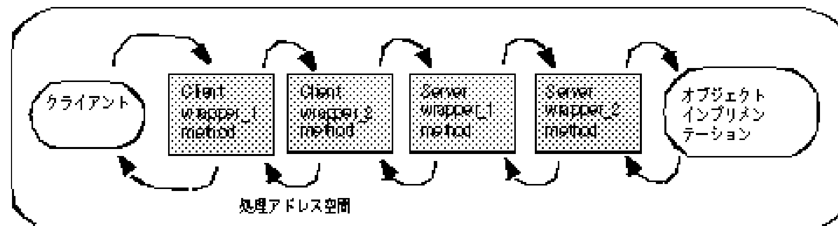
型付きオブジェクトラッパーメソッドにはチェイン内の前のメソッドに制御を戻す機能があり，この機能を使ってクライアントスタブやオブジェクトインプリメンテーションをまったく呼び出さないラッパーメソッドを作成することもできます。たとえば，頻繁に要求される処理の結果をキャッシュするようなオブジェクトラッパーメソッドを作成できます。この例では，バインドされたオブジェクトのメソッドを初めて呼び出すと，オブジェクトインプリメンテーションにオペレーションリクエストが送信されます。オブジェクトラッパーメソッドを介して制御の流れが戻る間に，その結果が格納されます。これ以降に同じメソッドが呼び出されると，オブジェクトラッパーメソッドは実際にはオブジェクトインプリメンテーションにオペレーションリクエストを送らずにキャッシュした結果をそのまま返します。

型付きと型なしのオブジェクトラッパーを併用する場合の呼び出しの順序については，366 ページの「型なしラッパーと型付きラッパーの複合的な使い方」を参照してください。

同じ場所にあるクライアント／サーバーの型付きオブジェクトラッパー

クライアントとサーバーが同じプロセス内に組み込まれている場合，最初に制御を受け取るのは，最初にインストールされたクライアント側のオブジェクトラッパーのメソッドになります。次の図はこの呼び出し順序を示しています。

図 26.5　型付きオブジェクトラッパーの呼び出し順序



型付きオブジェクトラッパーの使い方

型付きオブジェクトラッパーの使用に必要な手順は次のとおりです。各手順の詳細については，後で説明します。

1. 型付きオブジェクトラッパーを作成する 1 つまたは複数のインターフェースを決定します。
2. idl2cpp コンパイラで -obj_wrapper オプションを指定して IDL 仕様からコードを生成します。
3. コンパイラで生成した <interface_name>ObjectWrapper クラスから型付きオブジェクトラッパークラスを派生させ，ラップするメソッドにインプリメンテーションを提供します。
4. 型付きオブジェクトラッパーを登録するように，アプリケーションを変更します。

型付きオブジェクトラッパーの実装

idl2cpp コンパイラで生成した <interface_name>ObjectWrapper クラスから型付きオブジェクトラッパーを派生させます。

次のサンプルコードは，ファイル BankWrap.h の Account インターフェースの型付きオブジェクトラッパーのインプリメンテーションです。

このクラスは AccountObjectWrapper インターフェースから派生し，balance メソッドで簡単なキャッシュインプリメンテーションを提供します。その処理方法は次のとおりです。

1 _inited フラグをチェックして，このメソッドが以前に呼び出されているかどうかを確認します。

2 これが最初の呼び出しの場合，チェインにある次のオブジェクトの balance メソッドが呼び出され，その結果が balance に保存されます。さらに，_inited フラグは true に設定され，結果の値が返されます。

3 このメソッドが以前に呼び出されている場合は，キャッシュした値をそのまま返します。

```
class CachingAccountObjectWrapper : public Bank::AccountObjectWrapper {
  public:
    CachingAccountObjectWrapper() : _inited((CORBA::Boolean)0) {}
    CORBA::Float balance() {
      cout << "+ CachingAccountObjectWrapper: Before Calling Balance" << endl;
 if (! _inited) {
        _balance = Bank::AccountObjectWrapper::balance();
        _inited = 1;
      } else {
        cout << "+ CachingAccountObjectWrapper: Returning Cached Value" <<
      endl;
      }
      cout << "+ CachingAccountObjectWrapper: After Calling Balance" << endl;
      return _balance;
    }
  . . .
};
```

クライアント向け型付きオブジェクトラッパーの登録

型付きオブジェクトラッパーは，idl2cpp コンパイラがクラスに生成した <interface_name>::add method を呼び出して，クライアント側に登録されます。クライアント側のオブジェクトラッパーは，ORB_init メソッドが呼び出されてからオブジェクトがバインドされるまでに登録する必要があります。次のサンプルコードは，型付きオブジェクトラッパーを作成して登録する TypedClient.java ファイルの一部を示します。

```
. . .
int main(int argc, char* const* argv) {
  try {
    // ORB を初期化します。
 CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

    // 口座に対する型付きオブジェクトラッパーをインストールします。
 Bank::AccountObjectWrapper::add(orb,
        CachingAccountObjectWrapper::factory,
         VISObjectWrapper::Client);

    // マネージャの ID を取得します。
    PortableServer::ObjectId_var managerId =
        PortableServer::string_to_ObjectId("BankManager");
    // アカウントマネージャを検索します。
    Bank::AccountManager_var manager =
        Bank::AccountManager::_bind("/bank_ow_poa", managerId);
. . .
```

VisiBroker ORB は，クライアント側に登録されたすべてのオブジェクトラッパーを追跡します。その型のオブジェクトをバインドするためにクライアントが _bind メソッドを呼び出すと，必要なオブジェクトラッパーが作成されます。クライアントが特定のクラスのオブジェクトの複数インスタンスにバインドする場合，インスタンスごとにラッパーのセットが 1 つずつ作成されます。

サーバー向けの型付きオブジェクトラッパーの登録

クライアントアプリケーションの場合と同様に，型付きオブジェクトラッパーは <interface_name>::add メソッドの呼び出しにより，サーバー側に登録されます。サーバー側の型付きオブジェクトラッパーは，ORB_init メソッドが呼び出されてからオブジェクトインプリメンテーションによって要求にサービスが提供されるまでに登録する必要があります。次のサンプルコードは，型付きオブジェクトラッパーをインストールする TypedServer.C ファイルの一部を示します。

```
// TypedServer.C
#include "Bank_s.hh"
#include "BankImpl.h"
#include "BankWrap.h"
USE_STD_NS
int main(int argc, char* const* argv) {
   try {
      // ORB を初期化します。
      CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
      // POA を初期化します。
      CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
      PortableServer::POA_var rootPoa = PortableServer::POA::_narrow(obj);
      CORBA::PolicyList policies;
      policies.length(1);
      policies[(CORBA::ULong)0] = rootPoa->create_lifespan_policy(
            PortableServer::PERSISTENT);
      // POA マネージャを取得します。
      PortableServer::POAManager_var poa_manager = rootPoa->the_POAManager();
      // 適切なポリシーで myPOA を作成します。
      PortableServer::POA_var myPOA = rootPoa->create_POA("bank_ow_poa",
            poa_manager,
            policies);
      // 口座マネージャに対する型付きオブジェクトラッパーをインストールします。
      Bank::AccountManagerObjectWrapper::add(orb,
            SecureAccountManagerObjectWrapper::factory,
            VISObjectWrapper::Server);
      Bank::AccountManagerObjectWrapper::add(orb,
            CachingAccountManagerObjectWrapper::factory,
            VISObjectWrapper::Server);
      // 口座マネージャサーバントを作成します。
      AccountManagerImpl managerServant;
      // サーバントの ID を判定します。
      PortableServer::ObjectId_var managerId =
            PortableServer::string_to_ObjectId("BankManager");
      // その ID を使って myPOA でサーバントをアクティブ化します。
      myPOA->activate_object_with_id(managerId, &managerServant);
      // POA マネージャをアクティブ化します。
      rootPoa->the_POAManager()->activate();
      cout << "Manager is ready." << endl;
      // 要求の着信を待ちます。
      Orb>run();
   } catch(const CORBA::Exception& e) {
      cerr << e << endl;
      return 1;
   }
   return 0;
}
```

サーバーで特定のクラスのオブジェクトの複数インスタンスを作成する場合，各インスタンスごとにラッパーセットが 1 つずつ作成されます。

型付きオブジェクトラッパーの削除

idl2cpp コンパイラがクラスに生成した <interface_name>ObjectWrapper::remove メソッドで，クライアントアプリケーションやサーバーアプリケーションから型付きオブジェクトラッパーを削除できます。ファクトリを削除するときは，場所を指定する必要があります。つまり，VISObjectWrapper::Both にファクトリを追加した場合は，Client，Server，または Both のいずれかを選択してファクトリを削除できます。

メモ クライアントから 1 つまたは複数のオブジェクトラッパーを削除しても，クライアントがすでにバインドしていたクラスのオブジェクトには影響ありません。影響を受けるのは，それ以後にバインドされたオブジェクトだけです。サーバーからオブジェクトラッパーを削除しても，すでに要求に応えていたオブジェクトインプリメンテーションには影響ありません。影響を受けるのは，それ以後に作成されたオブジェクトインプリメンテーションだけです。

型なしラッパーと型付きラッパーの複合的な使い方

1 つのアプリケーションで型付きと型なしのオブジェクトラッパーを併用する場合は，オブジェクトに定義されているすべての型付きオブジェクトラッパーのメソッドよりも前に，そのオブジェクトの型なしラッパーに定義されている pre_method メソッドがすべての呼び出されます。反対に，型なしラッパーに定義されているすべての post_method メソッドよりも前に，そのオブジェクトに定義されている型付きオブジェクトラッパーのメソッドがすべて呼び出されます。

サンプルアプリケーション Client.C と Server.C は，使用する型付きオブジェクトラッパーと型なしのオブジェクトラッパーをコマンドラインのプロパティで指定できる優れた設計になっています。

型付きラッパーのコマンドライン引数

次の表は，Client.C と Server.C で実装したサンプル Bank アプリケーション用の型付きオブジェクトラッパーを有効にするためのコマンドライン引数です。

表 26.3 型付きオブジェクトラッパーを制御するコマンドライン引数

Bank ラッパーのプロパティ	説明
-BANKaccountCacheClnt <0|1>	クライアントアプリケーションの balance メソッドの結果をキャッシュする型付きオブジェクトラッパーを有効／無効にします。
-BANKaccountCacheSrvr <0|1>	サーバーアプリケーションの balance メソッドの結果をキャッシュする型付きオブジェクトラッパーを有効／無効にします。
-BANKmanagerCacheClnt <0|1>	クライアントアプリケーションの open メソッドの結果をキャッシュする型付きオブジェクトラッパーを有効／無効にします。
-BANKmanagerCacheSrvr <0|1>	サーバーアプリケーションの open メソッドの結果をキャッシュする型付きオブジェクトラッパーを有効／無効にします。
-BANKmanagerSecurityClnt <0|1>	クライアントアプリケーションの open メソッドで渡された許可されていないユーザーを検出する型付きオブジェクトラッパーを有効／無効にします。
-BANKmanagerSecuritySrvr <0|1>	サーバーアプリケーションの open メソッドで渡された許可されていないユーザーを検出する型付きオブジェクトラッパーを有効／無効にします。

型付きラッパーのイニシャライザ

型付きラッパーは BankInit::update イニシャライザで作成され，objectWrappers/BankWrap.C で定義されます。イニシャライザは ORB_init メソッドの呼び出しで呼び出され，指定した

コマンドラインプロパティに基づいてさまざまな型付きオブジェクトラッパーをインストールします。

次のサンプルコードでは，イニシャライザが PropStruct オブジェクトセットを利用して，指定済みのコマンドラインオプションを追跡し，指定場所の AccountObjectWrapper オブジェクトを追加，削除する手順を示しています。

```
. . .
static const CORBA::ULong kNumTypedAccountProps = 2;
static PropStruct TypedAccountProps[kNumTypedAccountProps] =
{ { "BANKaccountCacheClnt", CachingAccountObjectWrapper::factory,
      VISObjectWrapper::Client },
   { "BANKaccountCacheSrvr", CachingAccountObjectWrapper::factory,
      VISObjectWrapper::Server }
};
static const CORBA::ULong kNumTypedAccountManagerProps = 4;
static PropStruct TypedAccountManagerProps[kNumTypedAccountManagerProps] =
{ { "BANKmanagerCacheClnt", CachingAccountManagerObjectWrapper::factory,
      VISObjectWrapper::Client },
{ "BANKmanagerSecurityClnt", SecureAccountManagerObjectWrapper::factory,
      VISObjectWrapper::Client },
{ "BANKmanagerCacheSrvr", CachingAccountManagerObjectWrapper::factory,
      VISObjectWrapper::Server },
{ "BANKmanagerSecuritySrvr", SecureAccountManagerObjectWrapper::factory,
      VISObjectWrapper::Server },
};
void BankInit::update(int& argc, char* const* argv) {
   if (argc > 0) {
      init(argc, argv, "-BANK");
      CORBA::ULong i;

      for (i=0; i < kNumTypedAccountProps; i++) {
         CORBA::String_var arg(getArgValue(TypedAccountProps[i].propname));
         if (arg && strlen(arg) > 0) {
            if (atoi((char*) arg)) {
               Bank::AccountObjectWrapper::add(_orb,
                      TypedAccountProps[i].fact,
                 TypedAccountProps[i].loc);
            } else {
               Bank::AccountObjectWrapper::remove(_orb,
                      TypedAccountProps[i]Fact,
                     TypedAccountProps[i].loc);
            }
         }
      }
      for (i=0; i < kNumTypedAccountManagerProps; i++) {
            CORBA::String_var arg(
               getArgValue(TypedAccountManagerProps[i].propname));
            if (arg && strlen(arg) > 0) {
                  if (atoi((char*) arg)) {
                        Bank::AccountManagerObjectWrapper::add(_orb,
                              TypedAccountManagerProps[i]Fact,
                              TypedAccountManagerProps[i].loc);
                  }    else {
                        Bank::AccountManagerObjectWrapper::remove(_orb,
                             TypedAccountManagerProps[i]Fact,
                            TypedAccountManagerProps[i].loc);
                  }
            }
      }
   }
}
```

型なしラッパーのコマンドライン引数

次の表は，Client.C と Server.C で実装したサンプル Bank アプリケーション用の型なしオブジェクトラッパーを有効にするためのコマンドライン引数です。

表 26.4　型なしオブジェクトラッパーを制御するためのコマンドライン引数

Bank ラッパーのプロパティ	説明
-TRACEWRAPclient <numwraps>	クライアントアプリケーションについて，ラッパー追跡用の型なしオブジェクトラッパーファクトリを指定された数だけインスタンス化します。
-TRACEWRAPserver <numwraps>	サーバーアプリケーションについて，追跡用の型なしオブジェクトラッパーファクトリを指定された数だけインスタンス化します。
-TRACEWRAPboth <numwraps>	クライアントアプリケーションとサーバーアプリケーションの両方について，追跡用の型なしオブジェクトラッパーファクトリを指定された数だけインスタンス化します。
-TIMINGWRAPclient <numwraps>	クライアントアプリケーションについて，時間測定用の型なしオブジェクトラッパーファクトリを指定された数だけインスタンス化します。
-TIMINGWRAPserver <numwraps>	サーバーアプリケーションについて，時間測定用の型なしオブジェクトラッパーファクトリを指定された数だけインスタンス化します。
-TIMINGWRAPboth <numwraps>	クライアントアプリケーションとサーバーアプリケーションの両方について，時間測定用の型なしオブジェクトラッパーファクトリを指定された数だけインスタンス化します。

型なしラッパーのイニシャライザ

型なしラッパーは，BankWrappers/TraceWrap.C および TimeWrap.C で定義された TraceWrapInit::update および TimingWrapInit::update メソッドで作成および登録されます。これらのイニシャライザは，ORB_init メソッドの呼び出し時に呼び出され，各種型なしオブジェクトラッパーのインストールを処理します。

次のサンプルコードは，TraceWrap.C ファイルの一部です。このファイルは，指定したコマンドラインプロパティに基づいて，型なしオブジェクトラッパーファクトリをインストールします。

```
TraceWrapInit::update(int& argc, char* const* argv) {
   if (argc > 0) {
      init(argc, argv, "-TRACEWRAP");
      VISObjectWrapper::Location loc;
      const char* propname;
      LIST(VISObjectWrapper::UntypedObjectWrapperFactory_ptr) *list;

      for (CORBA::ULong i=0; i < 3; i++) {
         switch (i) {
            case 0:

               loc = VISObjectWrapper::Client;
               propname = "TRACEWRAPclient";
               list = &_clientfacts;
               break;
            case 1:
               loc = VISObjectWrapper::Server;
               propname = "TRACEWRAPserver";
               list = &_serverfacts;
               break;
            case 2:
               loc = VISObjectWrapper::Both;
               propname = "TRACEWRAPboth";
               list = &_bothfacts;
               break;
         }
         CORBA::String_var getArgValue(property_value(propname));
```

```
            if (arg && strlen(arg) > 0) {
               int numNew = atoi((char*) arg);
               char key_buf[256];
               for (CORBA::ULong j=0; j < numNew; j++) {
                  sprintf(key_buf, "%s-%d", propname, list->size());
                  list->add(new TraceObjectWrapperFactory(loc,
                        (const char*) key_buf));
               }
            }
         }
      }
   }
```

サンプルアプリケーションの実行

サンプルアプリケーションを実行する前に，ネットワーク上で osagent が動作していることを確認してください。詳細については，第 14 章「スマートエージェントの使い方」を参照してください。確認したら，次のコマンドを使ってトレースと時間測定のオブジェクトラッパーなしでサーバーアプリケーションを実行します。

```
prompt> Server
```

メモ　サーバーは共用アプリケーションとして設計されており，サーバーとクライアントの両方を実装します。

次のコマンドを使用して，別のウィンドウからトレースと時間測定のオブジェクトラッパーなしで，あるユーザーの口座残高を照会するクライアントアプリケーションを実行します。

```
prompt> Client John
```

デフォルトの名前を使用する場合は，次のコマンドを実行します。

```
prompt> Client
```

トレースおよび時間測定のオブジェクトラッパーをオンにする

型なしのトレースおよび時間測定のオブジェクトラッパー有効にしてクライアントを実行するには，次のコマンドを使用します。

```
prompt> Client -TRACEWRAPclient 1 -TIMINGWRAPclient 1
```

型なしのトレースと時間測定のラッパーを有効にしてサーバーを実行するには，次のコマンドを使用します。

```
prompt> Server -TRACEWRAPserver 1 -TIMINGWRAPserver 1
```

キャッシュとセキュリティのオブジェクトラッパーをオンにする

型付きのキャッシュとセキュリティのラッパーを有効にして，クライアントを実行するには，次のコマンドを使用します。

```
prompt> Client -BANKaccountCacheClnt 1 _BANKmanagerCacheClnt 1 ¥
         -BANKmanagerSecurityClnt 1
```

型付きのキャッシュとセキュリティのラッパーを有効にしてサーバーを実行するには，次のコマンドを使用します。

```
prompt> Server -BANKaccountCacheSrvr 1 -BANKmanagerCacheSrvr 1 ¥
         -BANKmanagerSecuritySrvr 1
```

型付きラッパーと型なしラッパーをオンにする

型付きと型なしのオブジェクトラッパーをすべて有効にしてクライアントを実行するには，次のコマンドを使用します。

```
prompt> Client -BANKaccountCacheClnt 1 -BANKmanagerCacheClnt 1 ¥
      -BANKmanagerSecurityClnt 1 ¥
      -TRACEWRAPclient 1 -TIMINGWRAPclient 1
```

型付きと型なしのオブジェクトラッパーをすべて有効にしてクライアントを実行するには，次のコマンドを使用します。

```
prompt> Server BANKaccountCacheSrvr 1 BANKmanagerCacheSrvr 1 ¥
         -BANKmanagerSecuritySrvr 1 ¥ -TRACEWRAPserver 1 -TIMINGWRAPserver 1
```

共用クライアント／サーバーを実行する

次のコマンドはすべての型付きラッパーが有効な共用サーバーとクライアントを実行します。さらに，クライアントの型なしラッパー，サーバーの型なしトレーシングラッパーが有効になっている場合もこのコマンドを使用できます。

```
prompt> Server -BANKaccountCacheClnt 1 -BANKaccountCacheSrvr 1 ¥
         -BANKmanagerCacheClnt 1 -BANKmanagerCacheSrvr 1 ¥
         -BANKmanagerSecurityClnt 1 ¥
         -BANKmanagerSecuritySrvr 1 ¥
         -TRACEWRAPboth 1 ¥
         -TIMINGWRAPboth 1
```

第 27 章

イベントキュー

ここでは，イベントキューの機能について説明します。イベントキューは，サーバー側だけで使用できる機能です。

サーバーは，サーバーが必要とするイベントタイプに基づいてリスナーをイベントキューに登録しておき，必要なときにそのイベントを処理することができます。

イベントタイプ

現在生成されるイベントタイプは，接続イベントタイプだけです。

接続イベント

VisiBroker ORB が生成して登録済みの接続イベントへ渡される 2 つの接続イベントは，次のとおりです。

- *接続設立*：これは，新規クライアントをサーバーへ正常に接続することを示します。
- *接続閉鎖*：これは，既存のクライアントをサーバーから切断することを示します。

イベントリスナー

サーバーは，サーバーが処理する必要があるイベントタイプに基づいてリスナーを実装し，VisiBroker ORB に登録します。サポートされているイベントリスナーは，接続イベントリスナーだけです。

IDL 定義

インターフェースの定義は次のとおりです。

```
module EventQueue {
   // 接続イベントタイプ
   enum EventType {UNDEFINED, CONN_EVENT_TYPE};
   // ピア（クライアント）接続情報
   struct ConnInfo {
```

```
        string ipaddress; // %d.%d.%d.%d 形式
        long port;
        long connID;
    };
    // すべてのタイプのイベントリスナーのマーカーインターフェース
    local interface EventListener {};
    typedef sequence<EventListener> EventListeners;
    // 接続イベントリスナーインターフェース
    local interface ConnEventListener : EventListener{
        void conn_established(in ConnInfo info);
        void conn_closed(in ConnInfo info);
    };
    // EventQueue マネージャ
    local interface EventQueueManager : interceptor::InterceptorManager {
        void register_listener(in EventListener listener, in EventType type);
        void unregister_listener(in EventListener listener, in EventType type);
        EventListeners get_listeners(in EventType type);
    };
};
```

次の節では，インターフェースの定義について詳しく説明します。

ConnInfo 構造体

ConnInfo 構造体には，次のクライアント接続情報が含まれます。

表 27.1　ConnInfo 構造体のクライアント接続情報

パラメータ	説明
ipaddress	クライアントの IP アドレスを格納します。
port	クライアントのポート番号を格納します。
connID	このクライアント接続に対するサーバーごとの一意の識別子を保持します。

EventListener インターフェース

EventListener インターフェースセクションは，すべての種類のイベントリスナーのマーカーインターフェースです。

ConnEventListeners インターフェース

ConnEventListeners インターフェースは，次のオペレーションを定義します。

表 27.2　ConnEventListeners インターフェースのオペレーション

オペレーション	説明
`void conn_established (in ConnInfo info)`	接続が設立されたイベントを実行するために，VisiBroker ORB がこのオペレーションをコールバックします。VisiBroker ORB はクライアントの接続情報を ConnInfo info パラメータに記述し，この値をコールバックオペレーションに渡します。
`void conn_closed (in ConnInfo info)`	接続が閉鎖されたイベントを実行するために，VisiBroker ORB がこのオペレーションをコールバックします。VisiBroker ORB はクライアントの接続情報を ConnInfo info パラメータに記述し，この値をコールバックオペレーションに渡します。

サーバー側のアプリケーションの役割は，ConnEventListener インターフェースのインプリメンテーションと，リスナーに対して実行されているイベントを処理することです。

EventQueueManager インターフェース

サーバー側のインプリメンテーションでイベントリスナーを登録する際に，EventQueueManager インターフェースをハンドルとして使用します。このインターフェースは，次のオペレーションを定義します。

オペレーション	説明
void register_listener (in EventListener listener, in EventType type)	このオペレーションは，指定したイベントタイプにイベントリスナーを登録する場合に提供されます。
EventListeners get_listeners (in EventType type)	このオペレーションは，指定したタイプの登録済みイベントリスナーのリストを返します。
void unregister_listener (in EventListener listener, in EventType type)	このオペレーションは，指定したタイプに事前に登録されたリスナーを削除します。

EventQueueManager を返す方法

EventQueueManager オブジェクトは，ORB を初期化する際に作成されます。サーバー側インプリメンテーションは，次のコードを使って EventQueueManager オブジェクトリファレンスを返します。

```
CORBA::Object *object =
  orb->resolve_initial_references("VisiBrokerInterceptorControl");
  interceptor::InterceptorManagerControl_var control =
    interceptor::InterceptorManagerControl::_narrow(object);
  interceptor::InterceptorManager_var manager =
    control->get_manager("EventQueueManager");
  EventQueue::EventQueueManager_var eq_mgr =
    EventQueue::EventQueueManager::_narrow(manager);
```

イベントキューのサンプルコード

この節では，EvenrListener を登録し，接続 EventListener を実装するサンプルコードを提供します。

EventListener の登録

SampleServerLoader クラスには，init() メソッドがあり，初期化時に ORB によって呼び出されます。ServerLoader の目的は，EventListener を作成して EventQueueManager に登録することです。

```
#ifdef _VIS_STD
#include <iostream>
#else
#include <iostream.h>
#endif
#include "vinit.h"
#include "ConnEventListenerImpl.h"

USE_STD_NS

class SampleServerLoader : VISInit {
  private:
    short int _conn_event_interceptors_installed;
  public:
    SampleServerLoader(){
      _conn_event_interceptors_installed = 0;
    }
    void ORB_init(int& argc, char* const* argv, CORBA::ORB_ptr orb) {
      if( _conn_event_interceptors_installed) return;
      cout << "Installing Connection event interceptors" << endl;
      ConnEventListenerImpl *interceptor =
```

```
      new ConnEventListenerImpl("ConnEventListener");
    // インターセプタマネージャの制御を取得します。
    CORBA::Object *object =
      orb->resolve_initial_references("VisiBrokerInterceptorControl");
    interceptor::InterceptorManagerControl_var control =
      interceptor::InterceptorManagerControl::_narrow(object);
    // POA マネージャを取得します。
    interceptor::InterceptorManager_var manager =
       control->get_manager("EventQueueManager");
    EventQueue::EventQueueManager_var eq_mgr =
       EventQueue::EventQueueManager::_narrow(manager);
    // POA インターセプタをリストに追加します。
    eq_mgr->register_listener(
      (EventQueue::ConnEventListener *)interceptor, EventQueue::CONN_EVENT_TYPE);
    cout << "Event queue interceptors installed" << endl;
    _conn_event_interceptors_installed = 1;
  }
};
```

EventListener の実装

ConnEventListenerImpl には，接続イベントリスナーのインプリメンテーションサンプルがあります。ConnEventListener インターフェースは，サーバー側のアプリケーションで conn_established と conn_closed operations オペレーションを実装します。詳細については，372 ページの「ConnEventListeners インターフェース」を参照してください。このインプリメンテーションによって，接続はサーバー側で要求を待機しながら 30,000 ミリ秒間アイドリングできるようになります。これらのオペレーションは，クライアントによって接続が確立されたときと接続が切断されたときに呼び出されます。

```
#ifdef _VIS_STD
#include <iostream>
#else
#include <iostream.h>
#endif
#include "vextclosure.h"
#include "interceptor_c.hh"
#include "IOP_c.hh"
#include "EventQueue_c.hh"
#include "vutil.h"

// USE_STD_NS は，std 名前空間 USE_STD_NS を使用するために VisiBroker によって設定される定義です

//-------------------------------------------------------------------
// サーバーインターセプタの機能を定義します。
//-------------------------------------------------------------------
class ConnEventListenerImpl : EventQueue::ConnEventListener
{
   private:
      char * _id;
   public:
      ConnEventListenerImpl( const char* id) {
         _id = new char[ strlen(id) + 1];
         strcpy( _id,id);
    }
   ~ConnEventListenerImpl() {
      delete[] _id;
      _id = NULL;
    }

    //-------------------------------------------------------------------
    // このメソッドは，要求がサーバーエンドに到着すると呼び出されます。
    //-------------------------------------------------------------------

   void conn_established(const EventQueue::ConnInfo& connInfo){
```

```
        cout <<"Processing connection established from" <<endl;
        cout << connInfo;
        cout <<endl;
        VISUtil::sleep(30000);
    }
    void conn_closed(const EventQueue::ConnInfo & connInfo) {
        cout <<"Processing connection closed from  " <<endl ;
        cout <<connInfo ;
        cout << endl;
        VISUtil::sleep(30000);
    }
};
```

第 28 章

動的に管理される型の使い方

この節では，実行時にデータ型を構築して解釈する VisiBroker の DynAny 機能について説明します。

DynAny インターフェースの概要

DynAny インターフェースは，実行時に基本データ型と構造データ型を動的に作成する方法を提供します。また，コンパイル時に，Any オブジェクトが保持するデータ型をサーバーが認識していない場合でも，そのオブジェクトの情報を解釈したり，抽出することができます。DynAny インターフェースを使用すると，実行時にデータ型を作成したり解釈する強力なクライアントおよびサーバーアプリケーションを構築できます。

DynAny サンプル

VisiBroker には，DynAny の使い方を紹介したサンプルクライアント／サーバーアプリケーションが付属しています。サンプルは次のディレクトリに置かれています。

```
<install_dir>¥examples¥vbe¥dynany¥
```

これらのサンプルプログラムは，この節で DynAny の概念を説明するために使用されます。

DynAny 型

DynAny オブジェクトは，基本データ型（boolean，int，または float）または構造データ型のどちらかの関連値を持ちます。DynAny インターフェースのメソッドとクラスの説明は，『VisiBroker API リファレンス』にも記載されています。『VisiBroker for C++ 開発者ガイド』の第 4 章「C++ 対応プログラマツール」には，含まれているデータ型を判定する方法と，プリミティブデータ型の値を設定および抽出する方法が記載されています。

構造型データは，DynAny からすべてが派生する次のインターフェースで表されます。これらのインターフェースには，それぞれが保持する値の設定や抽出に適したメソッドのセットが個別に用意されています。

表 28.1　構造データ型を表す DynAny から派生するインターフェース

インターフェース	TypeCode	説明
DynArray	_tk_array	同じデータ型の値の配列。要素数は固定。
DynEnum	_tk_enum	単一の列挙体の値。
DynFixed	_tk_fixed	サポートされてない。
DynSequence	_tk_sequence	同じデータ型の値のシーケンス。要素数を増減できる。
DynStruct	_tk_struct	構造体。
DynUnion	_tk_union	共用体。
DynValue	_tk_value	サポートされてない。

DynAny 使用上の制限

DynAny オブジェクトは，作成プロセスによってローカルでのみ使用できます。DynAny オブジェクトをバインドされたオブジェクトに対するオペレーションリクエストのパラメータとして使用したり，ORB::object_to_string メソッドを使って DynAny オブジェクトを外部化すると，MARSHAL 例外が生成されます。

さらに，DynAny オブジェクトをパラメータとして DII 要求で使用すると，NO_IMPLEMENT 例外が生成されます。

このバージョンでは，CORBA 2.6 で指定されている long double 型と fixed 型がサポートされません。

DynAny の作成

DynAnyFactory オブジェクトでオペレーションを呼び出し，DynAny オブジェクトを作成します。まず，DynAnyFactory オブジェクトへのリファレンスを取得し，次にそのオブジェクトを使って新しい DynAny オブジェクトを作成します。

```
CORBA::Object_var obj = orb->resolve_initial_references("DynAnyFactory");
DynamicAny::DynAnyFactory_var factory = DynamicAny::DynAnyFactory::_narrow(obj);
// 動的構造体を作成します。
DynamicAny::DynAny_var dynany = factory->create_dyn_any_from_type_code(
      Printer::_tc_StructType);
DynamicAny::DynStruct_var info = DynamicAny::DynStruct::_narrow(dynany);
info->set_members(seq);
CORBA::Any_var any = info->to_any();
```

DynAny の値の初期化とアクセス

DynAny::insert_<type> メソッドを使用すると，さまざまな基本データ型で DynAny オブジェクトを初期化できます。この <type> は，boolean，octet，char などになります。DynAny に定義されている TypeCode に一致しない型を挿入しようとすると，TypeMismatch 例外が生成されます。

C++ の DynAny::get_<type> メソッドや Java の DynAny.get_<type> メソッドは，DynAny オブジェクトに含まれる値にアクセスできます。この <type> は，boolean，octet，char などになります。DynAny に定義されている TypeCode に一致しない DynAny コンポーネントの値にアクセスしようとすると，TypeMismatch 例外が生成されます。

DynAny インターフェースには，Any オブジェクトと相互にコピー，代入，および変換を行うためのメソッドもあります。サンプルプログラムには，これらのメソッドの使用例が示されています。380 ページの「DynAny サンプルクライアントアプリケーション」と 381 ページの「DynAny サンプルサーバーアプリケーション」を参照してください。

構造データ型

次の型は DynAny インターフェースから派生し，構造データ型を表すために使用します。

構造データ型内の複数のコンポーネント間の移動

DynAny から派生されるインターフェースは，実際に複数のコンポーネントを持つ場合があります。DynAny インターフェースは，これらのコンポーネント内を巡回できるメソッドを提供します。複数のコンポーネントを含む DynAny から派生したオブジェクトは，ポインタを現在のコンポーネントに維持します。

DynAny のメソッド	説明
rewind	現在のコンポーネントのポインタを最初のコンポーネントに再設定します。オブジェクトが 1 つしかコンポーネントを持たない場合は，何も効果がありません。
next	ポインタを次のコンポーネントに進めます。次のコンポーネントがないか，オブジェクトが 1 つしかコンポーネントを持たない場合は，false が返されます。
current_component	DynAny オブジェクトを返します。コンポーネントの TypeCode に基づいて，このオブジェクトを適切な型にナローイングできます。
seek	現在のコンポーネントのポインタを特定のコンポーネントに設定します。コンポーネントは，0 から始まるインデックスで指定します。指定されたインデックスにコンポーネントがない場合は，false を返します。負のインデックスを指定した場合，現在のコンポーネントのポインタは -1（コンポーネントなし）に設定されます。

DynEnum

DynEnum インターフェースは，単一の列挙型定数を表します。この値を文字列または整数値として設定および取得するためのメソッドが提供されます。

DynStruct

DynStruct インターフェースは，動的に構築された struct 型を表します。NameValuePair オブジェクトのシーケンスを使用して，構造体のメンバーを取得したり設定できます。各 NameValuePair オブジェクトは，メンバー名とメンバーの型と値を含む Any を含みます。

rewind，next，current_component，seek の各メソッドを使用して，構造体内のメンバー間を巡回できます。構造体のメンバーを設定したり，取得するためのメソッドも提供されます。

DynUnion

DynUnion インターフェースは，union を表し，2 つのコンポーネントを含みます。最初のコンポーネントはディスクリミネータを表し，2 番めのコンポーネントはメンバーの値を表します。

rewind，next，current_component，seek の各メソッドを使用して，コンポーネント間を巡回できます。共用体のディスクリミネータとメンバーの値を設定したり，取得するためのメソッドも提供されます。

DynSequence と DynArray

DynSequence または DynArray は，シーケンスまたはアレーの各コンポーネント用に，別の DynAny オブジェクトを生成する必要がないベーシックまたは構造データ型のシーケンスを

表します。DynSequence のコンポーネント数は変更できますが，DynArray のコンポーネント数は固定です。

rewind，next，current_component，seek の各メソッドを使用して，DynArray または DynSequence 内のメンバー間を巡回できます。

DynAny サンプル IDL

次のサンプルコードは，サンプルクライアント／サーバーアプリケーションで使用される IDL です。StructType 構造体は，2 つの基本データ型と 1 つの列挙値を含みます。PrinterManager インターフェースは，Any の内容を表示するために使用されます。このとき，このオブジェクトが持つデータ型に関する静的な情報は不要です。

```
// Printer.idl
module Printer {
   enum EnumType {first, second, third, fourth};
   struct StructType {
      string str;
      EnumType e;
      float fl;
   };
   interface PrinterManager {
      void printAny(in any info);
      oneway void shutdown();
   };
};
```

DynAny サンプルクライアントアプリケーション

次のサンプルコードは, 次の VisiBroker 配布ディレクトリに置かれているクライアントアプリケーションを示します。

```
<install_dir>¥examples¥vbe¥dynany¥
```

クライアントアプリケーションは DynStruct インターフェースを使用し，動的に StructType 構造体を作成します。

DynStruct インターフェースは NameValuePair オブジェクトのシーケンスを使用して，構造体メンバーとそれらの対応値を表します。各名前と値の組は，構造体のメンバー名を保持する文字列，およびそのメンバーの値を保持する Any オブジェクトで構成されます。

通常の方法で VisiBroker ORB を初期化し，PrintManager オブジェクトにバインドした後，クライアントは次の手順を実行します。

1. 適切な型を使って空の DynStruct を作成する。
2. 構造体メンバーを持つ NameValuePair オブジェクトのシーケンスを作成する。
3. 構造体メンバーの各値に対して，Any オブジェクトを作成して初期化する。
4. 適切なメンバー名と値を使用して，各 NameValuePair を初期化する。
5. NameValuePair シーケンスを使用して，DynStruct オブジェクトを初期化する。
6. 変換された DynStruct を標準の Any 型に渡して，PrinterManager::printAny メソッドを呼び出す。

メモ オペレーションリクエストのパラメータとして DynAny またはその派生型のオブジェクトを渡すには，その前に DynAny::to_any メソッドを使用して，そのオブジェクトを Any に変換する必要があります。

次のサンプルコードは，DynStruct を使用するクライアントアプリケーションのサンプルです。

```
// Client.C
#include "Printer_c.hh"
#include "dynany.h"
int main(int argc, char* const* argv) {
  try {
    // ORB を初期化します。
    CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
    DynamicAny::DynAnyFactory_var factory =
        DynamicAny::DynAnyFactory::_narrow(
            orb->resolve_initial_references("DynAnyFactory"));
    // マネージャの ID を取得します。
    PortableServer::ObjectId_var managerId =
        PortableServer::string_to_ObjectId("PrinterManager");
    // AccountManager を検索します。完全な POA 名とサーバント ID を指定します。
    Printer::PrinterManager_ptr manager =
        Printer::PrinterManager::_bind("/serverPoa", managerId);
    DynamicAny::NameValuePairSeq seq(3);
    seq.length(3);
    CORBA::Any strAny,enumAny,floatAny;
    strAny <<= "String";
    enumAny <<= Printer::second;
    floatAny <<= (CORBA::Float)864.50;
    CORBA::NameValuePair nvpairs[3];
    nvpairs[0].id = CORBA::string_dup("str");
    nvpairs[0].value = strAny;
    nvpairs[1].id = CORBA::string_dup("e");
    nvpairs[1].value = enumAny;
    nvpairs[2].id = CORBA::string_dup("fl");
    nvpairs[2].value = floatAny;
    seq[0] = nvpairs[0];
    seq[1] = nvpairs[1];
    seq[2] = nvpairs[2];
    // 動的構造体を作成します。
    DynamicAny::DynStruct_var info =
        DynamicAny::DynStruct::_narrow(
            factory->create_dyn_any_from_type_code(
                          Printer::_tc_StructType));
    info->set_members(seq);
    manager->printAny(*(info->to_any()));
    manager->shutdown();
  }
  catch(const CORBA::Exception& e) {
      cerr << "Caught " << e << "Exception" << endl;
  }
}
```

DynAny サンプルサーバーアプリケーション

次のサンプルコードは, 次の VisiBroker 配布ディレクトリに置かれているサーバーアプリケーションを示します。

```
<install_dir>¥examples¥vbe¥dynany¥
```

このサーバーアプリケーションは，次の手順を実行します。

1. VisiBroker ORB を初期化する。
2. POA のポリシーを作成する。
3. PrintManager オブジェクトを作成する。
4. PrintManager オブジェクトをエキスポートする。
5. メッセージを出力し，オペレーションリクエストの着信を待つ。

```
...
int main(int argc, char* const* argv) {
  try {
    // ORB を初期化します。
     CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
    int Verbose = 0;
    // ルート POA へのリファレンスを取得します。
     PortableServer::POA_var rootPOA =
         PortableServer::POA::_narrow(
             orb->resolve_initial_references("RootPOA"));
     CORBA::PolicyList policies;
     policies.length(1);
     policies[(CORBA::ULong)0] = rootPOA->create_lifespan_policy(
             PortableServer::PERSISTENT);
     // 適切なポリシーで serverPOA を作成します。
     PortableServer::POA_var serverPOA = rootPOA->create_POA( "serverPoa",
             rootPOA->the_POAManager(),
             policies );
     // 動的 Any ファクトリを解決します。
     DynamicAny::DynAnyFactory_var factory =
             orb->resolve_initial_references("DynAnyFactory");
     PortableServer::ObjectId_var managerId =
             PortableServer::string_to_ObjectId("PrinterManager");
     // PrinterManager オブジェクトを作成します。
     PrinterManagerImpl manager( orb, factory, serverPOA, managerId);
     // 新しく作成したオブジェクトをエキスポートします。
     serverPOA->activate_object_with_id(managerId,&manager);
     // POA マネージャをアクティブ化します。
     rootPOA->the_POAManager()->activate();
     cout << serverPOA->servant_to_reference(&manager)
             << " is ready" << endl;
     // 着信要求を待機します。
     orb->run();
  }
  catch(const CORBA::Exception& e) {
    cerr << e << endl;
  }
}
```

次のサンプルコードは，PrinterManager のインプリメンテーションを示します。ここでは，次の手順にしたがい，DynAny を使って Any オブジェクトを処理します。コンパイル時には，この Any が保持する型はわかっていません。

1. DynAny オブジェクトを作成し，受け取った Any を使って初期化する。
2. DynAny オブジェクトの型に対して switch を実行する。
3. DynAny に基本データ型が含まれる場合は，その値を出力する。
4. DynAny に Any 型が含まれる場合は，DynAny を作成し，その内容を判定して値を出力する。
5. DynAny に enum が含まれる場合は，DynEnum を作成し，文字列値を出力する。
6. DynAny に共用体が含まれる場合は，DynUnion を作成し，共用体のディスクリミネータと番号を出力する。
7. DynAny に struct，array，または sequence が含まれる場合は，含まれるコンポーネントに順にアクセスして，各値を出力する。

```
// PrinterManager インプリメンテーション
class PrinterManagerImpl : public POA_Printer::PrinterManager
{
   CORBA::ORB_var                _orb;
   DynamicAny::DynAnyFactory_var  _factory;
   PortableServer::POA_var        _poa;
   PortableServer::ObjectId_var   _oid;
```

```
public:
   PrinterManagerImpl(CORBA::ORB_ptr orb,
      DynamicAny::DynAnyFactory_ptr DynAnyFactory,
      PortableServer::POA_ptr poa,
      PortableServer::ObjectId_ptr oid
      ) : _orb(orb), _factory(DynAnyFactory),
          _poa(poa), _oid(oid) {}

   void printAny(const CORBA::Any& info) {
    try {
       // DynAny オブジェクトを作成します。
      DynamicAny::DynAny_var dynAny = _factory->create_dyn_any(info);
        display(dynAny);
     }
     catch (CORBA::Exception& e) {
        cout << "Unable to create Dynamic Any from factory" << endl;
     }
   }
    void shutdown() {
    try {
       _poa->deactivate_object(_oid);
        cout << "Server shutting down..." << endl;
       _orb->shutdown(0UL);
     }
   catch (const CORBA::Exception& e) {
        cout << e << endl;
     }
   }

   void display(DynamicAny::DynAny_var value)  {
     switch(value->type()->kind()) {
     case CORBA::tk_null:
      case CORBA::tk_void: {
       break;
      }
      case CORBA::tk_short: {
        cout << value->get_short() << endl;
        break;
      }
      case CORBA::tk_ushort: {
        cout << value->get_ushort() << endl;
        break;
      }
      case CORBA::tk_long: {
        cout << value->get_long() << endl;
        break;
      }
      case CORBA::tk_ulong: {
        cout << value->get_ulong() << endl;
        break;
      }
      case CORBA::tk_float: {
        cout << value->get_float() << endl;
        break;
      }
      case CORBA::tk_double: {
        cout << value->get_double() << endl;
        break;
      }
      case CORBA::tk_boolean: {
```

```
  cout << value->get_boolean() << endl;
  break;
}
case CORBA::tk_char: {
  cout << value->get_char() << endl;
  break;
}
case CORBA::tk_octet: {
  cout << value->get_octet() << endl;
  break;
}
case CORBA::tk_string: {
  cout << value->get_string() << endl;
  break;
}
case CORBA::tk_any: {
  DynamicAny::DynAny_var dynAny = _factory->create_dyn_any(*(
                value->get_any()));
  display(dynAny);
  break;
}
case CORBA::tk_TypeCode: {
  cout << value->get_typecode() << endl;
  break;
}
case CORBA::tk_objref: {
  cout << value->get_reference() << endl;
  break;
}
case CORBA::tk_enum: {
  DynamicAny::DynEnum_var dynEnum = DynamicAny::DynEnum::_narrow(value);
  cout << dynEnum->get_as_string() << endl;
  break;
}
case CORBA::tk_union: {
 DynamicAny::DynUnion_var dynUnion = DynamicAny::DynUnion::_narrow(value);
  display(dynUnion->get_discriminator());
  display(dynUnion->member());
  break;
}
case CORBA::tk_struct:
case CORBA::tk_array:
case CORBA::tk_sequence: {
  value->rewind();
CORBA::Boolean next = 1UL;
  while(next) {
  DynamicAny::DynAny_var d = value->current_component();
    display(d);
    next = value->next();
  }
  break;
}
case CORBA::tk_longlong: {
  cout << value->get_longlong() << endl;
  break;
}
case CORBA::tk_ulonglong: {
  cout << value->get_ulonglong() << endl;
  break;
}
```

```
        case CORBA::tk_wstring: {
          cout << value->get_wstring() << endl;
          break;
        }
        case CORBA::tk_wchar: {
          cout << value->get_wchar() << endl;
          break;
        }
        default:
          cout << "Invalid Type" << endl;
      }
    }
};
```

第 29 章

valuetype の使い方

ここでは，VisiBroker における `valuetype` IDL 型の使い方を説明します。

valuetype について

`valuetype` IDL 型は，状態データを送信して渡すために使用されます。*valuetype* は，継承関係とメソッドを持つ構造体と考えることができます。valuetype が通常のインターフェースと違う点は，その状態を記述するプロパティを持つこと，およびインターフェースより詳細なインプリメンテーションを持つことです。

valuetype IDL サンプルコード

次に，簡単な `valuetype` を宣言する IDL コードを示します。

```
module Map {
   valuetype Point {
      public long x;
      public long y;
      private string label;
      factory create (in long x, in long y, in string z);
      void print();
   };
};
```

valuetype は常にローカルです。これらが ORB に登録されることはありません。また，各 valuetype は値で識別されるので，識別情報も不要です。valuetype をリモートで呼び出すことはできません。

具象 valuetype

具象 valuetype には状態データを格納します。これにより，IDL 構造体の機能が大幅に拡張されます。次の機能が付加されます。

- 単一の具象 valuetype の派生と複数の抽象 valuetype の派生
- 複数のインターフェースのサポート（1 つの具象 valuetype と複数の抽象 valuetype）

- 再帰的な valuetype 定義
- null 値セマンティクス
- 共有セマンティクス

valuetype の派生

1 つの具象 valuetype は，別の 1 つの具象 valuetype から派生させることができます。ただし，別の複数の抽象 valuetype から複数の valuetype を派生させることができます。

共有セマンティクス

valuetype のインスタンスは，ほかの valuetype のインスタンス内で，または複数のインスタンスにわたって共有できます。`struct`，`union`，`sequence` などのほかの IDL データ型は共有できません。共有されている valuetype は，送信コンテキストと受信コンテキストで*同じ*構造を持ちます。

また，1 つのオペレーションの複数の引数に同じ valuetype が渡されると，受信コンテキストは両方の引数に同じ valuetype リファレンスを受け取ります。

null セマンティクス

構造体，共用体，シーケンスなどの IDL データ型とは異なり，*null* の valuetype を送信して渡すことができます。たとえば，構造体をボックス化された valuetype としてボックス化することにより，null 値の構造体を渡すことができます。詳細については，391 ページの「ボックス化 valuetype」を参照してください。

ファクトリ

*ファクトリ*は valuetype 内で宣言されるメソッドで，移植性のある valuetype を作成できます。ファクトリの詳細については，390 ページの「ファクトリの実装」を参照してください。

抽象 valuetype

抽象 valuetype はメソッドだけを保持し，状態情報を持ちません。また，これをインスタンス化することはできません。抽象 valuetype は，完全にローカルで実装されるオペレーションのシグニチャをまとめたものです。

たとえば，次の IDL は状態を持たず，メソッド `get_name` が 1 つある抽象 valuetype `Account` を定義します。

```
abstract valuetype Account{
   string get_name();
}
```

次に，この `abstract valuetype` から `get_name` メソッドを継承する 2 つの valuetype を定義します。

```
valuetype savingsAccount:Account{
   private long balance;
}
valuetype checkingAccount:Account{
   private long balance;
}
```

この 2 つの valuetype は，変数 `balance` を持ち，抽象 valuetype `Account` から `get_name` メソッドを継承します。

valuetype の実装

アプリケーションで valuetype を実装するには，次の手順にしたがいます。

1. IDL ファイルで valuetype を定義します。
2. idl2cpp を使用して，IDL ファイルをコンパイルします。
3. valuetype 基底クラスを継承して，valuetype を実装します。
4. Factory クラスを実装して，IDL に定義されているファクトリメソッドを実装します。
5. create_for_unmarshal メソッドを実装します。
6. ファクトリを VisiBroker ORB に登録します。
7. _add_ref，_remove_ref，_ref_countvalue のいずれかのメソッドを実装するか，CORBA::DefaultValueRefCountBase から派生させます。

valuetype の定義

IDL のサンプル（387 ページの「valuetype IDL サンプルコード」を参照）では，グラフ上の点を定義する Point という名前の valuetype を定義します。これには，x 座標と y 座標を表す 2 つの public 変数，点の label を定義する private 変数，この valuetype の factory，および点を出力する print メソッドがあります。

IDL ファイルのコンパイル

IDL を定義したら，idl2cpp を使ってコンパイルし，ソースファイルを作成します。次に，valuetype を実装するようにソースファイルを変更します。

387 ページの「valuetype IDL サンプルコード」に示される IDL をコンパイルすると，出力は次のファイルで構成されます。

- Map_c.cc
- Map_c.hh
- Map_s.cc
- Map_s.hh

valuetype 基底クラスの継承

IDL のコンパイルが完了したら，valuetype のインプリメンテーションを作成します。インプリメンテーションクラスは基底クラスを継承します。このクラスには，ValueFactory で呼び出され，IDL で宣言されているすべての変数とメソッドを持つコンストラクタがあります。

obv¥PointImpl.java の PointImpl クラスは，IDL から生成された Point クラスを拡張します。

valuetype 基底クラスの継承

```
class PointImpl : public Map::OBV_Point, public CORBA::DefaultValueRefCountBase {
   public:
      PointImpl(){}
      virtual ~PointImpl(){}
      CORBA_ValueBase* _copy_value() {
         return new PointImpl(x(), y(), new Map::Label(
              CORBA::string_dup(label())));
      }
      PointImpl( CORBA::Long x, CORBA::Long y, Map::Label_ptr label )
            : OBV_Point( x,y,label->_boxed_in())
      {}
      virtual void print() {
         cout << "Point is [" << label() << ": ("
              << x() << ", " << y() << ")]" << endl << endl;
```

```
      }
};
```

Factory クラスの実装

インプリメンテーションクラスを作成したら，valuetype の Factory を実装します。

次の例では，生成された Point_init クラスに，IDL で宣言した create メソッドを組み込みます。このクラスは，CORBA::ValueFactoryBase を拡張します。次の例で示すように，PointDefaultFactory クラスは PointValueFactory を実装します。

```
class PointFactory: public CORBA::ValueFactoryBase {
   public:
      PointFactory(){}
      virtual  ~PointFactory(){}
      CORBA::ValueBase* create_for_unmarshal() {
         return new PointImpl();
      }
};
```

Point_init には public メソッド create_for_unmarshal が含まれています。これは，Map_c.hh に純粋仮想メソッドとして出力されます。Point_init からクラスを派生し，create_for_unmarshal メソッドを実装して，Factory クラスを生成します。IDL ファイルをコンパイルしても，このクラスのスケルトンクラスは作成されません。

ファクトリを VisiBroker ORB に登録

ファクトリを VisiBroker ORB に登録するには，ORB::register_value_factory を呼び出します。ファクトリの登録の詳細については，391 ページの「valuetype の登録」を参照してください。

ファクトリの実装

VisiBroker ORB は，受け取った valuetype をアンマーシャリングし，その型の新しいインスタンスを作成するために適切なファクトリを探す必要があります。インスタンスが作成されると，値データがアンマーシャリングされ，そのインスタンスに格納されます。型は，起動の一環として渡される RepositoryID によって識別されます。型とファクトリのマッピングは，言語によって異なります。

VisiBroker 4.5 以降のバージョンは，JDK 1.3 または JDK 1.4 のデフォルトの値ファクトリメソッドに対する正しいシグニチャを生成します。既存（バージョン 4.0）の生成コードは，下記のようにデフォルトの値ファクトリメソッドのシグニチャを変更しない限り，JDK 1.3 の下で実行できません。デフォルトの値ファクトリを変更せずに既存のコードを JDK 1.3 の下で使用すると，コードがコンパイルされないか，NO_IMPLEMENT 例外が生成されます。既存（4.0）の生成コードの場合は，正しいシグニチャが生成されるようにコードを再生成してください。

次のサンプルコードは，JDK 1.3 の下でコンパイルできるように，デフォルトの値ファクトリメソッドのシグニチャを変更する方法を示します。

```
class PointFactory: public CORBA::ValueFactoryBase
{
public:
   PointFactory(){}
   virtual  ~PointFactory(){}
   CORBA::ValueBase*  create_for_unmarshal() {
      return new PointImpl();
   }
};
```

ファクトリと valuetype

VisiBroker ORB は，valuetype を受け取ると，その型のファクトリを探します。ORB は，`<valuetype>DefaultFactory` という名前のファクトリを探します。たとえば，Point valuetype のファクトリは `PointDefaultFactory` です。正しいファクトリがこの命名規則（`<valuetype>DefaultFactory`）に沿っていない場合は，VisiBroker ORB が valuetype のインスタンスを作成できるように，正しいファクトリを登録してください。

指定された valuetype の正しいファクトリが見つからない場合は，`MARSHAL` 例外が生成され，識別されたマイナーコードとともに返されます。

valuetype の登録

登録の方法とタイミングは，各言語のマッピングによって決まります。`<valuetype>DefaultFactory` の命名規則に沿ってファクトリを作成すれば，このファクトリを暗黙的に ORB に登録していることになるので，明示的に登録する必要はありません。

`<valuetype>DefaultFactory` 命名規則にしたがっていないファクトリを登録するには，`register_value_factory` を呼び出します。ファクトリの登録を解除するには，VisiBroker ORB で `unregister_value_factory` を呼び出します。VisiBroker ORB で `lookup_value_factory` を呼び出して，登録済みの valuetype を探すこともできます。

ボックス化 valuetype

valuetype の*ボックス化*を使用すると，valuetype でない IDL データ型を valuetype としてラップできます。たとえば，次のようにボックス化 IDL valuetype を宣言します。

```
valuetype Label string;
```

この宣言は，次の IDL valuetyepe 宣言と同じです。

```
valuetype Label{
   public string name;
}
```

ほかのデータ型を valuetype としてボックス化すると，valuetype の null セマンティクスと共有セマンティクスを使用できます。

ボックス化 valuetype は，生成されるコードで完全に実装されます。ユーザーのコードは不要です。

抽象インターフェース

抽象インターフェースを使用すると，オブジェクトを値と参照のどちらで渡すかを実行時に選択できます。

抽象インターフェースは，次の点で IDL インターフェースと異なります。

- リファレンスによってオブジェクトが渡されるか，valuetype が渡されるかは，実際のパラメータの型によって決まります。パラメータの型は 2 つの規則に基づいて決められます。パラメータが標準型のインターフェースまたはそのサブタイプで，そのインターフェースの型がシグニチャ抽象インターフェース型のサブタイプであり，オブジェクトがすでに ORB に登録されている場合，そのパラメータはオブジェクトリファレンスとして扱われます。オブジェクトリファレンスとしては渡せなくても，値として渡すことができれば，値として扱われます。値として渡せない場合は，`BAD_PARAM` 例外になります。
- 抽象インターフェースが `CORBA::Object` から暗黙的に派生することはありません。抽象インターフェースはオブジェクトリファレンスまたは valuetype を表現できるからです。valuetype は，共通オブジェクトリファレンスオペレーションを必ずしもサポート

しません。抽象インターフェースをオブジェクトリファレンス型に正常にナローイングできた場合は，CORBA::Object のオペレーションを呼び出すことができます。

- 抽象インターフェースを派生できるのは，ほかの抽象インターフェースからだけです。
- valuetype は，1 つ以上の抽象インターフェースをサポートできます。

たとえば，次の抽象インターフェースを参照してください。

```
abstract interface ai{
};
interface itp : ai{
};
valuetype vtp supports ai{
};
interface x {
   void m(ai aitp);
};
valuetype y {
   void op(ai aitp);
};
```

メソッド m の引数の場合：

- オブジェクトリファレンスとして，常に itp が指定されます。
- vtp が値として指定されます。

custom valuetype

IDL で custom valuetype を宣言すると，デフォルトのマーシャリングおよびアンマーシャリングを使用しないで，独自にエンコーディングとデコーディングを行うことができます。

```
custom valuetype customPoint{
   public long x;
   public long y;
   private string label;
   factory create(in long x, in long y, in string z);
};
```

CustomMarshal インターフェースの marshal メソッドと unmarshal メソッドを実装する必要があります。

custom valuetype を宣言した場合，この valuetype は CORBA::CustomValue を拡張します。標準の valuetype が拡張する CORBA::StreamableValue とは異なります。コンパイラは，custom valuetype に読み取りまたは書き込み用のメソッドを生成しません。

CORBA::DataInputStream と CORBA::DataOutputStream でそれぞれ値を読み取り，または書き込んで，独自の読み取りまたは書き込み用のメソッドを実装する必要があります。

truncatable valuetype

truncatable な valuetype を使用すると，継承された valuetype をその親として扱うことができます。

次の IDL は，ベース型の Account を継承し，受信側のオブジェクトによって truncatable な valuetype checkingAccount を定義します。

```
valuetype checkingAccount: truncatable Account{
   private long balance;
}
```

これは，受信コンテキストが派生 valuetype の新しいデータメンバーやメソッドを必要としない場合，および受信コンテキストが派生 valuetype を認識できない場合に便利です。

ただし, 派生 valuetype の状態データのうち, 親のデータ型にないものは, この valuetype が受信コンテキストに渡されるときに失われます。

メモ　custom valuetype は truncatable にできません。

第 30 章

双方向通信

ここでは，GateKeeper を使用せず，VisiBroker を使って双方向接続を確立する方法について説明します。GateKeeper を使った双方向通信については，『Gatekeeper の概要』を参照してください。

メモ　双方向 IIOP を有効にする前に，399 ページの「セキュリティに関する注意」を参照してください。

双方向 IIOP の使用

インターネット経由で情報を交換するほとんどのクライアントとサーバーは，通常，企業のファイアウォールで保護されています。クライアントだけが要求を開始するシステムの場合，通常，ファイアウォールの存在はクライアントに対して透過的です。ただし，クライアントが情報を「非同期に」必要とする場合があります。これは，要求への応答という形以外で情報を受け取る場合です。クライアント側のファイアウォールは，サーバーがクライアントへの接続を開始しないようにします。したがって，クライアントが非同期に情報を受け取る場合は，追加の設定が必要になります。

以前のバージョンの IIOP と VisiBroker では，サーバーからクライアントへ非同期に情報を送信するには，クライアント側の GateKeeper を使ってサーバーからのコールバックを処理する方法しかありませんでした。

双方向 IIOP を使用すると，情報を非同期にクライアント側に送るときに，サーバーからクライアントへの接続を開くかわりに（いずれにしてもクライアント側のファイアウォールから拒否されますが），サーバーはクライアントによって開始された接続を使って情報をクライアントに伝送します。CORBA 仕様でも，この機能を簡単に制御できる新しいポリシーを追加しています。

双方向 IIOP を使用すると，GateKeeper がなくてもコールバックを設定できるため，クライアントの配布が非常に容易になります。

双方向 VisiBroker ORB のプロパティ

次のプロパティが双方向通信をサポートします。

396 ページの「enableBiDir プロパティ」

396 ページの「exportBiDir プロパティ」

396 ページの「importBiDir プロパティ」

enableBiDir プロパティ

vbroker.orb.enableBiDir プロパティは，双方向通信を有効にするためにサーバーとクライアントの両方で使用できます。このプロパティを使用すると，コードを変更せずに既存の一方向アプリケーションを双方向アプリケーションに変更できます。次の表では，vbroker.orb.enableBiDir プロパティの値オプションについて説明します。

表 30.1　enableBiDir プロパティの値

値	説明
client	すべての POA と発信接続に対して，双方向 IIOP を有効にします。この設定は，すべての POA の作成において BiDirectional ポリシーを both に設定し，BiDirectional ポリシーのポリシーオーバーライドを VisiBroker ORB レベルで both に設定するのと同じです。さらに，作成した SCM はすべて，各 SCM の exportBiDir プロパティが true に設定されているかのように双方向接続を許可します。
server	サーバーで双方向接続を受け付け，使用できるようになります。これは，SCM すべての importBiDir プロパティを true に設定するのと同じです。
both	このプロパティを client と server の両方に設定します。
none	双方向 IIOP を完全に無効にします（デフォルト値）。

exportBiDir プロパティ

vbroker.se.<sename>.scm.<scmname>.manager.exportBiDir プロパティは，クライアント側のプロパティです。デフォルトでは，VisiBroker ORB がこのプロパティをどちらかの値に設定することはありません。

このプロパティを true に設定すると，指定したサーバーエンジンで双方向コールバック POA を作成できるようになります。

このプロパティを false に設定すると，指定したサーバーエンジンで双方向 POA を作成できなくなります。

importBiDir プロパティ

vbroker.se.<se-name>.scm.<scm-name>.manager.importBiDir プロパティは，サーバー側のプロパティです。デフォルトでは，VisiBroker ORB がこのプロパティをどちらかの値に設定することはありません。

このプロパティを true に設定すると，サーバー側はすでにクライアントによって確立されている接続を再利用して，クライアントに要求を送信できます。

このプロパティを false に設定すると，接続の再利用は無効になります。

メモ　これらのプロパティは，SCM の作成時に一度だけ評価されます。どのような場合でも，SCM の exportBiDir プロパティと importBiDir プロパティの方が enableBiDir プロパティより優先されます。つまり，両方のプロパティに相反する値を設定すると，SCM 固有のプロパティが適用されます。このため，enableBiDir プロパティをグローバルに設定して，各 SCM では選択的に BiDir をオフにすることができます。

双方向サンプルについて

この機能の使い方を示すサンプルは，VisiBroker 配布の一部として次のサブディレクトリにインストールされています。

```
<install_dir>¥examples¥vbe¥bidir-iiop
```

どのサンプルも，次のような簡単な株価情報コールバックアプリケーションをベースにしています。

1 クライアントで株価情報の更新を処理する CORBA オブジェクトを作成する。

2 クライアントで，この CORBA オブジェクトのオブジェクトリファレンスをサーバーに送信する。

3 サーバーでこのコールバックオブジェクトを呼び出して，定期的に株価情報を更新する。

次に，これらのサンプルを使用して，双方向 IIOP 機能のさまざまな面について説明します。

既存のアプリケーションで双方向 IIOP を有効にする

ソースコードを変更しなくても，既存の VisiBroker for Java アプリケーションと VisiBroker for C++ アプリケーションで双方向通信を有効にできます。双方向 IIOP をまったく使用しないシンプルなコールバックアプリケーションが examples/vbe/bidir-iiop/basic/ ディレクトリに格納されています。

```
<install_dir>¥examples¥vbe¥bidir-iiop¥basic
```

コールバックサンプルで双方向 IIOP を有効にするには，次のように vbroker.orb.enableBiDir プロパティを設定します。

1 osagent が実行されているかどうかを確認します。

2 サーバーを起動します。

UNIX：

```
prompt> -Dvbroker.orb.enableBiDir=server Server &
```

Windows：

```
prompt> start -Dvbroker.orb.enableBiDir=server Server
```

3 クライアントを起動します。

```
prompt> -Dvbroker.orb.enableBiDir=client RegularClient
```

これで，既存のコールバックアプリケーションで双方向 IIOP を使用できるようになり，クライアント側のファイアウォールを介して機能します。

双方向 IIOP を明示的に有効にする

<install_dir>¥examples¥vbe¥bidir-iiop¥basic ディレクトリ内のクライアントは，397 ページの「既存のアプリケーションで双方向 IIOP を有効にする」で説明した RegularClient から派生されます。ただし，双方向 IIOP がプログラムによって有効にされるという点が異なります。

変更する必要があるのはクライアントコードだけです。一方向クライアントを双方向クライアントに変換するには，次の操作を実行します。

1 コールバック POA のポリシーリストに BiDirectional ポリシーを追加します。

2 双方向 IIOP を有効にするサーバーを参照するオブジェクトリファレンスのオーバーライドリストに，BiDirectional ポリシーを追加します。

3 クライアントで exportBiDir プロパティを true に設定します。

次のサンプルコードでは，双方向 IIOP を実装するためのコードが太字で示されます。

```
try {

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

   // マネージャの ID を取得します。
   PortableServer::ObjectId_var managerId =
       PortableServer::string_to_ObjectId("BankManager");
   PortableServer::ObjectId_var oid =
       PortableServer::string_to_ObjectId("QuoteServer");
   Quote::QuoteServer_var quoter =
       Quote::QuoteServer::_bind("/QuoteServer_poa", oid);

   // コールバックオブジェクトのセットアップ ... 最初に RootPOA を取得します。
```

```
    CORBA::Object_var obj =
       orb->resolve_initial_references("RootPOA");
    PortableServer::POA_var rootPOA =
       PortableServer::POA::_narrow(obj);
    PortableServer::POAManager_var the_manager =
       rootPOA->the_POAManager();
    PortableServer::POA_var consumer_poa;

   // ポリシーをセットアップします。
   CORBA::Any policy_value;
   policy_value <<= BiDirPolicy::BOTH;
   CORBA::Policy_var policy =
       orb->create_policy(
           BiDirPolicy::BIDIRECTIONAL_POLICY_TYPE,
           policy_value);
   CORBA::PolicyList policies;
   policies.length(1);
   policies[0] = CORBA::Policy::_duplicate(policy);
   consumer_poa = rootPOA->create_POA(
       "QuoteConsumer_poa", the_manager, policies );
   QuoteConsumerImpl* consumer = new QuoteConsumerImpl;
   oid = PortableServer::string_to_ObjectId("consumer");
   consumer_poa->activate_object_with_id(oid, consumer);
   the_manager->activate();
   CORBA::Object_var obj =
       quoter->set_policy_overrides(policies, CORBA::ADD_OVERRIDE);
   quoter = Quote::QuoteServer::_narrow(obj);
   obj = consumer_poa->id_to_reference(oid);
   Quote::QuoteConsumer_var quote_consumer =
       Quote::QuoteConsumer::_narrow(obj);
   quoter->registerConsumer(quote_consumer.in());
   cout << "implementation is running" << endl;
   orb->run();
}
catch(const CORBA::Exception& e) {
   cout << e << endl;
}
```

一方向または双方向接続

クライアント接続は，一方向か双方向のどちらかになります。サーバーは新しく接続を開かなくても，双方向接続を使ってクライアントをコールバックすることができます。双方向でない場合，接続は一方向とみなされます。

POA で双方向 IIOP を有効にする

コールバックオブジェクトをホストする POA は, BiDirectional ポリシーを BOTH に設定して双方向 IIOP を有効にする必要があります。この POA は SCM マネージャの vbroker.<sename>.scm.<scmname>.manager.exportBiDir プロパティを設定して，双方向サポートがすでに有効になっている SCM 上に作成する必要があります。このように設定しないと，クライアントで開始された接続を使用するサーバーからの要求を受信することができません。

POA が BiDirectional ポリシーを指定していない場合，その POA を発信接続でエクスポーズすることはできません。この要件を満たすため，exportBiDir プロパティが設定されている SCM を 1 つでも持つサーバーエンジンでは, BiDirectional ポリシーが設定されていない POA を作成することはできません。一方向 SE で POA を作成しようとすると，ServerEnginePolicy でエラーが発生して InvalidPolicy 例外が生成されます。

メモ 同じクライアント接続を使用する異なるオブジェクトどうしが, BiDirectional ポリシーについて競合するオーバーライドを設定できます。その場合でも，いったん双方向の接続が作成されると，後で有効になるポリシーに関係なくその接続は双方向性を維持します。

双方向設定を完全に適用したら，次のように iiop_tp SCM 上だけで双方向 IIOP を有効にします。

```
prompt> -Dvbroker.se.iiop_tp.scm.iiop_tp.manager.exportBiDir=
true Client
```

セキュリティに関する注意

双方向 IIOP を使用すると，セキュリティに関する重大な問題が発生する可能性があります。特にセキュリティメカニズムが設定されていない場合は，悪意のあるクライアントがホストとポートを任意に選択して，双方向接続を要求する可能性があります。また，自分のホストにはない，セキュリティ上重要なオブジェクトのホストとポートをクライアントが指定する場合もあります。さらにセキュリティメカニズムが設定されていないと，着信接続を受け付けたサーバーは，接続要求元のクライアントの ID を識別したり，クライアントの完全性を検査できません。また，サーバーが双方向接続を介してほかのオブジェクトにアクセスできる可能性があります。以上のことから，コールバックオブジェクトごとに独立した双方向 SCM を使用してください。クライアントの完全性に疑問がある場合は，双方向 IIOP を使用しないでください。

セキュリティ上の理由から，VisiBroker を実行するサーバーは，双方向 IIOP を使用するように明示的に設定されていない限り，双方向 IIOP を使用しません。プロパティ vbroker.<se>.<sename>.scm.<scmname>.manager.importBiDir を使用すると，SCM 単位で双方向性を制御できます。たとえば，SSL を使ってクライアントを認証するサーバーエンジンだけで双方向 IIOP を有効にし，その他の通常の IIOP 接続は双方向で使用できないように選択することもできます。詳細については，395 ページの「双方向 VisiBroker ORB のプロパティ」を参照してください。さらに，クライアントファイアウォール外でコールバックを行うサーバーとの双方向接続だけをクライアント側で有効にすることもできます。クライアントとサーバー間に高度なセキュリティを確立するには，相互認証（クライアントとサーバーの両方で vbroker.security.peerAuthenticationMode を REQUIRE_AND_TRUST に設定）の SSL を使用します。

第 31 章

VisiBroker における BOA の使い方

ここでは，VisiBroker で BOA を使用する方法について説明します。

メモ　BOA は，VisiBroker バージョン 4.0（CORBA 仕様 2.1）と 3.x バージョンに対する下位互換性としてサポートされます。現在の CORBA 仕様のサポートについては，第 9 章「POA の使い方」を参照してください。

VisiBroker を使った BOA コードのコンパイル

VisiBroker の以前のバージョンで開発した既存の BOA コードがある場合は，現在のバージョンでもそれを使用できます。

メモ　必要な BOA ベースのコードを生成するには，-boa オプション付きで idl2cpp ツールを使用する必要があります。idl2cpp を使ってコードを生成する方法の詳細については，第 5 章「IDL から C++ へのマッピング」を参照してください。

BOA オプションのサポート

VisiBroker 4.x でサポートされていた BOA コマンドラインオプションは，すべてそのままサポートされます。

オブジェクトアクティベータの使い方

VisiBroker でサポートされていた BOA オブジェクトアクティベータは BOA でのみそのまま使用できますが，POA では使用できません。POA では，オブジェクトアクティベータのかわりにサーバントアクティベータとサーバントロケータを使用します。

VisiBroker 3.x の BOA によって提供されていた機能は，このバージョンでは Portable Object Adaptor（POA）によってサポートされます。ただし，コードの下位互換性を保つため，コードで引き続きオブジェクトアクティベータを使用することもできます。

BOA の下でのネーミングオブジェクト

BOA は VisiBroker 4.x で使用されなくなっていますが，現在でも，クライアントプログラム内でバインド先となるサーバーオブジェクトの名前を指定するために, BOA をスマートエージェントと組み合わせて使用できます。

オブジェクト名

クライアントアプリケーションが osagent を介してオブジェクトを使用するには，サーバーがそのオブジェクトの作成時にオブジェクトの名前を指定している必要があります。サーバーが `BOA.obj_is_ready` メソッドを呼び出すと，オブジェクトに名前が付いている場合にだけ，オブジェクトのインターフェース名が VisiBroker の `osagent` に登録されます。作成時にオブジェクト名を指定されたオブジェクトは，**永続的な**オブジェクトリファレンスを返します。一方，オブジェクト名を指定されなかったオブジェクトは，**一時的な**オブジェクトとして作成されます。

メモ　VisiBroker for C++ のオブジェクトコンストラクタにオブジェクト名として空文字列を渡すと，永続的オブジェクト（スマートエージェントに登録されるオブジェクト）が作成されます。null リファレンスをオブジェクトコンストラクタに渡した場合は，一時的オブジェクトが作成されます。

1 つのオブジェクトの複数のインスタンスに一度にバインドすることがある場合, クライアントアプリケーションでオブジェクト名を使用する必要があります。そのオブジェクト名により, 1 つのインターフェースの複数のインスタンスが区別されます。オブジェクト名を指定しないで `bind()` メソッドが呼び出された場合，osagent は，指定されたインターフェースを持つ任意のオブジェクトを返します。

メモ　VisiBroker 3.x では，1 つのサーバープロセスで，同じオブジェクト名を持つ複数のインターフェースを提供できました。ただし，現在のバージョンの VisiBroker では，異なるインターフェースが文字列として等価な名前を持つことはできません。

第 32 章

オブジェクトアクティベータの使い方

ここでは，VisiBroker のオブジェクトアクティベータの使い方について説明します。

このリリースのポータブルオブジェクトアダプタ（POA）は，VisiBroker 4.1 以降のリリースと同様に，VisiBroker 3.x および 4.0 リリースの BOA で提供されていた機能をサポートします。下位互換性があるため，この節で説明するようにコード内でオブジェクトアクティベータを使用できます。このリリースでの BOA アクティベータの使い方については，第 31 章「VisiBroker における BOA の使い方」を参照してください。

オブジェクトのアクティブ化の遅延

1 つのサーバーが多数のオブジェクトに対してインプリメンテーションを提供する場合は，単一の Activator だけを使用し，サービスのアクティブ化を使用して，複数のオブジェクトインプリメンテーションのアクティブ化を遅らせることができます。

Activator インターフェース

Activator クラスから独自のインターフェースを派生すると，VisiBroker ORB が AccountImpl オブジェクトに対して使用する純粋仮想 activate メソッドと deactivate メソッドを実装できます。これで，BOA が AccountImpl オブジェクトに対するリクエストを受け取るまで，そのインスタンス化を遅らせることができます。また，BOA がオブジェクトを非アクティブ化したときに，クリーンアップ処理を行うことができます。

次のサンプルコードは Activator クラスを示します。

```
class Activator {
   public:
      virtual CORBA::Object_ptr activate(
         extension::ImplementationDef impl)=0;
      virtual void deactivate(
         Object_ptr, extension::ImplementationDef_ptr impl)=0;
};
```

次のサンプルコードは，AccountImpl インターフェースの Activator を作成します。

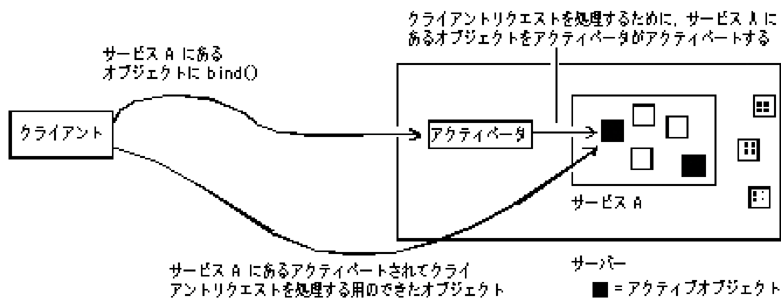
```
class extension {
   . . .
   class AccountImplActivator : public extension::Activator {
      public:
         virtual CORBA::Object_ptr activate(
               CORBA::ImplementationDef_ptr impl);
         virtual void deactivate(CORBA::Object_ptr,
               CORBA::ImplementationDef_ptr impl);
   };
   CORBA::Object_ptr AccountImplActivator::activate(
         CORBA::ImplementationDef_ptr impl) {
      // BOA によってアクティブ化されたら，AccountImpl オブジェクトをインスタンス化しま
す。
      extension::ActivationImplDef* actImplDef =
               extension::ActivationImplDef::_downcast(impl);
      CORBA::Object_var obj = new AccountImpl(actImplDef->object_name());
      return CORBA::_duplicate(obj);
   }
   void AccountImplActivator::deactivate(CORBA::Object_ptr obj,
         CORBA::ImplementationDef_ptr impl) {
      // BOA によって非アクティブ化されたら，Account オブジェクトを解放します。
      obj->_release;
   }
}
```

サービスのアクティブ化の使い方

サービスのアクティブ化を使用するには，サーバーが大量（数千から数百万）のオブジェクトにインプリメンテーションを提供する必要があります。一度にアクティブ化するインプリメンテーションがわずかであるような場合です。サーバーは単一の Activator を提供し，部分的にオブジェクトが必要になるたびに，この Activator が通知を受けます。サーバーは，オブジェクトが使用されないときに非アクティブ化することもできます。

たとえば，データベースに状態が保存されているオブジェクトインプリメンテーションをロードするサーバーにおいて，サービスのアクティブ化を利用するとします。Activator は，型の指定または論理的な識別に基づいて，すべてのオブジェクトをロードする役割を持ちます。VisiBroker ORB がこれらのオブジェクトへのリファレンスを要求すると，Activator が通知を受けて，新しいインプリメンテーションを作成します。このインプリメンテーションの状態は，データベースからロードされます。Activator は，オブジェクトがすでにメモリ上にないか，変更されていると判断した場合，そのオブジェクトの状態をデータベースに書き込み，インプリメンテーションを解放します。

図 32.1　サービスのアクティブ化の遅延プロセス



サービスアクティベータを使ってオブジェクトのアクティブ化を遅延する

サービスを構成するオブジェクトは，すでに作成されているものとします。サービスのアクティブ化を利用するサーバーを実装するには，次の手順にしたがう必要があります。

1. Activator によってアクティブ化または非アクティブ化されるすべてのオブジェクトを表すサービスの名前を定義します。
2. インターフェースにインプリメンテーションを提供します。これは，永続的オブジェクトではなく，サービスオブジェクトになります。この処理は，オブジェクトが自分自身をアクティブ化可能なサービスの一部として構築するときに行われます。
3. オンデマンドでオブジェクトインプリメンテーションを作成する Activator を実装します。そのインプリメンテーションで，extension::Activator から Activator インターフェースを派生し，activate メソッドと deactivate メソッドをオーバーライドします。
4. BOA にサービス名と Activator インターフェースを登録します。

サービスを使ってオブジェクトのアクティブ化を遅延するサンプル

次の節では，次の VisiBroker ディレクトリ内にあるサービスのアクティブ化のための odb サンプルについて説明します。

```
<install_dir>/examples/vbe/boa/odb
```

このディレクトリには，次のファイルが入っています。

表 32.1 サービスのアクティブ化を紹介する odb サンプルのファイル

名前	説明
odb.idl	DB インターフェースと DBObject インターフェースの IDL です。
Server.C	サービスアクティベータを使ってオブジェクトを作成し，オブジェクトの IOR を返します。また，オブジェクトを非アクティブ化します。
Creator.C	DB インターフェースを呼び出し，100 個のオブジェクトを作成します。さらに，文字列化されたオブジェクトリファレンスをファイル（objref.out）に保存します。
Client.C	文字列化されたオブジェクトリファレンスをファイルから読み取り，オブジェクトを呼び出します。その結果，サーバーのアクティベータによってオブジェクトが作成されます。
Makefile	odb サブディレクトリで make または nmake（Windows の場合）が起動されたとき，次のクライアントプログラムとサーバープログラムをビルドします。 Server.exe，Creator.exe，Client.exe

odb サンプルでは，単一のサービスから任意の数のオブジェクトを作成できることがわかります。BOA には，個々のオブジェクトではなく，サービスだけが登録されます。また，各オブジェクトのリファレンスデータが IOR の一部として保存されます。このようにすると，オブジェクトキーをオブジェクトリファレンスの一部として保存できるので，オブジェクト指向データベース（OODB）の統合が容易になります。まだ作成されていないオブジェクトをクライアントが呼び出すと，BOA は，ユーザー定義の Activator を呼び出します。そこで，アプリケーションは永続的ストレージから適切なオブジェクトをロードします。

このサンプルでは，作成された Activator が，DBService という名前のサービスのオブジェクトをアクティブ化または非アクティブ化する役割を持ちます。この Activator によって作成されるオブジェクトへのリファレンスには，ORB が DBService サービスの Activator を再検索し，Activator がこれらのオブジェクトをオンデマンドで再作成するための情報が入っています。

DBService サービスは，DBObject インターフェースを実装するオブジェクトを受け持ちます。これらのオブジェクトを手動で作成するためのインターフェースが odb.idl で提供されます。

odb.idl インターフェース

odb.idl インターフェースを使用すると，DBObject odb インターフェースを実装するオブジェクトを手動で作成できます。

```
interface DBObject {
   string get_name();
};
typedef sequence<DBObject> DBObjectSequence;
```

```
interface DB {
   DBObject create_object(in string name);
};
```

DBObject インターフェースは，DB インターフェースによって作成されるオブジェクトを表し，サービスオブジェクトとして扱われます。

DBObjectSequence は DBObject のシーケンスです。サーバーは，このシーケンスを使って現在アクティブなオブジェクトを追跡します。

DB インターフェースは，create_object オペレーションを使って 1 つ以上の DBObject を作成します。DB インターフェースによって作成された複数のオブジェクトは，グループ化して 1 つのサービスにまとめることができます。

サービスアクティブ化オブジェクトの実装

idl2cpp コンパイラは，boa/odb/odb.idl から _sk_DBObject スケルトンクラスの 2 種類のコンストラクタを生成します。1 つは手動でインスタンス化されたオブジェクト用に使用するコンストラクタ，もう 1 つはオブジェクトをサービスの一部に入れるためのコンストラクタです。次に示すように，DBObject のインプリメンテーションは，手動でインスタンス化されたオブジェクトで一般に使用される object_name コンストラクタではなく，サービスコンストラクタを使ってベースの _sk_DBObject> メソッドを構築します。このコンストラクタを呼び出すことにより，DBObject は，DBService と呼ばれるサービスの一部として自分自身を構築します。

```
class DBObjectImpl: public _sk_DBObject {
   private:
      CORBA::String_var   _name;
   public:
      DBObjectImpl(const char *nm, const CORBA::ReferenceData& data)
      : _sk_DBObject("DBService", data), _name(nm) {}
   . . .
};
```

ベースコンストラクタには，サービス名のほか，不透過の CORBA::ReferenceData 値が必要です。これらのパラメータは，クライアント要求に応じてオブジェクトがアクティブ化されるとき，オブジェクトを識別するために Activator によって使用されます。このサンプルで複数のインスタンスの識別に使用されるリファレンスデータは，0 から 99 までの数値になります。

サービスアクティベータの実装

通常，オブジェクトがアクティブ化されるのは，サーバーがそのオブジェクトを実装するクラスをインスタンス化し，BOA::obj_is_ready に続けて BOA::impl_is_ready が呼び出されたときです。オブジェクトのアクティブ化を遅らせるには，BOA がオブジェクトをアクティブ化する間に呼び出す activate メソッドを制御する必要があります。この制御を取得するには，extension::Activator から新しいクラスを派生し，activate メソッドをオーバーライドし，オーバーライドした activate メソッドを使ってオブジェクト固有のクラスをインスタンス化します。

odb サンプルでは，extension::Activator から DBActivator クラスを派生し，activate メソッドと deactivate メソッドをオーバーライドします。DBObject は，activate メソッド内に構築されます。

次のサンプルコードは，activate と deactivate のオーバーライドを示します。

```
class DBActivator: public extension::Activator {
      virtual CORBA::Object_ptr activate(CORBA::ImplementationDef_ptr impl);
      virtual void deactivate(CORBA::Object_ptr,
            CORBA::ImplementationDef_ptr impl );
   public:
      DBActivator(CORBA::BOA_ptr boa) : _boa(boa) {}
   private:
      CORBA::BOA_ptr _boa;
};
```

BOA は, Activator の制御下にあるオブジェクトに対するクライアント要求を受け取ると, Activator の activate メソッドを呼び出します。このメソッドを呼び出すと，BOA は, ImplementationDef パラメータに Activator を渡すことで，アクティブ化されるオブジェクトインプリメンテーションを一意に識別します。インプリメンテーションは, このパラメータから，要求されたオブジェクトの一意の識別子である CORBA::ReferenceData を取得できます。

次のサンプルコードに示すように，DBActivator クラスは，CORBA::ReferenceData パラメータに基づいてオブジェクトを作成します。

```
CORBA::Object_ptr DBActivator::activate(CORBA::ImplementationDef_ptr impl) {
   extension::ActivationImplDef* actImplDef =
         extension::ActivationImplDef::_downcast(impl);
   CORBA::ReferenceData_var id(actImplDef->id());
   cout << "Activate called for object=[" << (char*) id->data()
         << "]" << endl;
   DBObjectImpl *obj = new DBObjectImpl((char *)id->data(), id);
   _impls.length(_impls.length() + 1);
   _impls[_impls.length()-1] =  DBObject::_duplicate(obj);
   _boa->obj_is_ready(obj);
   return obj;
}
```

サービスアクティベータのインスタンス化

DBActivator サービスアクティベータは，DBService サービスに属するすべてのオブジェクトを受け持ちます。DBService サービスのオブジェクトに対する要求は, すべて DBActivator サービスアクティベータを介して指示されます。このサービスアクティベータによってアクティブ化されたすべてのオブジェクトは，DBService サービスに属していることを VisiBroker ORB に通知するためのリファレンスを持ちます。

次のサンプルコードに示すように，DBActivator サービスアクティベータは，メインサーバープログラムで BOA::impl_is_ready を呼び出すことで作成され，BOA に登録されます。

```
int main(int argc, char **argv) {
   CORBA::ORB_ptr orb = CORBA::ORB_init(argc, argv);
   CORBA::BOA_ptr boa = orb->BOA_init(argc, argv);
   MyDB db("Database Manager");
   boa->obj_is_ready(&db);
   DBObjectImplReaper reaper;
   reaper.start();
   cout << "Server is ready to receive requests" << endl;
   boa->impl_is_ready("DBService", new DBActivator(boa));
   return(0);
}
```

メモ　BOA::impl_is ready の呼び出しは，通常の BOA::impl_is_ready 呼び出しとは異なり，次の 2 つの引数をとります。

- サービス名。
- Activator インターフェースのインスタンス。これは，BOA がこのサービスに属するオブジェクトをアクティブ化するときに使用されます。

サービスアクティベータを使ったオブジェクトのアクティブ化

オブジェクトを作成するたびに，DBActivator::activate 内で BOA::obj_is_ready を明示的に呼び出す必要があります。サーバープログラムでは BOA::obj_is_ready を 2 度呼び出します。最初は，サーバーがサービスオブジェクトを作成し，作成元のプログラムに IOR を返すときです。

```
DBObject_ptr create_object(const char *name) {
   char ref_data[100];
   memset(ref_data,'¥0',100);
   sprintf(ref_data, "%s", name);
   CORBA::ReferenceData id(100, 100, (CORBA::Octet *)ref_data);
```

```
    DBObjectImpl *obj = new DBObjectImpl(name, id);
    _boa()->obj_is_ready(obj);
    _impls.length(_impls.length() + 1);
    _impls[_impls.length()-1] =  DBObject::_duplicate(obj);
    return obj;
}
```

2 度めの BOA::obj_is_ready 呼び出しは，DBActivator::activate 内で明示的に行う必要があります。

サービスアクティブ化オブジェクトインプリメンテーションの非アクティブ化

サービスのアクティブ化を使用する主な目的は，実際に一度にアクティブ化されているオブジェクトは少数だけなのに，サーバー内で大量のオブジェクトがアクティブ化されているかのように見せることです。このモデルを実現するには，サーバーが一時的にオブジェクトの使用を停止できる必要があります。マルチスレッド DBActivator サンプルプログラムには，30 秒ごとにすべての DBObjectImpls を非アクティブ化するためのスレッドがあります。deactivate メソッドが呼び出されると，DBActivator は，単にオブジェクトリファレンスを解放します。非アクティブ化されたオブジェクトに対するクライアント要求を受け取ると，ORB は，そのオブジェクトを再びアクティブ化するように Activator に指示します。

```
// 現在アクティブなインプリメンテーションの静的シーケンス
static VISMutex         _implMtx;
static DBObjectSequence _impls;
// グローバルシーケンスにアクティブ化されたインプリメンテーションを格納するために
// 更新された DBActivator
class DBActivator: public extension::Activator {
   virtual CORBA::Object_ptr activate(CORBA::ImplementationDef_ptr impl) {
      extension::ActivationImplDef* actImplDef =
            extension::ActivationImplDef::_downcast(impl);
      CORBA::ReferenceData_var id(actImplDef->id());
      DBObjectImpl *obj = new DBObjectImpl((char *)id->data(), id);
      VISMutex_var lock(_implMtx);
      _impls.length(_impls.length() + 1);
      _impls[_impls.length()-1] =  DBObject::_duplicate(obj);
      return obj;
   }
   virtual void deactivate(CORBA::Object_ptr,
      CORBA::ImplementationDef_ptr impl) {
      obj->_release();
   }
};
// 30 秒ごとにすべてのアクティブなオブジェクトを破棄する
// マルチスレッド Reader
class DBObjectImplReaper : public VISThread {
   public:
      // Reaper のメソッド
      virtual void start() {
         run();
      }
      virtual CORBA::Boolean startTimer() {
         vsleep(30);
         return 1;
      virtual void begin() {
         while (startTimer()) {
            doOneReaping();
         }
      }
   protected:
      virtual void doOneReaping() {
         VISMutex_var lock(_implMtx);
```

```
        for (CORBA::ULong i=0; i < _impls.length(); i++) {
            // 各要素に nil を割り当てて
            // _var に格納されているリファレンスを解放します。
            DBObject_var obj = DBObject::_duplicate(_impls[i-1];
            _impls[i] = DBObject::_nil();
            CORBA::BOA_var boa = obj->_boa();
            boa->deactivate_obj(obj);
        }
        _impls.length(0);
    }
};
```

第 33 章

リアルタイム CORBA 拡張

ここでは，VisiBroker for C++ によってサポートされるリアルタイム CORBA 拡張（リアルタイム CORBA 1.0 仕様で定義）について説明します。また，リアルタイム CORBA 拡張をアプリケーションコードに適用する方法についても説明します。

メモ　リアルタイム CORBA 拡張は，次のプラットフォームでサポートされます。

- Solaris
- HP-UX
- AIX
- Linux

概要

VisiBroker for C++ は，次のリアルタイム CORBA 拡張機能を提供しています。

- **リアルタイム ORB**
  スレッドプール，ミューテックスなどの他のリアルタイム CORBA エンティティの作成と破棄を管理するために使用されます。
- **リアルタイムオブジェクトアダプタ**
  拡張版のポータブルオブジェクトアダプタ（POA）。スレッドプールとともに機能し，設定可能なリアルタイム CORBA プロパティが数多く用意されています。
- **リアルタイム CORBA 優先順位**
  VisiBroker アプリケーションに関連するスレッドの優先順位を制御するために使用されるプラットフォームに依存しない優先順位スキーム。特定の OS の優先順位スキームではなく，リアルタイム CORBA 優先順位スキームに基づいて優先順位を指定することで，リアルタイムでも非リアルタイムでも，異なる OS 上で実行されるマシン間で首尾一貫したリアルタイムアクティビティをスケジュールするアプリケーションを開発できます。これは，将来，異なる OS にアプリケーションを移植したり拡張する場合にも役立ちます。
- **優先順位マッピング**
  これは，この機能により，リアルタイム CORBA 優先順位スキームが基盤となる OS の優先順位スキームに「マッピング」されるということです。優先順位マッピングを設定

して優先順位のマップ方法を制御することも，ORB から提供される「デフォルトマッピング」を使用することもできます。

- **スレッドプール**
  CORBA 呼び出しを実行するために ORB によって使用されるスレッドをアプリケーションが制御できるようにするリアルタイム CORBA エンティティ。
- **リアルタイム CORBA Current インターフェース**
  リアルタイム CORBA 優先順位値をアプリケーションスレッドに割り当てるための CORBA::Current インターフェースの拡張。
- **リアルタイム CORBA 優先順位モデル**
  CORBA 呼び出しを実行する際の優先順位を決定する 2 つのモデル。
- **リアルタイム CORBA ミューテックス API**
  ORB によって内部的に使用されるミューテックスインプリメンテーションと同じインプリメンテーションにアプリケーションからアクセスできるようにする IDL 定義のミューテックスインターフェース。これにより，首尾一貫した優先順位継承動作が保証されるとともに，アプリケーションの可搬性が向上します。
- **内部 ORB スレッド優先順位の制御**
  ORB によって内部的に使用される追加スレッドの優先順位の範囲制限と明示的制御を行うメカニズム。

メモ　基底の OS によっては，スレッド優先順位の制御にスーパーユーザー（*root*）特権が必要です。

リアルタイム CORBA 拡張の使用

リアルタイム CORBA 拡張を使用するアプリケーションは，VisiBroker の `include` ディレクトリにある C++ ヘッダーファイル `rtcorba.h` をインクルードする必要があります。

リアルタイム CORBA 機能の多くは，IDL で定義されたインターフェースを持ちます。これらの機能の IDL は，新しい RTCORBA モジュールで指定されています。この IDL の内容は，VisiBroker のインストールディレクトリにある `idl` ディレクトリの `RTCORBA.idl` ファイルで調べることができます。

ただし，リアルタイム CORBA 機能を使用するために `RTCORBA.idl` 内の IDL をコンパイルする必要はありません。アプリケーションで必要な作業は，他の VisiBroker ヘッダーファイルとともに提供される `rtcorba.h` ヘッダーファイルをインクルードすることだけです。

これは，モジュール内のすべてのインターフェースが「局所性制約付き」と指定されているためです。つまり，それらのオブジェクトリファレンスをノードの外に渡したり，インスタンスのオペレーションをリモートに呼び出すために使用することはできません。`CORBA::ORB`，`PortableServer::POA` などの他の CORBA エンティティと同様に，すべてのリアルタイム CORBA インターフェースの操作はローカルで実行する必要があります。

リアルタイム CORBA ORB

リアルタイム CORBA 拡張には，他のリアルタイム CORBA エンティティを管理するために使用されるリアルタイム ORB インターフェースが含まれます。このインターフェースは RTCORBA::RTORB という名前で，次のように定義されています。

```
module RTCORBA {

    // 局所性制約付きインターフェース
    interface RTORB {

        Mutex create_mutex();
        void destroy_mutex( in Mutex the_mutex );
        exception InvalidThreadpool {};

        ThreadpoolId create_threadpool(
                in unsigned long stacksize,
                in unsigned long static_threads,
                in unsigned long dynamic_threads,
                in Priority default_priority,
                in boolean allow_request_buffering,
                in unsigned long max_buffered_requests,
                in unsigned long max_request_buffer_size );

        void destroy_threadpool( in ThreadpoolId threadpool )
            raises (InvalidThreadpool);

        void threadpool_idle_time(
                in ThreadpoolId threadpool,
                in unsigned long seconds )
            raises (InvalidThreadpool);
    };
};
```

この IDL に示されているオペレーションは，下の 422 ページの「スレッドプール」および 429 ページの「リアルタイム CORBA ミューテックス API」で説明されています。

リアルタイム ORB は，明示的に初期化する必要はなく，通常の CORBA::ORB_init 呼び出しの中で暗黙的に初期化されます。リアルタイム ORB 初期化引数は，非リアルタイム引数とともに，CORBA::ORB_init の呼び出しに渡されます。無効なリアルタイム初期化引数がある場合は，ORB_init の呼び出しが失敗し，システム例外が生成されます。

アプリケーションでリアルタイム ORB オペレーションを使用するには，RTCORBA::RTORB インスタンスへのリファレンスを持つ必要があります。このリファレンスは，ORB_init の呼び出し後にいつでも取得でき，オブジェクト ID"RTORB" をパラメータとして CORBA::ORB の resolve_initial_references オペレーションを呼び出すことで取得できます。resolve_initial_references は CORBA::Object_ptr 型のリファレンスを返すため，これを使用する前に RTCORBA::RTORB_ptr にナローイングする必要があります。

メモ　リアルタイム CORBA 拡張をサポートするために，VisiBroker for C++ ORB は特殊な「リアルタイム互換」モードで動作する必要があり，その動作とセマンティクスは通常の動作モードと異なります。"RTORB" リファレンスを取得することで自動的に ORB がこの特殊モードになるため，動作の不整合を防ぐために，アプリケーションでは，**できるだけ早く "RTORB" リファレンスを取得する**必要があります。

次のコード例は，RTCORBA::RTORB リファレンスを取得する方法を示します。この VisiBroker リリースに付属するリアルタイム CORBA の例 priority_model，threadpool，visthread，および rtmutex に同様のコードがあります。

```
#include "corba.h"
#include "rtcorba.h"
```

```
// 最初に ORB を初期化します
CORBA::ORB_ptr orb;

VISTRY
{
    orb = ORB_init(argc, argv);
}
VISCATCH(CORBA::Exception, e)
{
    cerr << "Exception initializing ORB" << endl << e << endl;
    // ここでエラーを処理します
}
VISEND_CATCH

// 次に RTORB リファレンスを取得します
CORBA::Object_var ref;

// ref が自動的に解放されるように _var を使用しています
VISTRY
{
    ref = orb->resolve_initial_references("RTORB");
}
VISCATCH
{
    cerr << "Exception obtaining RTORB reference" << endl << e << endl;
    // ここでエラーを処理します
}
VISEND_CATCH

// 最後に RTORB リファレンスをナローイングします
RTCORBA::RTORB_ptr rtorb;
VISTRY
{
    rtorb = RTCORBA::RTORB::_narrow(ref);
    // ref は不要になります。これは _var なので，自動的に解放されます
}
VISCATCH(CORBA::Exception, e)
{
    cerr << "Error narrowing RTORB reference" << endl << e << endl;
    // ここでエラーを処理します
}
VISEND_CATCH
```

リアルタイムオブジェクトアダプタ

リアルタイム CORBA では，オブジェクトアダプタは常にリアルタイムオブジェクトアダプタです。つまり，すべてのオブジェクトアダプタは優先順位を認識しており，リアルタイム CORBA によって定義された規則にしたがって CORBA 呼び出しを処理します。ノード上のすべてのオブジェクトアダプタがリアルタイムであることが必要です。CORBA アプリケーション内の一部のオブジェクトアダプタが非リアルタイムだとすると，そのオペレーションがリアルタイムオブジェクトアダプタの動作に干渉します。これは，すべてのオブジェクトアダプタに関連付けられたスレッドを OS によってまとめてスケジュールする必要があるためです。

すべてのオブジェクトアダプタがリアルタイムなので，その管理には，通常のポータブルオブジェクトアダプタ（POA）インターフェースが使用されます。

リアルタイムオブジェクトアダプタは，`create_POA` の呼び出しによって通常の方法で作成されます。追加のリアルタイムプロパティの設定は，ポリシーリストパラメータに新しい

リアルタイムポリシーを渡すことで行われます。このような新しいポリシー（およびそれに関連する値）を指定する POA の作成の例を次に示します。

```
// リアルタイム CORBA 優先順位モデルポリシーを作成します
// （RTORB リファレンスは取得済みです）
RTCORBA::PriorityModelPolicy_ptr priority_model_policy =
    rtorb->create_priority_model_policy(
            RTCORBA::SERVER_DECLARED, 25 );

// この RT CORBA ポリシーを含むポリシーリストを作成します
// （非リアルタイムポリシーが必要な場合は，同じリストに入れます）
CORBA::PolicyList policies;
policies.length(1);
policies[0] = priority_model_policy;

// ポリシーリストを使用して，POA を作成します
// （特に指定がなければ，POA をルート POA の POA マネージャに関連付けます）
// （ルート POA リファレンスは取得済みです）
PortableServer::POAManager_var poa_manager =
   rootPOA->the_POAManager();
VISTRY
{
    poa = rootPOA->create_POA("myPOA", poa_manager, policies);
}
VISCATCH(CORBA::Exception, e)
{
    // ここで，例外を処理します
}
VISEND_CATCH
```

POA の作成時に設定可能なリアルタイムポリシーは，POA がサポートし，スレッドプールに関連付けられる優先順位モデルに関係するポリシーです。これらのプロパティの設定は，422 ページの「スレッドプール」および427 ページの「リアルタイム CORBA 優先順位モデル」で説明されています。

これらのリアルタイムプロパティの一部が POA の作成時にアプリケーションによって設定されていない場合，ORB は，それらのプロパティをデフォルト値で初期化します。優先順位モデルのデフォルトの動作では，POA がサーバー宣言優先順位モデルをサポートします。また，スレッドプールのデフォルトの動作では，POA が汎用スレッドプールに関連付けられます。これらのデフォルトは，427 ページの「リアルタイム CORBA 優先順位モデル」および422 ページの「スレッドプール」で説明されています。

リアルタイム CORBA 優先順位

リアルタイム CORBA は，「*リアルタイム CORBA 優先順位*」と呼ばれる普遍的でプラットフォームに依存しない優先順位スキームを定義します。このスキームを使用して，リアルタイム CORBA アプリケーションは，さまざまな OS で実行されているノード間で首尾一貫した方法で，優先順位付きの CORBA 呼び出しを実行できます。既存のシステムのすべてのノードが同じ OS で実行されている場合でも，このスキームを使用すると，システム内の優先順位の設定に役立ち，アプリケーションの可搬性が向上し，将来混成 OS 環境に拡張する作業が簡単になります。

一貫性と可搬性を維持するため，アプリケーションの CORBA 部分の優先順位を表現する際に，**リアルタイム CORBA アプリケーションでは，リアルタイム CORBA 優先順位を使用する必要があります**。これは，システム内のすべてのノードが同じ OS を使用し，したがって同じ優先順位スキームを使用している場合でも同様です。

リアルタイム CORBA 優先順位を表すには，次の RTCORBA::Priority 型が使用されます。

```
module RTCORBA {
    typedef short Priority;
    const Priority minPriority = 0;
    const Priority maxPriority = 32767;
};
```

Java 言語マッピングに対応するために，signed short が使用されます。ただし，0（minPriority）から 32767（maxPriority）までの値だけが有効です。

メモ リアルタイム CORBA 仕様にしたがい，RTCORBA::Priority の値が大きいほど，優先順位が高いと定義されます。

実際のアプリケーションで，有効な RTCORBA::Priority 値の範囲全体（0 ～ 32767）を使用する必要はありません。アプリケーションの必要性に合わせて，より狭い範囲を適格な範囲として定義できます。それには，優先順位マッピングを制御します。優先順位マッピングについては，次のセクションで説明します。

デフォルトでは，VisiBroker for C++ は，0 から 31 までの RTCORBA::Priority 値だけを使用する優先順位マッピングをインストールします。これは，POSIX のスレッド優先順位範囲です。詳細は，次のセクションを参照してください。

優先順位マッピング

特定のリアルタイム OS には特定の優先順位スキーム（使用される優先順位値の範囲および方向）があります。たとえば，一部の OS で使用されている Pthreads の優先順位スキームは，POSIX 準拠の 0 から 31 までの優先順位を持ちます。リアルタイム CORBA では，これは「*ネイティブスレッド優先順位スキーム*」と呼ばれ，この優先順位値は「*ネイティブ優先順位値*」と呼ばれます。

リアルタイム CORBA アプリケーションは RTCORBA::Priority 値に基づいて優先順位を記述し，OS はネイティブ優先順位値に基づいて動作するため，この 2 つの優先順位スキームのマッピングを定義する必要があります。このマッピングは，特定の RTCORBA::Priority 値に対応するネイティブ優先順位を取得したり，特定のネイティブ優先順位に対応する RTCORBA::Priority 値を取得するために，必要に応じて ORB によって使用されます。これは，たとえば，スレッドプールに特定の RTCORBA::Priority で作成されたスレッドを格納するようにアプリケーションで指定され，実際にスレッドを作成するときに，どのネイティブ優先順位を使用するように OS に指示するかを ORB が決定する必要がある場合に行われます。

優先順位マッピングは，アプリケーションから直接使用することもできますが，これは特別な状況に限定してください。詳細は，421 ページの「VisiBroker アプリケーションコードでのネイティブ優先順位の使用」で説明します。

ORB はデフォルトの優先順位マッピングを内蔵していますが，これは，リアルタイム CORBA 機能を試用するには十分な機能であり，また多くのリアルタイムアプリケーションに対しても十分な機能です (POSIX の優先順位スキームに基づいているため)。したがって，VisiBroker for C++ のリアルタイム機能について初めて学習する場合は，このセクションの残りの部分を飛ばし，その他のリアルタイム CORBA 機能（422 ページの「スレッドプール」以降）から学習することをお勧めします。その後，このセクションに戻り，優先順位マッピングの詳細やデフォルト以外のマッピングをインストールする理由を理解してください。

優先順位マッピングの種類

優先順位マッピングをサポートするために，RTCORBA::NativePriority 型と RTCORBA::PriorityMapping 型が定義されています。

```
module RTCORBA {
    typedef short NativePriority;
    native PriorityMapping
};
```

RTCORBA::NativePriority の値は，-32768 から +32767 までの整数である必要があります。ただし，OS によっては，この範囲の一部が有効な範囲であることがあります。

RTCORBA::PriorityMapping 型は，IDL ネイティブインターフェースとして定義されています。つまり，このインターフェースは，IDL で定義され，特定の CORBA 言語マッピングの規則にしたがって自動的に各言語にマップされるのではなく，各インプリメンテーション言語で直接定義されます。これは，効率性が理由です。

RTCORBA::PriorityMapping インターフェースの C++ マッピングを次に示します。

```
class PriorityMapping {
    public:
        virtual CORBA::Boolean to_native(
                RTCORBA::Priority corba_priority,
                RTCORBA::NativePriority &native_priority );

        virtual CORBA::Boolean to_CORBA(
                RTCORBA::NativePriority native_priority,
                RTCORBA::Priority &corba_priority );

        virtual RTCORBA::Priority max_priority();

        PriorityMapping();
        virtual ~PriorityMapping() {}
        static RTCORBA::PriorityMapping * instance();
};
```

特定の優先順位マッピングの動作を定義するメソッドは，to_native，to_CORBA，および max_priority です。これらのメソッドの目的は次のとおりです。

- to_native
  このメソッドは，corba_priority パラメータから RTCORBA::Priority 値を受け取り，それを RTCORBA::NativePriority 値にマップするか，マップに失敗します。値がマップされた場合は，native_priority パラメータ（C++ リファレンスパラメータ）によって参照される場所に結果のネイティブ優先順位値が格納され，マッピングが成功したことを示す true 値が返されます。値がマップされなかった場合，native_priority パラメータの内容は変更されず，マッピングオペレーションが失敗したことを示す false 値が返されます。
- to_CORBA
  to_native の逆で，to_CORBA メソッドは，native_priority パラメータから RTCORBA::NativePriority 値を受け取り，それを RTCORBA::Priority 値にマップするか，マップに失敗します。値がマップされた場合は，corba_priority パラメータ（C++ リファレンスパラメータ）によって参照される場所に結果の RTCORBA::Priority 値が格納され，マッピングが成功したことを示す true 値が返されます。値がマップされなかった場合，corba_priority パラメータの内容は変更されず，マッピングオペレーションが失敗したことを示す false 値が返されます。
- max_priority
  このメソッドは，このマッピングで有効な最大の RTCORBA::Priority 値を返します。さまざまな入力値に対する to_native や to_CORBA の動作を調べる方法では最大値を効率的に判断できないため，ORB に明示的に最大値を示す必要があります。

これらのメソッドのインプリメンテーションは，次に説明されている一定の規則にしたがう必要があります。

優先順位マッピングの規則

インストールする優先順位マッピング（デフォルトの優先順位マッピングを含む）は，次の規則を満たす必要があります。

- `to_native` および `to_CORBA` メソッドは，-32768 から +32767 までのすべての入力パラメータ値を処理できる必要があります。
- `to_native` は，0 から 32767 までの範囲外の値のマッピングには必ず失敗する必要があります。また，その範囲内の値のマッピングにも失敗することができます。たとえば，デフォルトの優先順位マッピングは, 0 から 31 までの範囲にない値のマッピングに失敗します。
- `to_CORBA` は，ネイティブ優先順位スキームの範囲外の値のマッピングには必ず失敗する必要があります。また，その範囲内の値のマッピングにも失敗することができます。デフォルトの優先順位マッピングは，0 から 31 までの値のマッピングを選択します。
- 下位の `RTCORBA::Priority` 値は，常に重要度が下位のネイティブ優先順位にマップし，上位の値は，常に上位のネイティブ優先順位にマップする必要があります。ただし，Pthreads ベースの OS の場合は，数値的に下位の RTCORBA::Priority が数値的に上位のネイティブ優先順位にマップされます。*これは，大多数のリアルタイム OS で使用されている規則にしたがっています。この規則により，他のリアルタイム OS で開発されたリアルタイム CORBA アプリケーションとの一貫性が維持されます。そうしないと，将来の移植や他のリアルタイムアプリケーションとの相互運用が非常に複雑になります。*
- `RTCORBA::Priority` の 0 は常にマップする必要があり，マップされるネイティブ優先順位値の範囲内で最も重要度が下位のネイティブ優先順位値にマップする必要があります。
- `max_priority` は，マップされている最高の `RTCORBA::Priority` 値（ネイティブ優先順位値が返される最高値）を返す必要があります。

次の規則は必須ではありませんが，他の方法を取る特別な理由がない場合は，通常，これにしたがいます。

- `to_native` および `to_CORBA` は，通常，同じ入力値で呼び出されたときに毎回同じ値を返します（またはマッピングに失敗します）。
- `to_native` および `to_CORBA` は，通常，互いに逆のマッピングを行います。
- マップされる `RTCORBA::Priority` およびネイティブ優先順位の値は，通常，1 つの連続する値の範囲です。

デフォルトの優先順位マッピング

VisiBroker for C++ には，デフォルトの優先順位マッピングが用意されてます。アプリケーション開発者が別の優先順位を記述し，それを 420 ページの「デフォルト以外の優先順位マッピングの使用」で説明されている手順にしたがってインストールしない限り，この優先順位マッピングが使用されます。

メモ 特定の VisiBroker アプリケーションに一度にインストールできる優先順位マッピングは 1 つだけです。ある優先順位マッピングをインストールすると，以前にインストールされていた優先順位マッピング（通常はデフォルトの優先順位マッピング）はアンインストールされます。

デフォルトの優先順位マッピングには次の特性があります。

- 有効な `RTCORBA::Priority` の範囲は 0 から 31 までです。これは，POSIX スレッドモデルにしたがいます。この範囲外のすべての優先順位は無効です。それらを使用しようとした場合は，例外が生成されます。
- 有効な `RTCORBA::Priority` の値は, ネイティブ OS の 32 段階の優先順位のサブ範囲（「ネイティブ優先順位範囲」）に 1 対 1 でマップされます。
- `RTCORBA::Priority` 値の 0 が，サブ範囲内の重要度が最下位のネイティブ優先順位に対応し，`RTCORBA::Priority` 値の 31 が，サブ範囲内の重要度が最上位のネイティブ優先順位に対応するように，有効な `RTCORBA::Priority` 値がネイティブ優先順位範囲にマップされます。次の表に, サポートされている OS での `RTCORBA::Priority` のデフォルトマッピングを示します。

オペレーティングシステム	RTCORBA::Priority 範囲	ネイティブ優先順位範囲	Solaris	0 - 31	10 - 41
HPUX	0 - 31	0 - 31	Linux	0 - 31	30 - 61

デフォルト優先順位マッピングは ORB 内で定義されているため，そのソースコードは，この VisiBroker リリースに含まれていません。そのかわり，次にこのマッピングのソースコードを記載して，マッピングの動作を正確に示します。

```
// VisiBroker for C++ のデフォルトの 'to_native' 優先順位マッピング
CORBA::Boolean
VISDefaultPriorityMapping::to_native( RTCORBA::Priority corba_priority,
        RTCORBA::NativePriority &native_priority )
{
    if ((corba_priority < 0) || (corba_priority > 31))
    {
        return (CORBA::Boolean) 0;
    }
    else
    {
#if defined(SOLARIS)
        native_priority = 10 + corba_priority;  // 0 -> 10, 31 -> 41
#elif defined(HPUX_11)
        native_priority = corba_priority;       // 0 ->  0, 31 -> 31
#elif defined(__linux)
        native_priority = 30 + corba_priority;  // 0 -> 30, 31 -> 61
#else
#       エラー：サポートされている OS が検出されませんでした
#endif
        return (CORBA::Boolean) 1;
    }
}

// VisiBroker for C++ のデフォルトの 'to_corba' 優先順位マッピング
CORBA::Boolean
VISDefaultPriorityMapping::to_CORBA( RTCORBA::NativePriority native_priority,
        RTCORBA::Priority &corba_priority )
{
#if defined(SOLARIS)
    if ((native_priority < 10) || (native_priority > 41))
#elif defined(HPUX_11)
    if ((native_priority <  0) || (native_priority > 31))
#elif defined(__linux)
    if ((native_priority < 30) || (native_priority > 61))
#else
#   エラー：サポートされている OS が検出されませんでした
#endif
    {
```

```
        return (CORBA::Boolean) 0;
    }
    else
    {
#if defined(SOLARIS)
        corba_priority = native_priority - 10;    // 10 -> 0, 41 -> 31
#elif defined(HPUX_11)
        corba_priority = native_priority;        //  0 ->  0, 31 -> 31
#elif defined(__linux)
        corba_priority = native_priority - 30;    // 30 -> 0, 61 -> 31
#else
#       エラー：サポートされている OS が検出されませんでした
#endif
        return (CORBA::Boolean) 1;
    }
}

// デフォルトの最大値メソッド：デフォルトの優先順位マッピングが
// サポートする最大の RTCORBA::Priority を返します
RTCORBA::Priority VISDefaultPriorityMapping::max_priority()
{
    return 31;
}
```

デフォルト以外の優先順位マッピングの使用

メモ 特定のシステムに一度にインストールできる優先順位マッピングは 1 つだけです。ある優先順位マッピングをインストールすると，以前にインストールされていた優先順位マッピング（通常はデフォルトの優先順位マッピング）はアンインストールされます。

アプリケーションでは，システム内の一部または全部のノードでデフォルト以外の優先順位マッピングを使用する必要がある場合があります。たとえば，次の理由によります。

- マップされているネイティブ優先順位値の範囲を全体的なネイティブ優先順位スキーム内で上位または下位にシフトするため。たとえば，デフォルトの優先順位マッピングのネイティブ優先順位の範囲が 10 から 41 である場合に，これを 50 から 81 の範囲（重要度の高位に）や 200 から 231 の範囲（さらに重要度の高位に）に置き換えます。
- 有効な（マップされている）値の範囲にある `RTCORBA::Priority` 値の数を増減するため。たとえば，0 から 8 までの `RTCORBA::Priority` 値だけをマップしたり，0 から 128 までの値をマップします。
- 有効な（マップされている）値の範囲にあるネイティブ優先順位値の数を増減するため。たとえば，128 から 256 までのネイティブ優先順位値をマップします。

マッピングが 1 対 1 になるかどうかが，マッピング内の有効な `RTCORBA::Priority` 値とネイティブ優先順位値の範囲の関係に依存することに注意してください。マッピングが 1 対 1 である必要はありませんが，その方が便利です。デフォルトの優先順位マッピングは 1 対 1 のマッピングです。

メモ インストールする優先順位マッピングは，`RTCORBA::Priority` の 0 が最下位の重要度になるという，デフォルト優先順位マッピングで使用されている規約にしたがう必要があります。つまり，`RTCORBA::Priority` の 0 を（マップされるサブ範囲の）数値的に最小のネイティブ優先順位値にマップしてください。こうすることで，複数の OS にまたがって開発されたリアルタイム CORBA アプリケーションで一貫性が維持されます。そうしないと，将来の移植や他のリアルタイムアプリケーションとの相互運用が非常に複雑になります。

新しい優先順位マッピングをインストールするには，`RTCORBA::PriorityMapping` クラスを継承する新しいクラスを定義し，その静的インスタンスをアプリケーション内に 1 つ作成します。（静的コンストラクタの実行中に）この静的インスタンスが初期化されると，ベー

ス RTCORBA::PriorityMapping クラスのコンストラクタが ORB に新しいマッピングを登録します。

新しい優先順位マッピングを記述してインストールする例については，VisiBroker のインストールディレクトリに含まれている threadpool の例の mapping.h および mapping.C ファイルを参照してください。mapping.C では，グローバルスコープで新しいクラスの単一のインスタンスが作成されていることに注目してください。生成される mapping.o が VisiBroker アプリケーションとともにビルドされ，アプリケーションの実行時に静的コンストラクタの初期化が実行されると，このインスタンスが初期化されてマッピングがインストールされます。

VisiBroker アプリケーションコードでのネイティブ優先順位の使用

アプリケーションは，さまざまな CORBA アプリケーション部分の優先順位（およびアプリケーションの各部分の CORBA 呼び出しの優先順位）を判断するためにリアルタイム CORBA 優先順位を使用する必要がありますが，ネイティブ優先順位を判断する必要がある場合もあります。たとえば，CORBA アプリケーションの外部のサブシステム（ネイティブ優先順位スキームしか認識できない）を設定したり，OS 呼び出し（ネイティブ優先順位値をパラメータとして受け取る）を直接使用する場合です。このような場合は，アプリケーション内でリアルタイム CORBA とネイティブ優先順位を変換する必要があります。そのために，VisiBroker for C++ には，RTCORBA::PriorityMapping クラスに instance 静的メソッドが用意されています。このメソッドは，現在インストールされている優先順位マッピングへのポインタを返します。

このメソッドを使用すると，マッピングの内部的なインプリメンテーションの詳細に関係なく，アプリケーションコードからどのような優先順位マッピングメソッドを呼び出しても，現在インストールされているマッピングに対して実行されることが保証されます。これにより，インストールされているマッピングが変更されても，コードは引き続き機能します。次の例では，インストールされている優先順位マッピングをアプリケーションコードから使用しています。

```
RTCORBA::Priority corba_priority;

// 優先順位マッピングメソッドは，例外を生成せず，
// ブール値フラグを返します
if
( !RTCORBA::PriorityMapping::instance()->to_CORBA(
                        100,corba_priority) )
{
    // ネイティブ優先順位から RT CORBA 優先順位へのマッピングの失敗を処理します
}
// corba_priority 値を使用します ...
```

スレッドプール

VisiBroker for C++ は，ORB のサーバー側での実行スレッドを管理するためにスレッドプールを使用します。スレッドプールは次の機能を提供します。

- スレッドの事前割り当て。
  一定数の同時呼び出しを処理するために十分なスレッドリソースをアプリケーションプログラマが確保できるようにすることで，リアルタイムシステムの動作を保証します。また，呼び出しごとにスレッドを破棄したり再生成しないことで，遅延を減らし，予測可能性を高めます。
- スレッドの分割。
  複数のオブジェクトアダプタに複数のスレッドプールを関連付けることで，アプリケーションシステムの一部を（優先順位が低い）別の部分から分離して，それぞれ独立にスレッドを使用させることができます。これも，システム全体のリアルタイム動作の実現に役立ちます。
- スレッドの使用範囲の指定。
  スレッドプールを使用すると，1 つまたは複数の POA が使用できるスレッド数の上限を設定できます。使用できるスレッドの総数が制約されているシステムでは，この機能とスレッドプールの分割を組み合わせて，システムの重要部分でスレッドが枯渇することを防止できます。

スレッドプール API

スレッドプールを管理するには，`RTCORBA::RTORB` インターフェースの次のオペレーションを使用します。

```
module RTCORBA {
    typedef unsigned long ThreadpoolId;

    // 局所性制約付きオブジェクト
    interface RTORB {
        exception InvalidThreadpool {};

        ThreadpoolId create_threadpool(
                in unsigned long stacksize,
                in unsigned long static_threads,
                in unsigned long dynamic_threads,
                in Priority default_priority,
                in boolean allow_request_buffering,
                in unsigned long max_buffered_requests,
                in unsigned long max_request_buffer_size );

        void destroy_threadpool( in ThreadpoolId threadpool )
            raises (InvalidThreadpool);

        void threadpool_idle_time(
                in ThreadpoolId threadpool,
                in unsigned long seconds )
            raises (InvalidThreadpool);
    };
};
```

これらのオペレーションについては，以下のセクションで説明します。スレッドプールを作成して POA に関連付ける例は，VisiBroker インストールディレクトリに含まれている `threadpool` の例にあります。

スレッドプールの作成および設定

スレッドプールを作成するには，リアルタイム ORB の create_threadpool オペレーションを呼び出します。create_threadpool の引数には次の意味があります。

- stacksize
  このスレッドプールで作成される各スレッドのスタックサイズ（バイト単位）。
- static_threads
  スレッドプールの作成時に作成されてプールに割り当てられるスレッドの数。これらのスレッドは，スレッドプール自体が破棄されるまで破棄されません。これらは，CORBA 呼び出しを実行するために使用された後で，スレッドプールに戻され，次の呼び出しの実行を待機します。
- dynamic_threads
  すべての静的スレッドが使用されているときに受け取った CORBA 呼び出しを実行するために動的に作成できるスレッドの数。この値は 0 にできます。その場合，スレッドプールの作成後には動的にスレッドを生成できなくなります。つまり，同時に実行できる呼び出しの数は，静的スレッドの数に制限されます。
- default_priority
  アイドル状態のスレッドがプール内で CORBA 呼び出しの実行を待機する間の RTCORBA::Priority。呼び出しが実行される優先順位は，使用中のリアルタイム CORBA 優先順位モデルに依存します。詳細は，427 ページの「リアルタイム CORBA 優先順位モデル」を参照してください。このパラメータは，呼び出しを処理していないときのスレッドの優先順位を決定します。
- allow_request_buffering，max_buffered_requests，max_request_buffer_size
  これらの引数は，リアルタイム CORBA 仕様に含まれる要求バッファリング機能をサポートします。この機能を使用して，スレッドプールの静的および動的スレッドの制限に達したときに，呼び出し要求をキューに入れることができます。この機能は，現在 VisiBroker for C++ ではサポートされておらず，これらの引数の値は無視されます。

dynamic_threads が 0 より大きく，スレッドを動的に作成できる場合，CORBA 呼び出しを処理するために作成されたスレッドは，その呼び出しの実行が完了した後もすぐには破棄されません。それらのスレッドは，静的スレッドと同様にスレッドプールに戻されます。ただし，スレッドプール内でアイドル状態のままの動的スレッドは，適宜行われるガベージコレクションによって最終的に破棄されます。

動的に作成されたスレッドを破棄するまでアイドル状態のままスレッドプールに格納しておく時間は，RTCORBA::RTORB の threadpool_idle_time オペレーションを使用して設定できます。このオペレーションでアイドル時間を設定しない場合は，デフォルトの 300 秒になります。

create_threadpool が成功した場合は，新しいスレッドプールの識別子が返されます。この識別子は RTCORBA::ThreadpoolId 型（unsigned long）で，これ以降，そのスレッドプールを参照するために使用されます。

メモ このリアルタイムのコンテキストでの dynamic_threads 値のセマンティクスは，「*ディスパッチャスレッドプール*」の threadMax 値のセマンティクスとは異なります。125 ページの「スレッドプールディスパッチポリシー」を参照してください。dynamic_threads が 0 の場合は，必要であっても，RTCORBA スレッドプールに追加スレッドは作成されません。対照的に，ディスパッチャスレッドプールの threadMax 値が 0 の場合，VisiBroker ORB は，必要に応じて自由に追加スレッドを作成します。

オブジェクトアダプタとスレッドプールの関連付け

VisiBroker for C++ を使用して作成された POA は，それぞれ 1 つのスレッドプールに関連付けられます。一方，各スレッドプールは，任意の数の POA に関連付けることができます。アプリケーション設計者は，複数の POA が同じスレッドプールを使用したり異な

るスレッドプールを使用するように設定することで，CORBA オブジェクトのセットごとに使用されるスレッドを制御できます。

POA をどのスレッドプールに関連付けるかは，目的のスレッドプールの RTCORBA::ThreadpoolId を RTCORBA::ThreadpoolPolicy ポリシーの値として create_POA オペレーションに渡すことで決定されます。次の例では，POA の初期化時に POA をスレッドプールに関連付けています。

```
// RTORB リファレンスを取得します
CORBA::Object_var objref =
    orb->resolve_initial_references("RTORB");
RTCORBA::RTORB_var rtorb = RTCORBA::RTORB::_narrow(objref);

// スレッドプールを作成します
RTCORBA::ThreadpoolId tpool_id =
    rtorb->create_threadpool(
            30000, // スタックサイズ
            5, // 静的スレッドの数
            0, // 動的スレッドの数
            20, // デフォルトの RT CORBA 優先順位
            0, 0, 0);

// POA の初期化で使用するスレッドプールポリシーオブジェクトを作成します
RTCORBA::ThreadpoolPolicy_ptr tpool_policy =
    rtorb->create_threadpool_policy( tpool_id );

// POA の初期化のためのポリシーリストを作成します
// （非リアルタイムポリシーが必要な場合は，同じリストに入れます）
CORBA::PolicyList policies;
policies.length(1);
policies[0] = tpool_policy;

// ポリシーリストを使用して，POA を作成します
// （特に指定がなければ，POA をルート POA の POA マネージャに関連付けます）
// （ルート POA リファレンスは取得済みです）
PortableServer::POAManager_var poa_manager =
        rootPOA->the_POAManager();
VISTRY
{
    poa = rootPOA->create_POA("myPOA", poa_manager, policies);
}
VISCATCH(CORBA::Exception, e)
{
    // ここで，例外を処理します
}
VISEND_CATCH
```

リアルタイム CORBA 設定のいずれかの部分が無効な場合，create_POA は失敗します。たとえば，ThreadpoolId が既存のスレッドプールに一致しない場合は，CORBA::BAD_PARAM システム例外が生成されます。

汎用スレッドプール

前のセクションで説明したように，POA の作成時にスレッドプールを指定しない場合，新しく作成された POA は，「*汎用スレッドプール*」と呼ばれる特殊なスレッドプールに関連付けられます。

汎用スレッドプールは，RTCORBA::RTORB の create_threadpool オペレーションを呼び出して作成する必要はありません。汎用スレッドプールは，最初に必要になったときに ORB が自動的に作成します。

汎用スレッドプールは，次の設定で作成されます。

- stacksize = 30000
- static_threads = 0
- dynamic_threads = 1000
- default_priority = 0
- max_thread_idle_time = 300

この設定がアプリケーションに適していない場合は，汎用スレッドプールを使用しないで，各 POA の作成時に明示的に POA を適切に設定されたスレッドプールに関連付ける必要があります。

スレッドプールの破棄

スレッドプールは，その ThreadpoolId を引数として渡して RTCORBA::RTORB の destroy_threadpool オペレーションを呼び出すことで破棄できます。

```
// RTORB リファレンスとスレッドプール ID はすでに取得されています

// RT ORB リファレンスを取得します
CORBA::Object_var objref =
        orb->resolve_initial_references("RTORB");
RTCORBA::RTORB_var rtorb = RTCORBA::RTORB::_narrow(objref);

VISTRY
{
    rtorb->destroy_threadpool( pool_id );
}
VISCATCH(CORBA::Exception, e)
{
    // ここでエラーを処理します
}
VISEND_CATCH
```

destroy_threadpool オペレーションが成功するには，特定のスレッドプールに関連付けられていた POA（作成時に，使用するスレッドプールとしてこのスレッドプールが指定されていた POA）が事前にすべて破棄されている必要があります。

スレッドプールに関連付けられている POA がまだ存在する場合は，呼び出しが失敗し，システム例外が生成されます。

リアルタイム CORBA Current

リアルタイム CORBA では，スレッドの CORBA 優先順位にアクセスするためにリアルタイム CORBA Current インターフェースが定義されています。次のサンプルは RTCORBA::Current インターフェースを示します。

```
module RTCORBA {
    interface Current : CORBA::Current {
        attribute Priority base_priority;
    };
};
```

リアルタイム CORBA 優先順位は，RTCORBA::Current オブジェクトの base_priority 属性を設定することで，現在のスレッドに関連付けることができます。これには，次の 2 つの効果があります。

- 現在のスレッドのネイティブ優先順位は，属性設定オペレーションのパラメータとして指定されたリアルタイム CORBA 優先順位値からマッププッピングされた値にただちに

設定されます。このため，この属性を設定することには，CORBA アプリケーションスレッドの優先順位を制御する効果があります。

- リアルタイム CORBA 優先順位値が格納され，そのスレッドから行われるすべての CORBA 呼び出しで使用されます。この値が関係するのは，「クライアント伝搬」優先順位モデルをサポートするように設定された POA から作成された CORBA オブジェクトに対して呼び出しを行う場合だけです。詳細は，427 ページの「リアルタイム CORBA 優先順位モデル」を参照してください。

この優先順位値は，新しい値が設定されるまで（上の両方の目的について）有効です。

対応する属性取得オペレーションを使用して，現在の値を読み取ることもできます。

有効な 0 から 32767 の範囲外の優先順位を設定しようとした場合は，属性設定オペレーションから CORBA::BAD_PARAM システム例外が生成されます。0 から 32767 の範囲内であっても，現在インストールされている優先順位マッピングがサポートしている範囲外の優先順位を設定しようとした場合は，CORBA::DATA_CONVERSION 例外が生成されます。

リアルタイム CORBA 優先順位値がまだ設定されていないスレッドから優先順位値を取得しようとした場合は，CORBA::INITIALIZE システム例外が生成されます。現在のスレッドのネイティブ優先順位は，リアルタイム CORBA 優先順位にマップされずに返されます。

RTCORBA::Current オブジェクトを使用するには，そのリファレンスを取得する必要があります。それには，次の例のように，CORBA::ORB の resolve_initial_references オペレーションを呼び出します。

```
// RTCORBA::Current リファレンスを取得します
CORBA::Object_var ref;

VISTRY
{
    ref = orb->resolve_initial_references("RTCurrent");
}
VISCATCH
{
    // ここでエラーを処理します
}
VISEND_CATCH

// RTCORBA::Current リファレンスをナローイングします
RTCORBA::Current_ptr rtcurrent;
VISTRY
{
    rtcurrent = RTCORBA::Current::_narrow(ref);
}
VISCATCH(CORBA::Exception, e)
{
    // ここでエラーを処理します
}
VISEND_CATCH
```

RTCORBA::Current リファレンスは一度だけ取得すれば十分です。同じ変数を異なるスレッドで使用でき，それぞれが固有であるかのように動作します（スレッド固有の優先順位値を設定および取得できます）。この動作は，ベース CORBA::Current オブジェクトから継承されます。

リアルタイム CORBA 優先順位モデル

リアルタイム CORBA は，システム全体の優先順位を調整するためのモデルを 2 つサポートしています。この 2 つのモデルの違いは，CORBA 呼び出しを実行する際の優先順位をどこから取得するかという問題に対する答えの違いです。

- クライアント伝搬優先順位モデル
  このモデルでは，RTCORBA::Current を使用してクライアント CORBA アプリケーションスレッドに関連付けられているリアルタイム CORBA 優先順位が，呼び出しのサーバー側の優先順位としても使用されます。呼び出しを実行するスレッド（スレッドプールから取り出される）は，呼び出しの実行前にクライアント側で設定されたリアルタイム CORBA 優先順位からマップされたネイティブ優先順位で実行されます。
- サーバー宣言優先順位モデル
  このモデルでは, クライアント CORBA アプリケーションスレッドに関連付けられたリアルタイム CORBA 優先順位が呼び出しのクライアント側の優先順位にのみ影響します。サーバー側で呼び出しが処理される優先順位は，CORBA オブジェクトとそれを作成した POA の設定によって決定されます。

どの優先順位モデルが使用されるかは，POA レベルで設定されるサーバー側の問題です。同じ POA から作成されたすべての CORBA オブジェクトでは，POA で設定された優先順位モデルにしたがって呼び出しが処理されます。

優先順位モデルは，create_POA のパラメータとして渡されるポリシーリストに RTCORBA::PriorityModelPolicy インスタンスを入れることで，POA の初期化時に選択されます。このポリシーは，次のいずれかの値に設定されます。

- RTCORBA::CLIENT_PROPAGATED
  クライアント伝搬優先順位モデルを選択します。
- RTCORBA::SERVER_DECLARED
  サーバー宣言優先順位モデルを選択します。

どちらの場合も，ポリシーの一部として RTCORBA::Priority 値が指定されます。次のように，2 つのモデルは，この優先順位値を異なる方法で使用します。

- クライアント伝搬優先順位モデルの場合，この値は，呼び出しの前に優先順位を設定していなかったクライアントからの呼び出しを実行するための優先順位になります。これには，非リアルタイム ORB（他のベンダーの非リアルタイム ORB を含む）からのクライアントと，RTCORBA::Current を使用して優先順位値をまだ設定していないスレッドからの呼び出しが含まれます。
- サーバー宣言優先順位モデルの場合，この値は，オブジェクトレベルで異なる優先順位が設定されていない場合に呼び出しを実行するための優先順位になります。オブジェクトレベルで優先順位を設定する方法の詳細は，次のセクションを参照してください。

サーバー宣言優先順位モデルがデフォルトモデルです。どちらのモデルを使用するかを指定せずに POA を初期化した場合は，サーバー宣言優先順位モデルを使用するように設定されます。ただし，この場合は，動作にわずかな違いがあります。つまり，優先順位が指定されていないため, POA が関連付けられているスレッドプールのデフォルトの優先順位で呼び出しが実行されます。デフォルトの優先順位は，スレッドプールの設定可能なプロパティです。これは，プール内に格納されているアイドル状態のスレッドの優先順位です。詳細は，422 ページの「スレッドプール」を参照してください。

次のコードでは，POA の作成時に優先順位モデルポリシーを設定しています。この場合は，クライアント伝搬優先順位モデルを選択し，デフォルトの優先順位は 7 です（非リアルタイムクライアントからの呼び出しに対応）。

```
// リアルタイム CORBA 優先順位モデルポリシーを作成します

RTCORBA::PriorityModelPolicy_ptr priority_model_policy =
    rtorb->create_priority_model_policy(
            RTCORBA::CLIENT_PROPAGATED, 7 );
```

```
// この RT CORBA ポリシーを含むポリシーリストを作成します
// （非リアルタイムポリシーが必要な場合は，同じリストに入れます）
CORBA::PolicyList policies;
policies.length(1);
policies[0] = priority_model_policy;

// ポリシーリストを使用して，POA を作成します
// （特に指定がなければ，POA をルート POA の POA マネージャに関連付けます）
// （ルート POA リファレンスは取得済みです）
PortableServer::POAManager_var poa_manager =
    rootPOA->the_POAManager();
VISTRY
{
    poa = rootPOA->create_POA("myPOA", poa_manager, policies);
}
VISCATCH(CORBA::Exception, e)
{
    // ここで，例外を処理します
}
VISEND_CATCH
```

ほかに 2 つの異なる優先順位モデルを設定する例については，VisiBroker のインストールディレクトリに含まれている priority_model の例を参照してください。

オブジェクトレベルでの優先順位の設定

サーバー宣言優先順位モデルが選択された場合は，ORB のサーバー側で呼び出しを実行するための優先順位を決定する優先順位値が指定されます。この優先順位値は，その POA によって作成されるすべての CORBA オブジェクトに対する呼び出しを処理するときに使用されます。

ただし，アプリケーションによっては，この優先順位制御のスコープが粗すぎることがあります。これを解決するために，リアルタイム CORBA では，サーバー宣言モデルで呼び出しを実行するための優先順位をオブジェクトごとに上書きできます。

呼び出しを実行するための優先順位を特定のオブジェクトに対して上書きするには，activate_object_with_priority または activate_object_with_id_and_priority オペレーションを使用してそのオブジェクトをアクティブ化します。これらのオペレーションは，activate_object および activate_object_with_id と同様に機能しますが，追加パラメータとしてリアルタイム CORBA 優先順位値を受け取ります。

リアルタイム CORBA 仕様では，これらのメソッドが RTPortableServer モジュールで定義されています。ただし，VisiBroker for C++ では，これらのオペレーションが VisiBroker 拡張 POA インターフェース PortableServerExt::POA の一部として定義されています。このインターフェースには，静的 C++ メソッド PortableServerExt::POA::_narrow を使用して POA オブジェクトリファレンスをナローイングすることでアクセスできます。

オブジェクトごとに優先順位を設定する例については，VisiBroker に含まれている priority_model の例の model_srvr.C ファイルを参照してください。

リアルタイム CORBA ミューテックス API

VisiBroker for C++ は，次のリアルタイム CORBA ミューテックスインターフェースを実装しています。

```
#include "timebase.idl"
module RTCORBA {

    // 局所性制約付きインターフェース
    interface Mutex {
        void lock();
        void unlock();
        boolean try_lock(in TimeBase::TimeT max_wait);
        // max_wait = 0 の場合は，すぐに戻ります
    };

    interface RTORB {
        ...
        Mutex create_mutex();
        void destroy_mutex( in Mutex the_mutex );
        ...
    };
};
```

新しい RTCORBA::Mutex オブジェクトは，`RTCORBA::RTORB` の `create_mutex` オペレーションを使用して取得します。Mutex オブジェクトには，locked および unlocked の 2 つの状態があります。Mutex オブジェクトは，unlocked 状態で作成されます。Mutex オブジェクトが unlocked 状態の場合は，最初に `lock()` オペレーションを呼び出したスレッドがその Mutex オブジェクトを locked 状態に変更し，呼び出したスレッドにその Mutex オブジェクトの所有権が割り当てられます。

Mutex オブジェクトがまだ locked 状態である場合は，オーナースレッドが `unlock()` オペレーションを呼び出してロック解除するまで，後続のスレッドが `lock()` を呼び出してもブロックされます。

`try_lock()` オペレーションは `lock()` オペレーションと同様に機能しますが，`max_wait` の時間内にロックできなかった場合は false を返す点が異なります。`try_lock()` オペレーションは，`max_wait` の時間内にロックできた場合は true を返します。

内部 ORB スレッド優先順位の制御

VisiBroker for C++ では，ORB が内部使用の目的で作成するスレッドの優先順位をアプリケーションが制御できます。

次の内部 ORB スレッドがあります。

- *DSUser スレッド*
  ORB が VisiBroker スマートエージェント（`osagent`）との通信を最初に試みたときに，単一の DSUser スレッドが作成されます。これは，通常，`activate_object` または `_bind` メソッドが最初に呼び出されたときに行われます。このスレッドは，ORB とスマートエージェントの間のすべての通信を管理します。スレッド名は “VISDSUser” です。
- *リスナースレッド*
  リスナースレッドは，サーバーエンジンの初期化の一環として作成されます。これは，POA がまだ使用されていないサーバーエンジンを使用するたびに，POA の初期化中に行われます。これらのスレッドは，ネットワーク接続から受信される着信 CORBA 呼び出しを待機します。IIOP 通信のリスナースレッドは，`VISLis<N>` 形式のスレッド名を持ちます。ここで，`<N>` は，0 から始まるインデックス番号で，リスナーが作成された順序を示します。

- *ガベージコレクションスレッド*
  この単一のインスタンスは，スレッドプールが最初に作成されるときに作成されます。これは，アプリケーションが明示的にスレッドプールを作成したときと，アプリケーションがスレッドプールを指定せずに POA を作成したとき（この場合は，汎用スレッドプールが作成されて使用される）に行われます。ガベージコレクションスレッドは，VISGC<N> 形式のスレッド名を持ちます。ここで，<N> は，関連付けられたスレッドプール ID に対応します。

アプリケーションでこれらのスレッドの優先順位を設定しない場合は，インストールされている優先順位マッピング内で最高の RTCORBA::Priority で実行されます。これは，優先順位マッピングの max_priority メソッドから返される優先順位です。そのため，デフォルトの優先順位マッピングがインストールされている場合は，すべて RTCORBA::Priority 31 で実行されます。

これらの内部 ORB スレッドタイプの優先順位を設定するには，次の 2 つの方法があります。

- VisiBroker のプロパティを使用して，タイプごとに（場合によっては，さらにインスタンスごとに）設定する。
- ORB 内部スレッドに範囲制限を設定して，まとめて設定する。この場合は，上のすべてのタイプのスレッドが，指定された範囲の最大優先順位で実行されます。

個別内部 ORB スレッドの優先順位の設定

内部 ORB スレッドの優先順位は，特定の VisiBroker プロパティを指定することで，タイプごと（場合によっては，さらにインスタンスごと）に制御できます。

どの場合も，この優先順位値はリアルタイム CORBA 優先順位値として指定されます。この値は，インストールされている優先順位マッピング内の有効な優先順位である必要があります。

- vbroker.se.default.socket.listener.priority
  リスナースレッドが実行されるデフォルト優先順位を設定します。これはいつでも変更できます。サーバーエンジンの作成（POA の作成時に行われる）時の現在の値が，新しいリスナーが作成されるときに使用されます。これは，次のプロパティを使用して上書きできます。
- vbroker.se.<SE 名 >.scm.<SCM 名 >.listener.priority
  ここで，<SE 名 > はサーバーエンジンの名前，<SCM 名 > はサーバー接続マネージャの名前です。特定のサーバーエンジンで特定の SCM に関連付けられたリスナースレッドの優先順位を設定します。これは，サーバーエンジンの作成（サーバーエンジンを使用する最初の POA の作成時に行われる）前ならいつでも設定できます。
- vbroker.agent.threadPriority
  ORB の DSUser スレッドが実行される優先順位を設定します。ORB が VisiBroker スマートエージェントとの通信を最初に試みる（通常は，POA が作成されるとき，オブジェクトがアクティブ化されるとき，または _bind メソッドが呼び出されるとき）前に設定する必要があります。
- vbroker.garbageCollect.thread.priority
  すべてのガベージコレクションスレッドの優先順位を設定します。これはいつでも変更できます。スレッドプールの作成時の現在の値が使用されます。

メモ vbroker.agent.threadPriority プロパティの名前は，以前のバージョンの VisiBroker リアルタイム製品では vbroker.dsuser.thread.priority でした。VisiBroker スマートエージェントの他のプロパティ名と合わせるために，この名前に変更されました。ただし，古いプロパティ名もまだサポートされており，既存の配布アプリケーションで引き続き使用できます。

内部 ORB スレッドの優先順位範囲の制限

ORB_init に引数 -ORBRTPriorityRange <min>,<max> を渡すことで，内部 ORB スレッドに範囲制限を設定できます。

次の例に示すように，-ORBRTPriorityRange を 1 つの引数として指定し，コンマで区切った 2 つの値を別の引数として一緒に指定します。

```
// ORB_init の引数を用意します
int argc = 3;
char * argv[] = { "app_name", "-ORBRTPriorityRange", "10,17" };

// ORB を初期化します
CORBA::ORB_ptr = ORB_init(argc, argv);
```

この 2 つの値は，内部 ORB スレッドが実行を許可される最小の RTCORBA::Priority と最大の RTCORBA::Priority を設定します。この引数が指定された場合，デフォルトの VisiBroker 内部 ORB スレッドは，指定された最大の優先順位で実行されます。

範囲が無効な場合，ORB_init の呼び出しは失敗し，CORBA システム例外が生成されます。一方または両方の値が有効な RTCORBA::Priority 値でないか，min が max より大きいために範囲が無効である場合は，CORBA::BAD_PARAM 例外が生成されます。一方または両方の値が，インストールされている優先順位マッピングでサポートされている範囲にないために，範囲が無効である場合は，CORBA::DATA_CONVERSION 例外が生成されます。

第 34 章

CORBA 例外

ここでは，VisiBroker ORB によって生成される CORBA 例外に関する情報を提供し，それが生成される原因について説明します。

CORBA 例外の説明

次の表は，CORBA 例外のリストです。また，VisiBroker ORB がこれらの例外を生成する場合に考えられる原因について説明します。

例外	説明	考えられる原因
CORBA::BAD_CONTEXT	サーバーに無効なコンテキストが渡されました。	クライアントがオペレーションを呼び出したが，渡されたコンテキストに，そのオペレーションに必要なコンテキスト値が含まれていな場合は，オペレーションがこの例外を生成します。
CORBA::BAD_INV_ORDER	問題のあったオペレーションリクエストの前に，必要なオペレーションが呼び出されていません。	実際に要求を送信する前に，CORBA::Request::get_response() または CORBA::Request::poll_response() メソッドを呼び出そうとしました。リモートメソッド呼び出しのインプリメンテーションの外で exception::get_client_info() メソッドを呼び出そうとしました。この関数は，リモート呼び出しのインプリメンテーション内でのみ有効です。すでにシャットダウンされている VisiBroker ORB に対してオペレーションが呼び出されました。
CORBA::BAD_OPERATION	無効なオペレーションが実行されました。	サーバーは，そのインプリメンテーションのインターフェースに定義されていないオペレーションに対する要求を受け取ると，この例外を生成します。クライアントとサーバーが同じ IDL からコンパイルされているかどうかを確認してください。要求が戻り値を持たないように設定されている場合，CORBA::Request::return_value() メソッドはこの例外を生成します。DII 呼び出しで戻り値が必要な場合は，CORBA::Request::set_return_type() メソッドを呼び出して，戻り値の型を設定してください。

（続き）

例外	説明	考えられる原因
CORBA::BAD_PARAM	VisiBroker ORB に渡されたパラメータが無効です。	無効なインデックスへのアクセスが試みられると，シーケンスによって CORBA::BAD_PARAM が生成されます。シーケンスの要素を保存または取得する前に，length() メソッドを使ってシーケンスの長さを設定してください。 無効な Object_ptr が in 引数として渡される場合，VisiBroker ORB はこの例外を生成します。たとえば，nil リファレンスが通過する場合などです。 IDL から C++ 言語へのマッピングのため，初期化済みの C++ オブジェクトが必要な場合に，NULL ポインタを渡そうとしました。たとえば，メソッドから sequence を返す必要がある場合に，戻り値または out パラメータとして NULL を返そうとすると，この例外が生成されます。この場合，かわりに（長さが 0 の）新しい sequence を返す必要があります。C++ NULL 値として渡すことができない型には，Any，Context，struct，および sequence があります。 Any に nil オブジェクトリファレンスを挿入しようとしました。 列挙データ型の範囲外の値を送信しようとしました。 無効な kind 値で TypeCode を作成しようとしました。 DII と一方向メソッド呼び出しを使用して，OUT 引数が指定された可能性があります。IR オブジェクトのオペレーションに渡された引数が既存の設定と競合する場合に，インターフェースリポジトリがこの例外を生成します。詳細については，コンパイラエラーを参照してください。
CORBA::BAD_QOS	QoS（Quality of service）をサポートできません。	QoS セマンティクスが関連付けられている呼び出しパラメータで必要な QoS をオブジェクトがサポートできない場合に生成されます。
CORBA::BAD_TYPECODE	ORB が不正な形式のタイプコードを検知しました。	
CORBA::CODESET_INCOMPATIBLE	クライアントとサーバーのネイティブコードセット間に互換性がないため，それらのコードセットどうしの通信がエラーになります。	クライアントとサーバーで使用されているコードセットどうしに互換性がありません。たとえば，クライアントが ISO 8859-1 を使用し，サーバーが日本語コードセットを使用しています。
CORBA::COMM_FAILURE	クライアントから要求が送信された後で応答が返される前の処理の進行中に，通信が失われました。	既存の接続のもう一方の終端で障害が発生したため，接続が切断されました。 処理は開始されている可能性があります。 システム例外が原因で COMM_FAILURE が発生した場合は，COMM_FAILURE のマイナーコードにシステムエラー番号が設定されます。マイナーコードでシステム固有のエラー番号（たとえば，include/sys/errno.h または msdev¥include¥winerror.h ファイルに含まれる）をチェックします。
CORBA::DATA_CONVERSION	VisiBroker ORB が，マーシャリングされたデータからネイティブなデータ，またはその逆の変換を行うことができません。	Output.write_char() または Output.write_string で Unicode 文字のマーシャリングを試みましたが失敗しました。
CORBA::FREE_MEM	VisiBroker ORB が動的メモリの解放に失敗しました。	VisiBroker ORB が解放しようとしているメモリのセグメントがロックされています。ヒープが破損しています。
CORBA::IMP_LIMIT	VisiBroker ORB の実行時にインプリメンテーションの限度を超過しました。	VisiBroker ORB が 1 つのアドレス空間内に同時に保持できるリファレンスの最大数に達しました。パラメータのサイズが許容される最大値を超えました。実行できるクライアントとサーバーの最大数を超えました。
CORBA::INITIALIZE	必要な初期化が行われていません。	ORB_init() メソッドが呼び出されなかった可能性があります。すべてのクライアントは，VisiBroker ORB に関連する処理の実行前に，ORB_init() メソッドを呼び出す必要があります。この呼び出しは，通常，プログラムの起動直後にメインルーチンの先頭で行われます。
CORBA::INTERNAL	内部 VisiBroker ORB エラーが発生しました。	内部 VisiBroker ORB エラーが発生した可能性があります。たとえば，VisiBroker ORB の内部データ構造が損傷しています。

（続き）

例外	説明	考えられる原因
CORBA::INTF_REPOS	インターフェースリポジトリのインスタンスが見つかりませんでした。	get_interface() メソッドの呼び出し中に，オブジェクトインプリメンテーションがインターフェースリポジトリを見つけることができない場合は，この例外がクライアントに向けて生成されます。インターフェースリポジトリが動作中であり，要求されたオブジェクトのインターフェース定義がそのインターフェースリポジトリにロードされていることを確認してください。
CORBA::INV_FLAG	オペレーションに無効なフラグが渡されました。	動的起動インターフェース要求が，無効なフラグを使って作成されました。
CORBA::INV_IDENT	IDL 識別子の構文が無効です。	インターフェースリポジトリに渡した識別子の形式が正しくありません。動的起動インターフェースに不正なオペレーション名が使用されています。
CORBA::INV_OBJREF	無効なオブジェクトリファレンスが検出されました。	使用できるプロファイルがないオブジェクトリファレンスを取得した場合，VisiBroker ORB はこの例外を生成します。文字列化されたオブジェクトリファレンスが「IOR:」という文字で始まっていない場合は，ORB::string_to_object() メソッドがこの例外を生成します。
CORBA::INV_POLICY	無効なポリシーオーバーライドが検出されました。	この例外は，あらゆる呼び出しから生成されます。一部の呼び出しに適用されるポリシーオーバーライド間に互換性がないため，その呼び出しを実行できない場合に生成されます。
CORBA::INVALID_TRANSACTION	要求が無効なトランザクションコンテキストを保持しています。	この例外は，リソースの登録中にエラーが発生すると生成されます。
CORBA::MARSHAL	パラメータまたは結果のマーシャリングエラー。	ネットワークを介した要求や応答の構造が正しくありません。通常，このエラーは，クライアントまたはサーバー側に実行時のバグがあることを示します。たとえば，サーバーからメッセージが 1000 バイトあることを示す応答があったときに，実際のメッセージがこれより長いか短いと，VisiBroker ORB がこの例外を生成します。MARSHAL 例外は，DII や DSI を不正な方法で使用した場合にも生成されます。たとえば，実際に送信されたパラメータがオペレーションの IDL シグニチャと一致しない場合です。
CORBA::NO_IMPLEMENT	要求されたオブジェクトが見つかりませんでした。	呼び出されたオペレーションが存在していても（オペレーションの IDL 定義がある），オペレーションのインプリメンテーションが存在しないことを示します。たとえば，クライアントがバインドを開始したときにサーバーが存在しないか，実行されていない場合に，NO_IMPLEMENTATION が生成されます。
CORBA::NO_MEMORY	VisiBroker ORB ランタイムがメモリ不足になりました。	
CORBA::NO_PERMISSION	呼び出し元が呼び出しを完了するための十分な権限を持っていません。	Object::get_implementation() メソッドや BOA::dispose() メソッドは，クライアント側で呼び出されると，この例外を生成します。これらのメソッドは，オブジェクトインプリメンテーションをアクティブ化したサーバー内で呼び出される場合にだけ有効です。 トランザクションオリジネータ以外のオブジェクトが Current::commit() または Current::rollback() を実行しています。
CORBA::NO_RESOURCES	必要なリソースを取得できませんでした。	新しいスレッドを作成できない場合に，この例外が生成されます。サーバーは，リモートクライアントが接続を確立しようとしたとき，ソケットを作成できないと，この例外を生成します。たとえば，サーバーがファイルデスクリプタを使い切っている場合です。マイナーコードには，失敗した ::socket() または ::accept() の呼び出しの後に取得されたシステムエラー番号が記されます。同様にクライアントも，ファイルデスクリプタを使い切ったため，::connect() 呼び出しに失敗すると，この例外を生成します。メモリ不足の場合にも，この例外が生成される可能性があります。

（続き）

例外	説明	考えられる原因
CORBA::NO_RESPONSE	クライアントが遅延同期呼び出しの結果を取得しようとしていますが，その要求への応答がまだ使用できません。	BindOptions を使ってタイムアウトを設定した場合，指定した時間内に送信および受信呼び出しが行われないと，この例外が生成されます。
CORBA::OBJ_ADAPTER	整合性のない管理操作が行われました。	サーバーが，すでに使用されている名前またはリポジトリに認識されない名前を使用して，自分自身をインプリメンテーションリポジトリに登録しようとしました。アプリケーションのサーバントマネージャに問題があり，POA が OBJ_ADAPTER エラーを生成しました。
CORBA::OBJECT_NOT_EXIST	要求されたオブジェクトが存在しません。	サーバーに存在しないインプリメンテーションのオペレーションを実行しようとすると，サーバーがこの例外を生成します。これは，クライアントがアクティブでないインプリメンテーションのオペレーションを呼び出そうとすると受け取る例外です。たとえば，オブジェクトへのバインドが失敗したり，自動のリバインドが失敗した場合に，OBJECT_NOT_EXIST が生成されます。
CORBA::PERSIST_STORE	永続的ストレージにエラーがありました。	データベースへの接続の確立に失敗したか，データベースが破損しています。
CORBA::REBIND	クライアントが受け取った IOR が QOS ポリシーと競合しています。	設定されている QOS ポリシーと競合する IOR をクライアントが受け取るたびに生成されます。RebindPolicy　の値が NO_REBIND，NO_CONNECT，または VB_NOTIFY_REBIND の場合に，バインドされているオブジェクトリファレンスを使って呼び出しを行った結果，オブジェクト転送メッセージまたはロケーション転送メッセージが出されると，この例外が生成されます。
CORBA::TIMEOUT	VisiBroker ORB での処理がタイムアウトになりました。	接続を確立しようとしたり，要求／応答を待機しているときに，指定された時間までに処理が完了しない場合は，TIMEOUT 例外が生成されます。 • 0x56420001: 接続タイムアウト（接続タイムアウト以内に接続できませんでした） • 0x56420002: 要求タイムアウト（指定されたタイムアウト以内に要求を送信できませんでした） • 0x56420003: 応答タイムアウト（指定されたラウンドトリップタイムアウト以内に応答が受信されませんでした）
CORBA::TRANSACTION_REQUIRED	要求が null のトランザクションコンテキストを保持しており，アクティブなトランザクションが必要です。	トランザクションの一部として実行する必要があるメソッドが呼び出されたが，クライアントスレッドにアクティブなトランザクションがありません。
CORBA::TRANSACTION_ROLLEDBACK	要求に関連付けられているトランザクションは，すでにロールバックされたか，ロールバック対象としてマークされています。	トランザクションは，すでにロールバック対象としてマークされているため，要求されたオペレーションを実行できません。
CORBA::TRANSACTION_MODE	IOR の TransactionPolicy と現在のトランザクションモードの間に不一致が検出された場合に，VisiBroker ORB によって生成されます。	
CORBA::TRANSACTION_UNAVAILABLE	トランザクションサービスとの接続が異常終了したため，トランザクションサービスコンテキストを処理できない場合に，VisiBroker ORB によって生成されます。	

（続き）

例外	説明	考えられる原因
CORBA::TRANSIENT	エラーが発生しましたが，VisiBroker ORB は処理を再試行できると認識しています。	通信障害が発生した可能性があります。VisiBroker ORB は，通信に失敗したサーバーにリバインドするための信号を送っています。enable_rebind() メソッドで BindOptions が false に設定されている場合，または RebindPolicy が正しく設定されている場合，この例外は生成されません。クライアントまたはサーバーのマシンにおけるリソースの制限（最大接続数に達するなど）により，新しい接続の要求が失敗しました。システム例外が原因で TRANSIENT 例外が発生した場合は，COMM_FAILURE のマイナーコードにシステムエラー番号が設定されます。マイナーコードでシステム固有のエラー番号（たとえば，include/sys/errno.h または msdev¥include¥winerror.h files ファイルに含まれる）をチェックします。
CORBA::UNKNOWN	VisiBroker ORB は生成された例外を識別できませんでした。	サーバーが生成した例外が Java 実行時例外などの正しい例外ではありません。サーバーとクライアントの間で IDL の不一致があり，この例外がクライアントプログラムで定義されていません。DII では，コンパイル時にクライアントには未知の例外がサーバーから生成され，クライアントが CORBA::Request の例外リストを指定していない場合に生成されます。問題の原因となった実行時例外を調べるには，サーバーで vbroker.orb.warn=2 のプロパティを設定します。
CORBA::UnknownUser	ユーザー例外を受け取りましたが，クライアントがコンパイル時にこの例外を認識していません。	クライアントがサーバーからのユーザー例外を読み取るとき，コンパイル時にその型の例外を認識していない場合は，この例外が生成されます。クライアントは例外の型を調べることができ，例外の内容が入ったマーシャリング済みのバッファを提供されます。VisiBroker ORB は，自分で例外をアンマーシャリングすることはできません。

システム例外	マイナーコード	説明
BAD_PARAM	1	value ファクトリの登録，登録解除，検索の失敗。
BAD_PARAM	2	RID がインターフェースリポジトリ内にすでに登録されている。
BAD_PARAM	3	名前がインターフェースリポジトリ内のコンテキストですでに使用されている。
BAD_PARAM	4	ターゲットが有効なコンテナでない。
BAD_PARAM	5	継承されたコンテキスト内での名前の競合。
BAD_PARAM	6	抽象インターフェースの型が正しくない。
MARSHAL	1	value ファクトリが見つからない。
NO_IMPLEMENT	1	ローカルの値のインプリメンテーションがない。
NO_IMPLEMENT	2	値のインプリメンテーションの実装に互換性がない。
BAD_INV_ORDER	1	インターフェースリポジトリ内の依存関係のため，オブジェクトを破棄できない。
BAD_INV_ORDER	2	インターフェースリポジトリ内の破棄できないオブジェクトを破棄しようとした。
BAD_INV_ORDER	3	オペレーションがデッドロック状態にある。
BAD_INV_ORDER	4	VisiBroker ORB がシャットダウンした。
OBJECT_NOT_EXIST	1	非アクティブ化された（登録解除された）値をオブジェクトリファレンスとして渡そうとした。

OMG 指定のヒューリスティックな例外

*ヒューリスティックな*決定とは，最初に VisiTransact Transaction Service によって決定される合意結果を取得せず，トランザクションの参加者による一方的な決定によって更新をコミットまたはロールバックすることです。ヒューリスティックの詳細については，『トランザクションの完了』を参照してください。

次の表に，OMG CORBA サービス仕様で定義されているヒューリスティックな例外をリストし，それらの例外が生成される原因について説明します。

表 34.1 OMG CORBA サービス仕様で定義されているヒューリスティックな例外

例外	説明	考えられる原因
`CosTransactions::HeuristicCommit`	ヒューリスティックな決定が行われ，すべての関連する更新がリソースによってコミットされました。	VisiTransact Transaction Service が，すでに作業をコミットするようにヒューリスティックな決定を行っているリソースオブジェクトの `rollback()` を呼び出しました。リソースは，`HeuristicCommit` 例外を生成して，VisiTransact Transaction Service に自分の状態を通知します。
`CosTransactions::HeuristicHazard`	リソースは，ヒューリスティックな決定を行ったかどうかはわかりませんが，関連するすべての更新が行われたかどうかを認識していません。認識されている更新は，すべてコミットされたか，ロールバックされています。この例外は，`HeuristicMixed` より優先されます。	VisiTransact Transaction Service は，ヒューリスティックな決定を行ったかどうかがわからないリソースオブジェクトの `commit()` または `rollback()` を呼び出しました。 リソースは，この例外を生成して，自分の状態が完全には認識していないことを VisiTransact Transaction Service に通知します。VisiTransact Transaction Service は，すべてのリソースが更新を行ったかどうかがわからない場合，アプリケーションにこの例外を返します。
`CosTransactions::HeuristicMixed`	ヒューリスティックな決定が行われ，関連する更新の一部はコミットされ，それ以外はロールバックされました。	VisiTransact Transaction Service は，ヒューリスティックな決定を行ったが関連するすべての更新を行っていないリソースオブジェクトの `commit()` または `rollback()` を呼び出しました。 リソースは，この例外を生成して，自分の状態が完全には一致していないことを VisiTransact Transaction Service に通知します。VisiTransact Transaction Service は，混合した応答をリソースから受け取った場合，この例外をアプリケーションに返します。
`CosTransactions::HeuristicRollback`	ヒューリスティックな決定が行われ，すべての関連する更新がリソースによってロールバックされました。	VisiTransact Transaction Service が，すでに作業をロールバックするようにヒューリスティックな決定を行っているリソースオブジェクトの `commit()` を呼び出しました。リソースは，`HeuristicRollback` 例外を生成して，VisiTransact Transaction Service に自分の状態を通知します。

OMG 指定のその他の例外

次の表に，OMG CORBA サービス仕様で定義されているその他の例外をリストし，それらの例外が VisiTransact Transaction Service によって生成される原因について説明します。詳細については，『トランザクション処理の概要』を参照してください。

表 34.2 OMG CORBA サービス仕様で定義されているその他の例外

例外	説明	考えられる原因
`CosTransactions::Inactive`	トランザクションがすでに準備されているか，終了しています。	この例外は，トランザクションがすでに準備された後で，`register_synchronization()` が呼び出されると生成されます。
`CosTransactions::InvalidControl`	無効な Control が渡されました。	この例外は，`resume()` が呼び出され，そのパラメータが null オブジェクトリファレンスではないが，現在の実行環境で有効でもない場合に生成されます。

表 34.2　OMG CORBA サービス仕様で定義されているその他の例外（続き）

例外	説明	考えられる原因
CosTransactions::NotPrepared	リソースが準備されていません。	まだ準備されていないリソースで replay_completion() または commit() を呼び出すと，この例外が生成されます。
CosTransactions::NoTransaction	クライアントスレッドに関連付けられたトランザクションがありません。	commit()，rollback()，または rollback_only() メソッドは，クライアントスレッドに関連付けられているトランザクションがない場合に呼び出されると，この例外を生成します。
CosTransactions::NotSubtransaction	現在のトランザクションはサブトランザクションではありません。	ネストされたトランザクションがサポートされていないため，この例外は VisiTransact Transaction Manager では生成されません。かわりに NoTransaction 例外が生成されます。
CosTransactions::SubtransactionsUnavailable	クライアントスレッドには，すでに関連付けられたトランザクションがあります。VisiTransact Transaction Service は，ネストされたトランザクションをサポートしません。	トランザクションがすでに開始された後で，begin() 呼び出しが実行されています。トランザクション内でトランザクションオブジェクトを操作する必要がある場合は，begin() を呼び出す前に，トランザクションがすでに開始されていないかどうかを確認する必要があります。 create_subtransaction() メソッドが呼び出されましたが，VisiTransact Transaction Manager はサブトランザクションをサポートしません。
CosTransactions::SynchronizationUnavailable	Coordinator は Synchronization オブジェクトをサポートしません。	Synchronization オブジェクトがサポートされているため，この例外は VisiTransact Transaction Manager では生成されません。
CosTransactions::Unavailable	要求されたオブジェクトを提供できません。	Control::get_terminator() または Control::get_coordinator() が呼び出されたときに，Control オブジェクトが Terminator オブジェクトまたは Coordinator オブジェクトを提供できません。 VisiTransact Transaction Service が PropagationContext の利用を制限しており，Coordinator::get_txcontext() が呼び出されても PropagationContext を返しません。
CORBA::WrongTransaction	遅延同期要求への応答を返す際に，ORB によって生成されます。この例外は，要求の発行時に要求が現在のトランザクションに暗黙的に関連付けられている場合にだけ生成されます。	get_response() メソッドと get_next_response() メソッドは，要求に関連付けられているトランザクションが呼び出し元のスレッドに関連付けられているトランザクションとは異なる場合，この例外を生成します。

第 35 章

VisiBroker プラグイン可能トランスポートインターフェース

VisiBroker for C++ には，ORB に組み込まれているプロトコル以外に CORBA 呼び出しを伝達するためのトランスポートプロトコルを使用できるように，プラグイン可能トランスポートインターフェースが提供されています。このインターフェースは，複数のトランスポートプロトコルを同時に“プラグイン”でき，さまざまなトランスポートタイプに使用できる共通のインターフェースを提供するように設計されています。このインターフェースは，可能な限り CORBA 標準クラスを使用していますが，それ自体は VisiBroker 独自のインターフェースです。

プラグイン可能トランスポートインターフェースファイル

VisiBroker プラグイン可能トランスポートインターフェースは，次のライブラリおよびサポートヘッダーファイルとして配布されます。

- DLL pluggable<bitmode>_<p>r_<version>.dll（Windows の場合）
- libpluggable<bitmode>_<p>r.so.<version>（Solaris と Linux の場合）
- libpluggable<bitmode>_<p>r.sl.<version>（HP-UX の場合）
- libpluggable<bitmode>_<p>r.a.<version>（AIX の場合）
- ヘッダーファイル vptrans.h は include ディレクトリにあります。

ここで，bitmode は，64 ビットプラットフォームの場合は 64，p は標準の C++ バージョン，version は VisiBroker のバージョンです。ライブラリの API は，`vptrans.h` ヘッダーファイルに公開されています。

トランスポート層の要件

プラグイン可能トランスポートインターフェースを介して VisiBroker にプラグインされるトランスポートプロトコルは，CORBA 仕様として定義されている標準 GIOP プロトコルを使用してエンコードされたメッセージを送受信するために ORB によって使用されます。

GIOP では，これらのメッセージの交換に使用されるトランスポート層について，いくつかのことが想定されています。プラグイン可能トランスポートインターフェースの設計にも，同じ想定が使用されています。したがって，特定のトランスポートと ORB のインターフェースとなるユーザーコードは，これらの要件と実際のトランスポートの動作に相違がある場合，これを“ マスク ”する必要があります。

プラグイン可能トランスポートインターフェースは，以下のことを想定しています。

- トランスポートの単一サーバーエンドポイントとトランスポートの単一クライアントエンドポイントの間の“ポイントツーポイント”のデータ送信には,信頼できる双方向データ交換チャネル（接続）が使用されること。したがって，サーバーからの応答メッセージは，この接続から要求が送信された後で接続エンドポイントを検査することで，信頼して受信されると想定されます。ただし，これにより，ORB が同じサーバーへの複数クライアント要求に同じ接続を使用できなくなることはありません。
- トランスポートを介して送信されるデータは（原則として），サイズの制限がなく，連続するバイトストリームとみなすことができること。データのパッケージングと，フロー制御，パッケージのリアセンブリ，およびエラーハンドリングに関連する問題は，すべて隠蔽される必要があります。
- クライアントからの要求時に接続を動的に開き，また閉じることできること。接続を開く要求は特定のエンドポイントに対して行われ，このエンドポイントは，クライアントがサーバーによって生成される IOR から取得します。なお，接続要求メッセージは GIOP プロトコルに含まれませんが，プラグイン可能トランスポート接続管理の対象になり，トランスポート固有のコードで処理する必要があります。
- CORBA 仕様で指定されているように，サーバー接続エンドポイントが IOR に格納できる方法で記述されていること。このエンドポイントは，トランスポートのアドレッシングスキーム内で固有で，サーバーへのコンタクトにいつでも使用できる必要があります。CDR 準拠表現のエンドポイントアドレスを作成し，IOR 内のプロファイルに入れて使用できるように，変換機能を提供する必要があります。

プロトコルプラグインに必要なユーザー提供コード

ユーザーは，プラグイン可能トランスポートインターフェースを介して ORB にプラグインされるトランスポートプロトコルごとに，次の 3 つのメインクラスを実装する必要があります。

1. 接続クラス - データをトランスポート層に書き込み／読み取り，そのデータをクライアントとサーバーの間の特定の“接続”に関連付ける方法を提供します。“接続”という概念が使用されていても，使用する物理トランスポート層が接続指向の IO をサポートする必要があるという意味ではありませんが，ユーザーコードでは，このようなビューをプラグイン可能トランスポートインターフェースに提示し，以下に説明する関連機能をすべて提供する必要があります。
2. リスナークラス - トランスポートのサーバー側“エンドポイント ”を表します。このクラスは,クライアント要求を受け取りって“ 接続 ”インスタンスを作成し,そような接続を動的に開いたり閉じる処理を行い，開いた接続を介して着信クライアント要求の“ ディスパッチ ”を開始します。
3. プロファイルクラス - リスナーインスタンスのサーバー側エンドポイント情報を“ 可搬性 ”がある方法で記述できるようにします。これは，この記述を CORBA 仕様に定義さ

れているように IOR に組み込むことができ，したがって GIOP などの適切なプロトコルを使用して他の ORB と交換できることを意味します。

また，プラグイン可能トランスポートインターフェースは，“ ファクトリ ” パターンを使用して，これらのクラスのインスタンス化を管理します。したがって，それぞれが上のクラスの 1 つのインスタンスを作成する 3 つのファクトリクラスを用意する必要があります。

トランスポートプロトコルは，3 つのファクトリクラスをインスタンス化し，プラグイン可能プロトコルインターフェースを介してそれらを ORB に登録することで初期化されます。システムの初期化段階で，どの CORBA サーバーまたはクライアントコードを起動するより前に，プラグイン可能プロトコルインターフェースの静的関数を呼び出すと，登録が実行されます。

固有のプロファイル ID タグ

各プラグイントランスポートには，他のプロトコルと区別するために 4 バイトのプロファイル ID タグが必要です。プロファイル ID タグは，OMG によって管理されます。Borland は，多くのプロファイル ID タグを OMG に登録しており，それらのタグのうち次の 4 つはプロトコルプラグインで使用できます。

1. 0x48454901 ("HEI\001")
2. 0x48454902 ("HEI\002")
3. 0x48454903 ("HEI\003")
4. 0x48454904 ("HEI\004")

ランダムに選択した値ではなく，これらのタグを使用して，サードパーティの CORBA ベース製品との競合を避ける必要があります。

ただし，プロトコルプラグインを使用するシステムが VisiBroker for C++ に基づく他のシステムと統合され，そのシステムに偶然同じプロファイル ID タグを選択したプロトコルプラグインが含まれる場合は，競合の可能性が残ります。これは，同じ組織内で独立して開発された複数のサブシステムが統合される場合や，最終的なシステムが別の組織によって開発された別の CORBA ベースシステムと相互運用する必要がある場合にも発生する可能性があります。

これらのケースに該当する可能性が高い場合は，OMG から予約済みの番号を取得する必要があります。CORBA タグの詳細は，OMG FAQ（ftp://ftp.omg.org/pub/docs/ptc/99-02-01.txt）を参照してください。必要最小限のタグだけを予約する必要があります。ただし，一連のタグを予約できる機会は，通常，年 1 回だけです。番号の予約は，開発したシステムの配布が間近に迫ったときに行うことをお勧めします。

サンプルコード

プラグイントランスポートを実装する方法と CORBA アプリケーションで使用する方法を示す 2 つのサンプルが examples/pluggable ディレクトリに用意されています。このサンプルでは，複雑なトランスポート層の説明でなく，インターフェース自体に重点を置くために，トランスポートとして TCP/IP を使用しています。

新しいトランスポートの実装

vptrans.h ファイルで公開されている以下のインターフェースを実装する必要があります。

VISPTransConnection と VISPTransConnectionFactory

このクラスは，サーバーとクライアントの間の 1 つの接続を表します。プログラムで接続から読み取る／接続に書き込むと，そのデータがリモート側の単一のピアエンドポイントから受信／ピアエンドポイントに送信されます。クライアントからサーバーに要求を送信する場合，ORB は，そのサーバーへの有効な接続を探し，まだない場合は作成します。接続のリモート側のエンドポイントは，サーバーの特定のプロファイルを使用して設定され，サーバー側のリスナー（下の「リスナークラス」を参照）と通信します。このクラスは，一般的な状態情報を提供するほか，（a）接続から着信するデータを待機して，特定の秒数が経過したらタイムアウトになる方法を提供するか，(b) “ プラグイン可能トランスポートブリッジ ” クラスを使用して，着信データがあったときにブリッジに通知することで，この機能を実行する必要があります。

ファクトリクラスは，プラグイン接続のインスタンスを作成するために使用されます。また，静的 *VISPTransRegistrar::addTransport* API を使用してレジストラに登録する必要があります。

```
class _VBPTEXPORT VISPTransConnection {
public:
...

// データをリモートピアに送信します
virtual void write(CORBA::Boolean _isFirst, CORBA::Boolean_isLast,
const char* _data, CORBA::ULong_offset,
CORBA::ULong _length, CORBA::ULongLong _timeout) = 0;


// リモートピアからの送信データを接続から読み取ります
virtual void read(CORBA::Boolean _isFirst, CORBA::Boolean_isLast,
char* _data, CORBA::ULong _offset,
CORBA::ULong _length,
CORBA::ULongLong _timeout) = 0;

// バッファリングトランスポートでデータを即座にフラッシュするために使用します
virtual void flush() = 0;

// 接続を正規に閉じます
virtual void close() = 0;

// リモートピアリスナーと通信して新しい接続を設定します
virtual void connect(CORBA::ULongLong _timeout) = 0;

// 各接続インスタンスの（このトランスポートでの）一意の ID を返す必要があります
virtual CORBA::Long id() = 0;

// リモートピアがまだ接続されている場合は True を返す必要があります
virtual CORBA::Boolean isConnected() = 0;

// データが読み取り可能な状態の場合は True を返す必要があります
virtual CORBA::Boolean isDataAvailable() = 0;

// トランスポートを逆のクライアント／サーバー設定で使用できる場合は True を返す必要が
あります
virtual CORBA::Boolean no_callback() = 0;

// トランスポートが次のメッセージを待機できない場合は True を返す必要があります。これ
により，ORB は
// 次のメッセージを待機する間，ブリッジを使用してタイムアウトになります
virtual CORBA::Boolean isBridgeSignalling() = 0;
```

```
// データが到着するかタイムアウトになるまでブロックします。データがある場合は True を
返す必要があります
virtual CORBA::Boolean waitNextMessage(CORBA::ULong _timeout) = 0;

// ピアエンドポイントを記述するプロファイルのコピーを返す必要があります
virtual IOP::ProfileValue_ptr getPeerProfile() = 0;

// 入力ピアプロファイルはピアについて接続に通知します。接続中に使用されます
virtual void setupProfile(const char* prefix, VISPTransProfileBase_ptr peer) = 0;

};

class _VBPTEXPORT VISPTransConnectionFactory {
public:
...

// 新しい接続インスタンスを作成します。そのポインタ返す必要があります
virtual VISPTransConnection_ptr create(const char* prefix) = 0;

};
```

VISPTransListener と VISPTransListenerFactory

このクラスは，クライアントからの着信接続および要求を待機するためにサーバー側のコードで使用されます。新しい接続と，既存の接続での要求は，プラグイン可能トランスポートインターフェースのブリッジクラス（下の「トランスポートブリッジクラス」を参照）を介して ORB に通知されます。

特定のトランスポートプロトコルを指定するサーバー接続マネージャ（SCM）を含むサーバーエンジンが作成されるたびに，このクラスのインスタンスが作成されます。プロトコルを指定する SCM インスタンスごとに 1 つのインスタンスが作成されます。

既存の接続で要求が受信されると，接続は“ディスパッチサイクル”に入ります。ディスパッチサイクルは，接続がデータをトランスポート層に転送すると開始されます。この初期ステージでは，ブリッジ（下の「トランスポートブリッジクラス」を参照）を介して，このデータの着信を ORB に通知する必要があります。リスナーは，ディスパッチプロセスが完了するまで接続を無視します。このとき，接続は“ディスパッチ状態”にあると言います。ORB がリスナーの completedData() メソッドを呼び出すと，接続が初期状態に戻ります。ディスパッチ状態の間，ORB は，ブリッジ／リスナー間通信によるオーバーヘッドを回避するため，要求がなくなるまで接続から直接読み取りを行います。

ほとんどの場合，トランスポート層は，ブロック呼び出しを使用して新しい接続を待機します。この状況を処理するために，リスナーは，クラス VISThread のサブクラスを作成し，ORB 全体を停止しないでブロックできる独立した実行スレッドを開始する必要があります。

ファクトリインスタンスは，接続と同様に，実装されたプラグインリスナーのインスタンスを返し，*VISPTransRegistrar::addTransportAPI* を使用して登録される必要があります。

```
class _VBPTEXPORT VISPTransListener {
public:
...

// ブリッジとのリンクを確立するために ORB から呼び出されます。これで，
// リスナー／ORB 間通信が有効になります
virtual void setBridge(VISPTransBridge* up) = 0;

// リスナーエンドポイントを記述するプロファイルを返す必要があります
virtual IOP::ProfileValue_ptr getListenerProfile() = 0;

// ORB が特定の ID の要求の読み取りを完了し，新しい要求があれば再度
// ブリッジを介して通知するようにリスナーに要求するときに呼び出されます
virtual void completedData(CORBA::Long id) = 0;

// 特定の ID の接続に読み取り可能なデータがある場合は True を返す必要があります

virtual CORBA::Boolean isDataAvailable(CORBA::Long id) = 0;
```

```
// リスナーがエンドポイントを破棄し，関連するすべてのアクティブな接続を閉じる
// 必要がある場合に呼び出されます
  virtual void destroy() = 0;
};

class _VBPTEXPORT VISPTransListenerFactory {
public:
...

// リスナーの新しいインスタンスを作成します。そのポインタを返す必要があります
virtual VISPTransListener_ptr create(const char* propPrefix) = 0;

};
```

VISPTransProfileBase と VISPTransProfileFactory

このクラスは，トランスポート固有のエンドポイント記述と，他の CORBA インプリメンテーションと交換可能な IOP ベースの IOR との変換機能を提供します。また，ある " 解析 " 関数に ProfileValue を渡すことで，クライアントをサーバーにバインドするプロセスでも使用されます。この関数は，このトランスポートに使用できる IOR が ProfileValue 内にあるかどうかに基づいて，TRUE または FALSE を返す必要があります。

このクラスのインスタンスは，その基底クラス型のポインタを介して煩雑に関数に渡されます。どの C++ コンパイラを使用しても実行時に安全にダウンキャストできるように，キャストが適正かどうかをテストする "_downcast" 関数を提供する必要があります。

追加クラス - VISPTransBridge と VISPTransRegistrar

ユーザー提供のトランスポートプラグインコードから呼び出されるさらに 2 つのクラスがプラグイン可能トランスポートインターフェースに用意されています。

VISPTransBridge は，ORB とトランスポートプラグインの間でさまざまなイベントを通信する汎用インターフェースです。たとえば，次の通信があります。

1. 新しい接続要求について ORB に通信する
2. 新しい入力データについて ORB に通信する
3. ピア接続を閉じることについて ORB に通信する

```
Class _VBPTEXPORT VISPTransBridge {
public:

// 接続ポインタを渡して，新しい接続要求について ORB に通知します。
// ORB が接続を受け入れた場合は True を返し，そうでない場合 False を返します
virtual CORBA::Boolean addInput(VISPTransConnection_ptr con);

// 接続での新しい要求を ORB に通知します。通常は，これによってディスパッチサイクルが
// 開始されます
virtual void signalDataAvailable(CORBA::Long conId);

// リモートピアによって接続が閉じられたことを ORB に通知します
virtual void closedByPeer(CORBA::Long conId);
};
```

VISPTransRegistrar は, 新しいトランスポートを ORB に登録するために使用する必要があるクラスです。登録時に指定される文字列は，このトランスポートの識別名として使用され，その ORB の範囲内で一意である必要があります。また，このトランスポートに関連するプロパティのプレフィクス文字列としても使用されます。

```
class _VBPTEXPORT VISPTransRegistrar {
public:
// トランスポートの識別名文字列と，この新しいトランスポートの特定のインスタンスに使用
される
// 3 つのファクトリを登録します
```

```
static void addTransport(const char* protocolName
                         VISPTransConnectionFactory* connFac,
                         VISPTransListenerFactory*   listFac,
                         VISPTransProfileFactory*    profFac);
};
```

第 36 章

VisiBroker ログ

VisiBroker for C++ のログメカニズムを使用すると，アプリケーションでログメッセージしたり，アペンダという構成可能なログ転送機能を介して，ログメッセージを適切な宛先に転送することができます。ORB とすべての ORB サービス自体も，このメカニズムを使用して，すべてのエラー，警告，または情報メッセージを出力します。アプリケーションは，この機能を使用して，アプリケーションのメッセージと ORB のメッセージを同じ場所にあるログに記録してシステム全体で 1 つのメッセージログを作成することも，ソースが異なるメッセージを別々の場所にあるログに記録することもできます。フィルタとレイアウトは，ログメッセージを絞り込んだりフォーマットする機能を追加します。

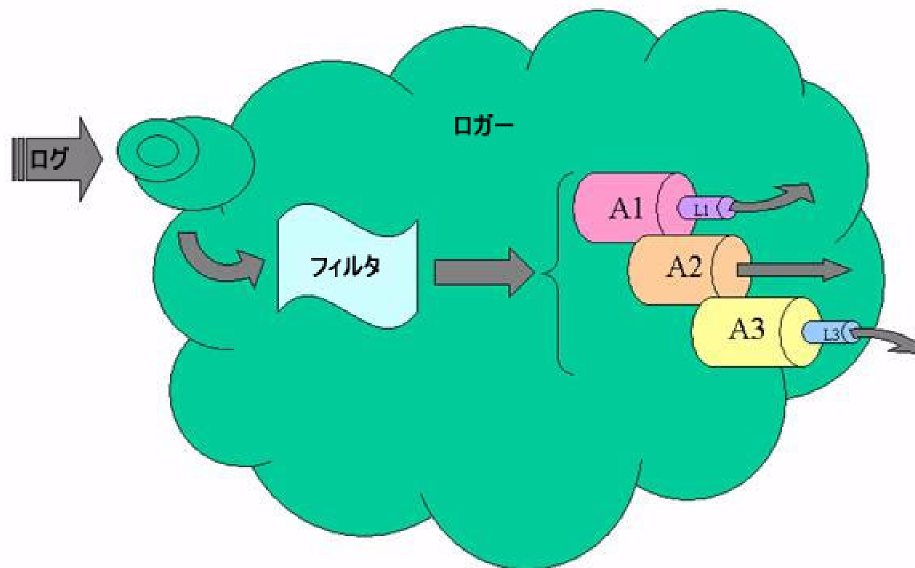
このログメカニズムは，OMG の Telecom Logging Service（VisiTelcoLog）とは異なります。これは，C++ 専用の軽量ログライブラリで，ORB と ORB サービス（VisiTelcoLog など）が内部メッセージのログに使用します。ログフレームワーク全体と，組み込みアペンダおよびレイアウトは，vdlog<bitmode>_<p>r_<version>.dll（Windows），libvdlog<bitmode>_r.so.<version>（Solaris および Linux），libvdlog<bitmode>_<p>r.sl.<version>（HPUX），または libvdlog<bitmode>_<p>r.a.<version>（AIX）という名前の共有ライブラリにあります。ここで，p は標準 C++ ライブラリ，bitmode は，64 ビットプラットフォームの場合は 64，version は VisiBroker のバージョンです。ライブラリの API は，vdlog.h ヘッダーファイルに公開されています。

ログの概要

VisiBroker のログは 1 つ以上のロガーオブジェクトを使用し，（ORB を含む）アプリケーションは，そのオブジェクトにログメッセージを記録します。ORB とすべての ORB C++ サービスは，特別なロガーインスタンス（「default」という名前の “ デフォルトロガー ”）を使用します。これは，ORB が初めてログメッセージを記録するときに自動的に作成されます。アプリケーションは，やはりデフォルトロガーにログメッセージを記録してアプリケーションのログ出力と ORB のログ出力を統合することも，別に 1 つ以上のロガーを作成して前述のようにログメッセージを独立に記録することもできます。

フレームワーク内の各ロガーには 1 つ以上のアペンダを関連付けることができます。各アペンダにはすべてのログメッセージが順に転送され，アペンダはそれらを標準エラー，ファイル，ネットワーク，OMG Telecom Logging Service ログなどのそれぞれの宛先に出力

する役割を持ちます。次の図では，1 つのロガーが A1，A2，A3 の 3 つのアペンダに関連付けられています。各アペンダはタイプが異なり，異なるタイプの宛先に転送を行います。



アペンダは，メッセージを転送するとともに，出力の前にログをフォーマットするために設定済みのレイアウトを選択することもできます。ここでは，アペンダ A1 と A3 がレイアウト L1 と L3 を使用しています。明示的に設定されていない場合，ロガーは，デフォルトで「simple」タイプのレイアウトインスタンスを使用し，「stdout」タイプのアペンダにログを記録します。

ロガーは，ログメッセージをフィルタリングする機能も備えています。ロガーは，一連のアペンダにログメッセージを送信する前に，フィルタを使用してメッセージのソースの名前やログレベルを処理し，その結果に基づいて，メッセージをアペンダに転送するか破棄するかを判断します。許容するソース名とログレベルの設定を変更することで，フィルタリングを厳密に行うことができます。

この章では，次の項目について説明します。

- ロガーマネージャ
- ログ
- フィルタリング
- カスタムアペンダおよびレイアウト
- 設定

ロガーマネージャ

ロガーマネージャは，ロガーフレームワークの機能を使用するための出発点です。ロガーマネージャの主要な機能の 1 つは，ロガーのライフサイクルを管理することです。ロガーマネージャはシングルトンオブジェクトで，ロガーマネージャへのリファレンスは，その静的インスタンスメソッドを呼び出して取得できます。ロガーマネージャでは参照カウントが行われません。次のコードは，静的インスタンス関数を使用して，シングルトンロガーマネージャオブジェクトにアクセスする方法を示します。

```
// 静的インスタンス関数を使用してロガーマネージャ参照を取得します
VISLoggerManager_ptr logger_manager = VISLoggerManager::instance();
VISLogger_ptr logger = logger_manager->get_default_logger();
...
// また，ロガーマネージャ参照は，ロガーマネージャの使用時に毎回取得する
// こともできます。これは，get_default_logger メソッドを呼び出す場合の例です
VISLogger_ptr logger = VISLoggerManager::instance()->get_default_logger();
```

ロガーにアクセスできるようにするほか，このシングルトンには，カスタムアペンダ／レイアウトファクトリのレジストラとしての役割もあります。さらに，グローバル有効化スイッチと詳細レベルも提供および設定します。

ログ

前述のように，アプリケーションでは，デフォルトロガーまたは個別ロガーのいずれかを使用してログメッセージを記録するようにログインターフェースを使用できます。1 つのロガーに送信されるすべてのログメッセージは，いくつかの共通の宛先に転送されることになります。また，ログに複数のロガーを使用することで，さまざまなコンポーネントからのメッセージをさまざまな個別のエンドポイントに出力できます。

次のコードでは，サーバーアプリケーションがデフォルトロガーを使用してアプリケーション固有のログメッセージを記録しています。ソース名「bankagentserver」を使用してサーバーコードからのログメッセージを識別し，「bankagentimpl」を使用してインプリメンテーションコードからのログメッセージを識別します。ソース名は，モジュール化のツールとしてたいへん便利で，フィルタ設定の段階で役立つことがわかります。

```
#define MYLOG(LVL, COMP, MESG) \
    if (VISDLoggerMgr::instance()->global_log_enabled()) { \
       VISDLoggerMgr::instance()->get_default_logger()->log( \
            LVL, COMP, MESG \
            , __FILE__, __LINE__ \
            ); \
    }

#define MYLOGDBG(COMP, MESG) \
  MYLOG(VISDLogLevel::DEBUG_, COMP, MESG)
#define MYLOGINF(COMP, MESG) \
  MYLOG(VISDLogLevel::INFO_, COMP, MESG)

#define BAS "bankagentserver"
#define BAS_LOGDBG(MESG) MYLOGDBG(BAS,MESG)
#define BAS_LOGINF(MESG) MYLOGINF(BAS,MESG)

#define BAI "bankagentimpl"
#define BAI_LOGDBG(MESG) MYLOGDBG(BAI,MESG)
#define BAI_LOGINF(MESG) MYLOGINF(BAI,MESG)

int main(int argc, char* const* argv)
{
  BAS_LOGINF("Bank agent server start");
```

```
try {
  BAS_LOGINF("Initializing ORB");
  // ORB を初期化します。
  CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

 BAS_LOGINF("Resolving Initial reference to Root POA");
  // ルート POA へのリファレンスを取得します。
  CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
...
}
```

また，次のようにマクロ MYLOG を変更して，アプリケーションのログとして固有のロガーを使用するように選択することもできます。

```
#define MYLOG(LVL, COMP, MESG) \
    if (VISDLoggerMgr::instance()->global_log_enabled()) { \
       VISDLoggerMgr::instance()->get_logger("mylogger")->log( \
            LVL, COMP, MESG \
            , __FILE__, __LINE__ \
            ); \
    }
```

ソース「bankagentserver」および「bankagentimpl」からのすべてのログメッセージは，「mylogger」ロガーと，その設定アペンダに転送されます。ORB からのログメッセージは，デフォルトロガーに設定されたアペンダに転送されます。

フィルタリング

トラブルシューティングを実施する際，特定のモジュールまたはソース名からのログメッセージをフィルタリングで除外したり，詳細レベルを変更できるとたいへん便利な場合があります。ロガーフレームワークは，ロガーフィルタを使用した高度なフィルタリングメカニズムを提供できる強力かつ簡単な方法を備えています。各ロガーのフィルタは，メッセージをアペンダに転送するかどうかの基準として，ログメッセージの 2 つの要素を考慮します。各ログメッセージは，ログのソースとログレベルに関する情報を含んでいます。フィルタは，この情報を使用し，次のロジックにしたがって転送を判断します。

- If（ソース名がフィルタに登録されている）
  - If（ソース名が有効）
    - If（ログのログレベルがソースのログレベル以上）
      - 転送
    - Else
      - 却下
  - Else
    - 却下
- Else If（ソース名「all」が有効）
  - If（ログのログレベルが（ログマネージャでの）グローバルログレベルの設定以上）
    - 転送
  - Else
    - 却下
- Else
  - 却下

「all」は，フィルタに登録されていないすべてのソースを示す特別なソース名です。

ORB は，次のソース名を使用して，それ自体がモジュール化されています。

- connection：クライアント側接続，サーバー側接続，接続プールなどの接続関連ソース領域のログ
- client：クライアント側呼び出しパスのログ
- agent：Osagent 通信のログ
- cdr：GIOP 領域のログ
- se：ディスパッチャ，リスナーなどのサーバーエンジンのログ
- server：サーバー側呼び出しパスのログ
- orb：ORB からのログ出力

各サービスも，そのコンポーネントをモジュール化し，適切なソース名を使用しています。

以下に，いくつかのフィルタリングの目的と，それを実現する設定の例を示します。プロパティの詳細は，後の設定のセクションを参照してください。

次のプロパティは，ログを有効にし，グローバルな詳細レベルを info に設定します。これよりレベルが低いログメッセージはフィルタリングで除外されます。

```
vbroker.log.enable=true
vbroker.log.logLevel=info
```

次のプロパティは，ログを有効にし，osagent 通信を実行するコンポーネントからのログメッセージだけを除外します。その他のログメッセージはすべて記録されます。

```
vbroker.log.enable=true
vbroker.log.default.filter.register=agent
vroker.log.default.filter.agent.enable=false
```

次のプロパティは，ログを有効にし，osagent 通信を実行するコンポーネントからの詳細レベルが info より低いメッセージを除外して，他のすべてのログメッセージを許可します。

```
vbroker.log.enable=true
vbroker.log.default.filter.register=agent
vbroker.log.default.filter.agent.enable=true
vbroker.log.default.filter.agent.logLevel=info
```

次のプロパティは，ログを有効にし，osagent 通信を実行するコンポーネントからの詳細レベルが info 以上のログメッセージだけを許可します。他のすべてのメッセージはフィルタリングで除外されます。

```
vbroker.log.enable=true
vbroker.log.default.filter.register=agent
vbroker.log.default.filter.agent.enable=true
vbroker.log.default.filter.agent.logLevel=info
vbroker.log.default.filter.all.enable=false
```

予約名

以下の名前は予約されているため，ロガー，アペンダ／レイアウトタイプ，アペンダインスタンス，またはソース名としては使用できません。「default」，「appender」，「appenders」，「layout」，「filter」，「simple」，「full」，「xml」，「stdout」，「rolling」，「all」，「v」で始まるすべての名前。このような文字列を使用した場合の動作は予測できません。

カスタマイズ

組み込みのアペンダとレイアウトでは十分でない場合は，カスタムオブジェクトを実装し，実行時にロガーフレームワークによってロードされるように共有ライブラリとして提供できます。

それには，次の手順を実行します。

1 共有ライブラリまたは DLL で VISDAppenderFactory および VISDAppender インターフェースを実装します。
2 register_app_factory を使用して, 実装されたファクトリのグローバルインスタンスをコンストラクタでロガーマネージャに登録する必要があります。
3 以下に説明するプロパティ設定を使用して，ロガーフレームワークが共有ライブラリをロードし，このカスタムファクトリとそのアペンダを使用するように指定します。

カスタムレイアウトについても同様の手順を実行できます。

たとえば，ログメッセージだけを出力し，他の詳細はすべて省略するカスタムレイアウトを使用して，コンソールにログを記録する独自のアペンダを使用する場合は，次に示すように，まずアペンダとレイアウトのインターフェースを実装する必要があります。

次のコードは，アペンダファクトリとアペンダを実装するクラスを示します。

```
class StdOutAppFactory : public VISDAppenderFactory {
public:
  // コンストラクタ
  StdOutAppFactory() {
  // グローバルインスタンスオブジェクトが作成されたときに登録します
  VISDLoggerMgr::instance()->register_app_factory(this);
  }
  ...
  // このカスタムアペンダの一意のアペンダタイプ名
  virtual const char* type_name() { return "mystdout"; }
  // このタイプのアペンダインスタンスが必要な場合に,
  // フレームワークから呼び出される API
  virtual VISDAppender_ptr create(const char* logger_name,
  VISDConfig::LogAppenderConfig_ptr p);
  // このファクトリによって作成されたアペンダインスタンスを破棄する必要がある場合に,
  // フレームワークから呼び出される API
  virtual void destroy(VISDAppender_ptr app);
  // ライブラリまたは DLL がロードされたときに作成される
  // グローバルインスタンスオブジェクト
  static StdOutAppFactory _instance;
};
class StdOutApp : public VISDAppender {
public:
  ...
  // アペンダが ORB 機能を使用する場合は TRUE，そうでない場合は FALSE を返す必要があり
ます
  // このアペンダタイプは ORB 機能を必要としないため
  // FALSE を返します
  virtual CORBA::Boolean ORB_initialized(void* orb_ptr);
  // シャットダウン通知後，ORB 機能は使用できません
  virtual void ORB_shutdown();
  // 実際のアペンダインプリメンテーション。アペンダのオペレーションが成功した場合は
  // TRUE を返す必要があります
  virtual CORBA::Boolean append(const VISDLogRecord& record);
  ...
};
```

次のコードは，同様にカスタムレイアウトのインプリメンテーションを示します。

```
class SimpleLayoutFactory : public VISDLayoutFactory {
public:
  // コンストラクタ
  SimpleLayoutFactory() {
    // グローバルインスタンスオブジェクトが作成されたときに登録します
    VISDLoggerMgr::instance()->register_lyt_factory(this);
  }
  // このレイアウトの一意のタイプ名
  virtual const char* type_name() { return "mysimple"; }
  // ロガーフレームワークは，このタイプのレイアウトインスタンスが必要な場合に,
  // この API を呼び出します
  virtual VISDLayout_ptr create(const char* logger_name,
```

```
    VISDConfig::LogAppenderConfig_ptrp);
    // このファクトリによって作成されたレイアウトインスタンスを破棄する場合に，
    // この API を呼び出します
    virtual void destroy(VISDLayout_ptr layout);
    // ライブラリまたは DLL がロードされたときに作成される
    // グローバルインスタンスオブジェクト
    SimpleLayoutFactory _instance;
    ...
  };
  class SimpleLayout : public VISDLayout {
  public:
    ...
    // このレイアウトを使用してメッセージをフォーマットするために
    // アペンダから呼び出される API
    virtual void format(const VISDLogRecord& record,
                        char* buf,
                        CORBA::ULong buf_size,
                        CORBA::String_var& other_buf);
    ...
  };
```

上のコードが，たとえば，Custom.dll（または libCutom.so）にビルドされた場合は，次のプロパティを使用して，フレームワークがこれを使用するように指定できます。

```
vbroker.log.enable=true

# フレームワークが使用するアペンダとレイアウトのタイプを定義します

vbroker.log.appender.register="mystdout"

vbroker.log.appender.mystdout.sharedLib=Custom.dll (or libCustom.so)

vbroker.log.layout.register="mysimple"

vbroker.log.layout.mysimple.sharedLib=Custom.dll (or libCustom.so)

# 上のカスタムタイプのインスタンスをデフォルトロガーに関連付けます

vbroker.log.default.appenders=app1

vbroker.log.default.appender.app1.appenderType=mystdout

vbroker.log.default.appender.app1.layoutType=mysimple
```

実行時，デフォルトロガーが最初にアクセスされると，フレームワークは，設定情報を読み取り，デフォルトロガーに必要なアペンダが共有ライブラリにあることを認識します。さらに，共有ライブラリに含まれるカスタムオブジェクトが register_<>_factory API を使用して自分自身をロガーマネージャに登録しているものとして，共有ライブラリをロードし，ロガーをアセンブルします。

設定

ロガーフレームワークのすべてのセットアップは，設定と実行時プロパティベースのメカニズムを介して行われます。次の項目を設定できます。

1 ロガーフレームワークが有効かどうかを示すロガーマネージャ上のグローバルスイッチと，メッセージの詳細レベル
2 カスタムアペンダ／レイアウトファクトリの登録
3 ロガー上のアペンダ設定と，各ロガー上の個別アペンダインスタンスの設定
4 フィルタリングと詳細レベルを細かく制御するためのロガー上のフィルタ設定

ログマネージャの設定

```
vbroker.log.enable={true|false}
```

上のプロパティ設定は，ロガーマネージャを有効または無効にします。入力値は「True」または「False」で，デフォルト値は「False」です。

```
vbroker.log.logLevel={emerg|alert|crit|err|warning|notice|info|debug}
```

上のプロパティ設定は，ロガーフレームワークのグローバルな詳細レベルを大まかに設定します。ただし，この設定は，ロガーにフィルタを設定することによってさらに詳細に制御できます。デフォルトで選択される値は debug です。

ログレベルは全体で 8 レベルありますが，ORB とすべての ORB サービスは，次の 4 レベルだけを使用します。

- debug - 最低のレベルです。開発者がアプリケーションをデバッグする際に最も役立つ詳細な情報イベントを指定します。オフラインデバッガに似ています。たとえば，パラメータまたは引数の値，マーシャリングバッファなどの複雑なデータ構造の内容，接続ワイヤ上のメッセージのような特定のメモリ内容のピークなどがあります。
- info - アプリケーションの進捗を大まかなレベルで通知する情報メッセージを指定します。これらは一般的なトレースステートメントです。これにより，オブジェクトが作成および破棄されるようす，さまざまな呼び出し元および呼び出し先関数のフロー，特定のアクションが実行されるようす，およびさまざまなコンポーネントが相互に対話するようすをリストにして表示できます。
- エラー - 引き続きアプリケーションを実行できる程度のエラーイベントを指定します。これらは，エラー条件が検出されたが，修正アクションを行って，実行を継続できるような場合です。例外ハンドラのログステートメントは，このレベルにすることができます。
- emerg - 最高のレベルです。アプリケーションが中断されるような非常に重大なエラーイベントを示します。これらは，ORB が機能上の要件を継続できず，修正アクションを行うこともできないため，予測できない動作が発生する可能性がある場合です。

アペンダとレイアウトの登録設定

```
vbroker.log.appender.register=<comma separated list of appender type names>
vbroker.log.appender.<at>.sharedLib=< shared library file >
vbroker.log.layout.register=<comma separated list of layout type names>
vbroker.log.layout.<lt>.sharedLib=<shared library file>
```

上のプロパティを使用して，カスタムアペンダ／レイアウトのタイプ名と，共有ライブラリおよび DLL 内でのインプリメンテーションの場所をロガーフレームワークに通知するこができます。ここで，「at」と「lt」はそれぞれ，導入されるタイプ名のコンマ区切りリストに含まれる，アペンダとレイアウトのタイプ名です。

ロガーでのアペンダとレイアウトの設定

```
vbroker.log.<ln>.appenders=<comma separated list of app instance names>
vbroker.log.<ln>.appender.<an>.appenderType=<at>
vbroker.log.<ln>.appender.<an>.layoutType=<lt>
```

上のプロパティを使用して，ロガーにアペンダインスタンスを設定できます。「ln」はロガー名を表します。最初の 2 つのプロパティを使用して，ロガーに関連付けられるすべてのアペンダインスタンス名と，各アペンダのタイプをロガーフレームワークに通知します。組み込まれていないアペンダタイプがある場合は，前述のプロパティで説明したように，共有ライブラリがロードされ，アペンダが取得されます。ただし，ロガーフレームワークは，ロードされる共有ライブラリに実装されているすべてのアペンダとレイアウトファクトリがロガーマネージャに自動的に登録されると想定することに注意してください。

3 番めのプロパティは，レイアウトのアペンダインスタンスを指示します。アペンダがレイアウトを使用しない場合，この情報は無視されます。そうでない場合は，このレイアウトタイプのインスタンスが取得されます。

フレームワークは，カスタムアペンダ／レイアウトの使用方法を提供するほかに，いくつかのアペンダ／レイアウトタイプが組み込まれています。「stdout」は，そのすべてのメッセージをコンソールに出力し，「rolling」は，ローリングファイルベースのデータストアへのログアペンドオペレーションを実行します。これらのアペンダはどちらもレイアウトを使用し，「simple」，「full」，「xml」レイアウトまたはカスタムレイアウトを設定できます。「xml」は，メッセージを Log4J xml フォーマットにします。

フィルタ設定

```
vbroker.log.<ln>.filter.register=<Comma separated source names>
vbroker.log.<ln>.filter.<sn>.enable=true/false
vbroker.log.<ln>.filter.<sn>.logLevel={emerg|alert|crit|err|warning|notice|info|debug}
```

出力される各ログメッセージには，発生元（ソース）が記録されます。このソース名は，実際にログレコード自体に含まれます。詳細なフィルタリングメカニズムが用意されており，ロガーマネージャによるグローバルスイッチに加えて，ソース名に基づくフィルタリングを行うことができます。プロパティを使用して，特定のソース名やソース名のコンテキスト内の詳細レベルに基づくログメッセージをアペンダに転送するようにロガー上にフィルタを設定できます。フィルタのこれらの属性を設定するために，上のプロパティを使用できます。「ln」はロガー名を表します。初めに，最初のプロパティを使用して，制御するすべてのソース名をフィルタに登録します。次に，2 番めと 3 番めのプロパティを使用して，各ソース名の設定を細かく調整します。「sn」は，最初のプロパティでコンマ区切りのソース名として登録されているソース名を表します。特別なソース名「all」は，上のプロパティを使用して設定されていないすべてのソース名を表します。

プロパティの設定

上のプロパティは，これらのプロパティを含むプロパティファイル（環境変数 VDLOG_PROP_FILE でポイントされる）を使用して，ロガーフレームワークに導入できます。

ORB とすべての ORB サービスは，「default」という名前のデフォルトロガーを使用します。したがって，「-D」コマンドラインパラメータを使用するか，「-ORBpropStorage」コマンドラインパラメータでポイントされるプロパティファイルを使用して上のプロパティが導入された場合，ORB は，VisiBroker for C++ プロパティマネージャを使用して，デフォルトロガーの設定を再度上書きします。

第 37 章

Web サービスの概要

Web サービスは，標準 XML メッセージ通信を使用してネットワーク上で記述，公開，検索，呼び出しを実行するためのアプリケーションコンポーネントです。Web サービスは，SOAP，Web Services Description Language（WSDL），Universal Discovery, Description and Integration（UDDI）などの新しいテクノロジーで定義され，World Wide Web でアクセスできる再利用可能なソフトウェアモジュールから e ビジネスアプリケーションを作成するとともに，古いさまざまなアプリケーションを統合する手段も提供する新しいモデルです。

Web サービスアーキテクチャ

標準 Web サービスアーキテクチャは，Web サービスの公開，検索，バインドを行う 3 つのロールからなります。

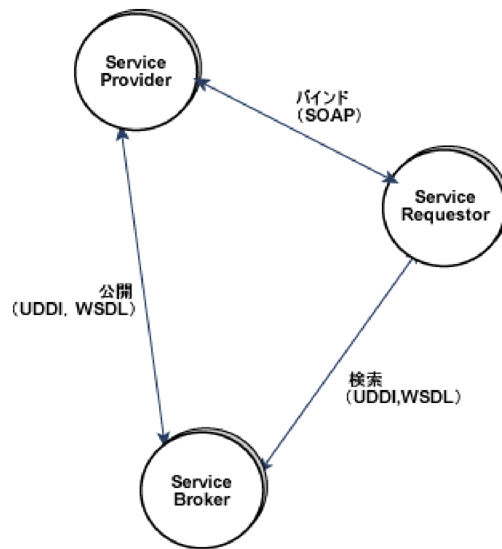
- *Service Provider* は，利用できるすべての Web サービスを Service Broker に登録します
- *Service Broker* は，Service Requestor がアクセスする Web サービスを公開します。公開される情報の内容は，Web サービスとその場所です。また，Web サービスを公開するほかに，Web サービスのホスティングを調整します。
- *Service Requestor* は，Service Broker との対話から Web サービスを検索します。その結果を受けて，Service Requestor は，Web サービスをバインドまたは呼び出します。

Service Provider は Web サービスを処理し，Web 経由でクライアントに提供します。Service Provider は，Web サービス定義とバインド情報を，Universal Description, Discovery，Integration（UDDI）レジストリに公開します。Web Service Description Language（WSDL）ドキュメントには，受信メッセージと返信用の応答メッセージなど，Web サービスに関する情報が収められます。

Service Requestor は，Web サービスを利用するクライアントプログラムです。Service Requestor は，UDDI や電子メールなどの方法で Web サービスを検索します。その後，Web サービスをバインドして呼び出します。

Service Broker は，Service Provider と Service Requestor 間の対話を管理します。Service Broker では，すべてのサービス定義とバインド情報を提供します。現在は，SOAP（分散環境の情報通信向けの XML ベースのメッセージ通信，エンコードプロトコル形式）が Service Requestor と Service Broker 間の通信標準となっています。

標準 Web サービスアーキテクチャ



VisiBroker Web サービスアーキテクチャ

このアーキテクチャには，次の 2 つの側面があります。

WSDL を使用して，Service Requestor が呼び出しを行うための CORBA インターフェースを公開する。

Service Requestor が SOAP/HTTP を通じて CORBA オブジェクトにアクセスできるようにするための実行時環境を提供する。これには，Services Provider と Service Broker をサポートするインフラストラクチャが含まれます。

第 1 の側面は，OMG の「CORBA to WSDL/SOAP Inter-working specification」（CORBA および WSDL/SOAP 間相互作用仕様）で指定された標準にしたがって IDL インターフェースを WSDL ドキュメントに変換する Web サービス開発ツールを使用することで解決できます。呼び出しを行う Service Requestor または Web サービスクライアントは，SOAP over HTTP をトランスポートとして使用して，生成された WSDL を使用できます。

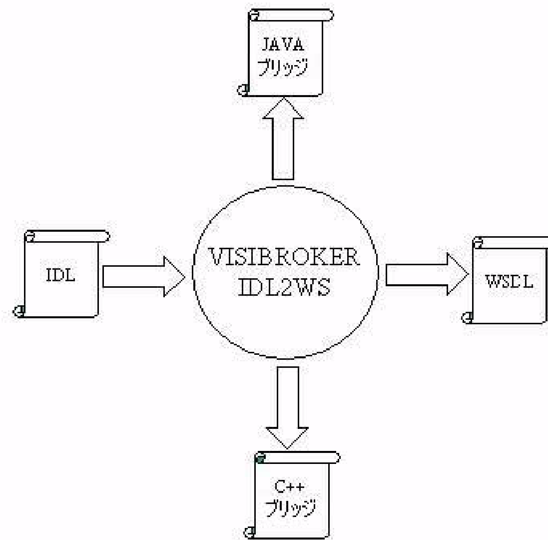
Web サービスランタイムを提供するために，VisiBroker は，Apache Axis テクノロジーを使用して Service Broker の複雑な部分を処理します。独自の型固有ブリッジ（ツールが生成）を使用して，配布されたステートレス CORBA オブジェクトにアクセスできます。型固有ブリッジインスタンスは，CORBA オブジェクトバックエンドの機能を Service Requestor に公開する Service Provider として動作します。

Web サービス関連ファイル

次の図は，IDL ファイルから WSDL ドキュメントとブリッジコードを生成する VisiBroker 付属の Web サービス開発ツールを示します。WSDL ドキュメントは Service Requestor によって使用されます。また，サービス記述とともに，SOAP 準拠クライアントが呼び出しを行うために使用する SOAP バインド情報も提供します。

実際に生成されるブリッジ関連ファイルは，Service Broker（Axis ランタイム）として配布される言語／型固有のサービスプロバイダコンポーネントです。このインスタンスは，

Service Requestor からの着信 SOAP メッセージをバインドされた CORBA オブジェクトに適用する役割を持ちます。



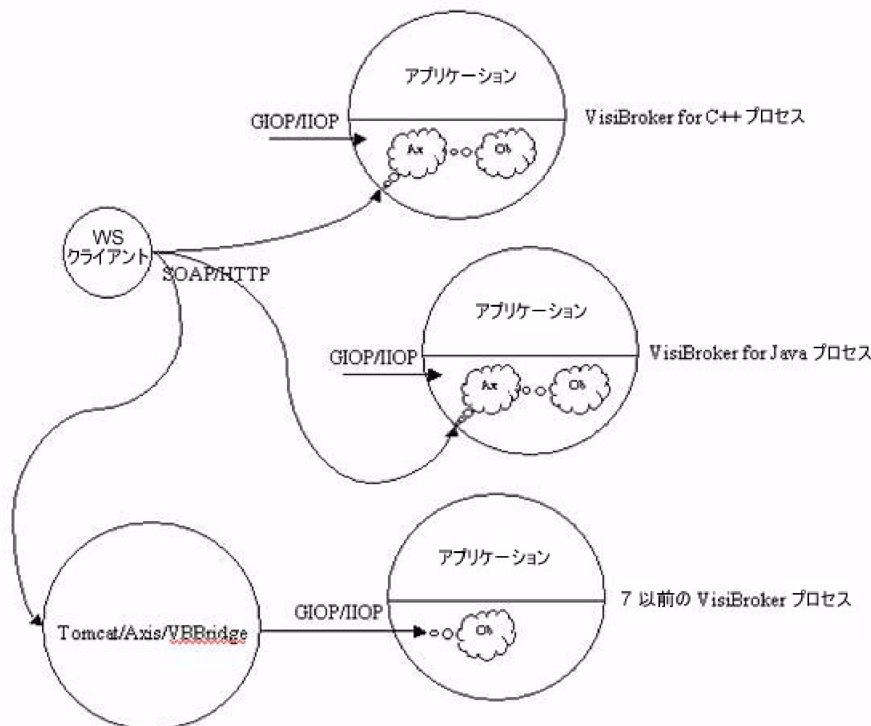
Web サービスランタイム

実行時の動作を説明するため，次の図は，VisiBroker for C++，Java，および Pre 7.0 VisiBroker プロセス内で，生成された WSDL を使用して，Web サービスとして公開された 3 つの CORBA オブジェクトに SOAP/HTTP 呼び出しを行う SOAP クライアントを示します。

VisiBroker プロセスには，HTTP/SOAP リスナー（内部的には Apache Axis テクノロジー）のインフラストラクチャが付属し，これはデフォルトでオフになっています。コマンドラインプロパティ vbroker.ws.enable=true を設定することで，この実行時インフラストラクチャを開始できます。インフラストラクチャが開始されたら，Axis の WSDD メカニズムを使用して，CORBA オブジェクトの Service Provider（ブリッジ）を配布できます。CORBA オブジェクトのバインドに関連する VisiBroker 独自の WSDD 要素を使用して，配布されたブリッジインスタンスを CORBA オブジェクトにバインドできます。このブリッジ上の SOAP 呼び出しは，インプロセス CORBA 呼び出しに変換されます。実際のブリッジは，Axis のサーバー側生成コードの一変形で，各 Web サービスインプリメンテーションスケルトンが型固有の CORBA オブジェクトスタブのメソッドにマップされています。ブリッジは IDL から直接生成されるため，完全にタイプセーフで忠実な IDL タイプが組み込まれています。また，ブリッジは CORBA オブジェクトと同じプロセスでロードされるため，ブリッジから CORBA オブジェクトへのすべての呼び出しは，VisiBroker の「インプロセス」ビッドのために最適化されます。

図の中の「Ax」部分は，VisiBroker プロセスにロードされた Axis + HTTP リスナーコンポーネントを示します。「Ob」部分は，ORB 内部の CORBA オブジェクトを示します。「Ax」と「Ob」の間の 2 つの小さな円で示された両者の関連は，CORBA オブジェクトを Service Requestor に公開する Axis ランタイムでのブリッジの配布を示します。既存の

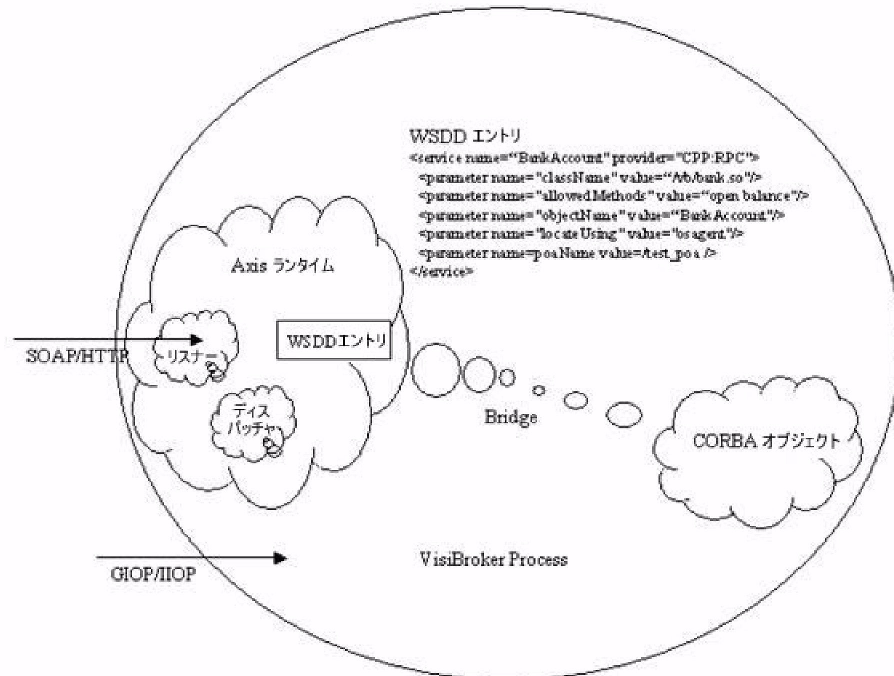
CORBA クライアントは，何の影響もなく通常どおり，GIOP/IIOP リスナーを通じて引き続き GIOP over IIOP 呼び出しを行うことができます。



Pre 7.0 VisiBroker 配布での CORBA オブジェクトの公開をサポートするために，VisiBroker プロセスの外部で実行されている Axis インスタンスにブリッジを配布できます。この場合の唯一の違いは，SOAP から GIOP への適用がリモートであり，したがってネットワークを介して行われることです。上の図では，これは，Apache Tomcat に埋め込まれた Axis for Java にブリッジを配布することで示されています。「Ob」部分は，リモートの Pre 7 VisiBroker プロセスで実行されている CORBA オブジェクトインスタンスを示し，ブリッジからの要求が GIOP/IIOP エンドポイントを通じて着信します。

次の図は，VisiBroker プロセス内部のコンポーネントを示します。「Axis Runtime」部分には，Axis ランタイム，HTTP リスナー，および SOAP 要求ディスパッチャが含まれます。プロセス内部の CORBA オブジェクトは，Axis WSDD メカニズムを使用してサービスプロバイダまたはブリッジを Web サービスとして配布することで，Web サービスとして公開されます。SOAP クライアントが Web サービスに対して呼び出しを行うと，HTTP リスナーが SOAP 要求を取り出し，要求がディスパッチャに渡されます。ディスパッチャは，Axis ランタイムを呼び出して SOAP 要求を渡します。Axis ランタイムは SOAP 要求をデコードし，配布されている Service Provider（ブリッジ）のインスタンスに対して

呼び出しを行います。ブリッジは，WSDD として提供されるバインド情報を使用して，実際の CORBA オブジェクトにバインドし，CORBA 呼び出しを行います。



上のコンテキストでは，Service Broker には HTTP トランスポートの SOAP ノードだけが含まれます。Web サービス配布に必要な UDDI サービスなどの他のサービスは提供されていません。これらのさまざまなインプリメンテーションがあり，簡単に使用できます。

Web サービスとしての CORBA オブジェクトの公開

VisiBroker for C++ で CORBA オブジェクトを Web サービスとして公開するには，次の手順にしたがう必要があります。

開発

1. IDL ファイルから IDL インターフェースの WSDL ドキュメントを生成する
2. IDL ファイルからインターフェース型固有の C++ ブリッジを生成する
3. ブリッジを共有ライブラリにビルドする

配布

1. Web サービスランタイムを有効化／設定する
2. Axis WSDD メカニズムを使用して，VisiBroker プロセスでブリッジ共有ライブラリを配布する

ここでは，examples ディレクトリの vbroker/ws/bank サブディレクトリに用意されている例を説明します。この例は，vbroker/basic/bank_agent の例を変更したもので，Account と AccountManager という 2 つのインターフェースで構成されます。AccountManager では，新しい名前の口座を作成できます。特定の名前の口座がすでに存在する場合は，新しい口座を作成しないで，その口座が取得されます。Account インターフェースは，口座の残高を照会できます。サーバーは，ルート POA の下に POA を設定し，AccountManager インターフェースを実装するオブジェクトを起動します。このオブジェクトの open オペレーションを実行すると，Account インターフェースを実装する別のオブジェクトが作成および保存されて，返されます。次のサンプルコードは，この 2 つのインターフェースを示します。

```
// Bank.idl
module Bank {  interface Account {
    float balance();
  };
  interface AccountManager {
    Account open(in string name);
  };
};
```

この例では，このステートフルアプリケーションを拡張し，Web サービスを使用して SOA をサポートする方法を示します。開発の最初の手順として，ステートフルオペレーションを SOA に適した粗い抽象オペレーションに変換する必要があります。次に示すインターフェースはその 1 例です。このインターフェースは，指定された口座がまだ存在しない場合は口座を開き，その口座の残高を返す 1 つのオペレーションをサポートします。

```
// BankWebService.idl

module BankWebService {

  interface AccountManagerWebService {

    // まだ開いていない場合は口座を開き，残高を返します

    float openAndQueryBalance(in string name);

  };

};
```

次に，このインターフェースを実装する CORBA オブジェクトを実装します。これは，Account および AccountManager インターフェースを内部的に使用し，既知のオブジェクト ID を使用して既知の POA で起動されます。

サーバーをステートレスオペレーション用に拡張したら，次のセクションで示すように，Web サービスのサポートを実装できます。

開発

IDL からの WSDL の生成

idl2wsc コンパイラ（Windows では idl2wsc.exe）は，OMG の「CORBA to WSDL/SOAP Inter-working specification」にしたがって，IDL ファイルの WSDL ドキュメントを生成します。次のように，BankWebService.idl に対してコンパイラを実行すると，BankWebService.wsdl という名前の WSDL ドキュメントが生成されます。この WSDL ドキュメントは，外部的な手段によって潜在的な Web サービスクライアントやクライアント開発チームに公開できます。

```
prompt> idl2wsc BankWebService.idl
```

C++ インターフェース型固有のブリッジの生成

-gen_cpp_bridge オプションを付けて idl2wsc コンパイラを使用すると，特定のインターフェース型の C++ ブリッジを生成することもできます。次のコマンドは，BankWebService_ws_s.cpp および BankWebService_ws_s.hh という名前のファイルにブリッジコードを生成します。このコードはアプリケーションにとって不透過であり，変更することはできません。

```
prompt> idl2wsc -src_suffix cpp -gen_cpp_bridge BankWebService.idl
```

上の 2 つのコマンドは組み合わせることができるので注意してください。

生成された C++ ブリッジを Web サービスとして配布するには，BankWebService.idl のスタブコードとリンクされて共有ライブラリとしてパッケージする必要があります。

使用できるオプションのリストについては，「C++ 対応プログラマツール」の章の idl2wsc のセクションを参照してください。

配布

配布 WSDD の作成

配布の最初の手順は，ブリッジまたはサービスプロバイダの Axis WSDD ドキュメントを編集することです。WSDD または Web サービス配布ディスクリプタは，配布関連情報を指示するための Axis の標準の方法です。ブリッジのテンプレート WSDD は，ブリッジの作成時に作成されます。次に示すサンプル WSDD は，「bank_agent_poa」という名前の POA で，オブジェクト ID が「BankManagerWebService」である CORBA オブジェクトによってホストされる Web サービスを配布します。このオブジェクトへのオブジェクトリファレンスは，osagent を使用してバインドされます。

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment
    <service
name="BankWebService.AccountManagerWebServiceService"
        provider="CPP:RPC"
        description="VisiBroker C++ web service">
        <parameter
            name="className"
            value="[PATH]/libbridge.so"/>
        <parameter
            name="allowedMethods"
            value="openAndQueryBalance"/>
        <parameter
            name="objectName"
          value="BankManagerWebService"/>
       <parameter
            name="locateUsing"
            value="osagent"/>
        <parameter
           name=poaName
               value=/bank_agent_poa/>
    </service>
</deployment>
```

作成された WSDD を使用して配布します。

初期化時に，Axis C++ は，axiscpp.conf という名前の設定ファイルを読み取ります。このファイルは $AXISCPP_DEPLOY/etc（UNIX）または %AXISCPP_DEPLOY%（Windows）にあり，このファイルを使用して，ユーザーは使用するパーサーライブラリや配布ディスクリプタファイルの場所などの環境設定を指定できます。

この設定ファイルでは，WSDD を検索する場所と配布されるサービスを Axis が認識できるように，WSDDFilePath を定義する必要があります。XMLParser は，VisiBroker に付属するパーサーとは異なるパーサーを使用する場合にのみ必要です。VisiBroker WSRT には，独自のトランスポートインプリメンテーションがあるため，トランスポートとチャネルの設定は使用されません。

axiscpp.conf ファイルのサンプル

```
# コメント文字は '#'
#
# WSDDFilePath：WSDD のパス
# XMLParser：Axis XML パーサーライブラリ

WSDDFilePath: /usr/local/VisiBroker/etc/server.wsdd
```

サービスを配布する方法は 2 つあります。

- WSDD ファイルを修正して手動でサービスを追加します。
- Axis C++ のツール AdminClient を使用します。

メモ　AdminClient は Axis C++ クライアント側ライブラリに依存していますが，このライブラリは VisiBroker for C++ に付属しないため，このツールを使用するには，ツールと必須ライブラリを Apache Axis から入手する必要があります。また，AdminClient は Axis C++ 1.5 では使用できません。

Web サービスランタイムの設定

Web サービスランタイムを設定するには，プロパティファイル server.prop を作成します。サンプルのプロパティファイルを次に示します。次のプロパティは，ホスト 143.186.141.54，ポート 19000 で HTTP サーバーを開始するように Service Broker を設定します。接続マネージャは，最大 30 の同時接続を許可し，接続アイドル時間を 300 秒とするように設定されます。着信 SOAP 要求にサービスを提供するスレッドプールは，最大 300 のスレッドを持ち，スレッドアイドル時間を 300 秒とするように設定されます。設定できるプロパティの一覧については，「VisiBroker のプロパティ」の章の「Web サービスランタイムのプロパティ」セクションを参照してください。

```
vbroker.ws.enable=true
vbroker.ws.listener.host=143.186.141.54
vbroker.ws.listener.port=19000
vbroke.ws.keepAliveConnection=true
vbroker.ws.connectionMax=30
vbroker.ws.connectionMaxIdle=300
vbroker.ws.dispatcher.threadMin=0
vbroker.ws.dispatcher.threadMax=300
vbroker.ws.dispatcher.threadMaxIdle=300
```

次のようにサーバーを実行します。

```
prompt> Server -ORBpropStorage server.prop
```

WSDD リファレンス

WSDD の詳細は http://www.oio.de/axis-wsdd/ または http://ws.apache.org/axis/java/reference.html#DeploymentWSDDReference を参照してください。

VisiBroker によって使用されるパラメータを次に示します。

- className - このサービスに要求が到達するときに，Axis サーバーエンジンによってロードされる共有（ダイナミックリンク）ライブラリの名前。通常，これは IDL に基づくインターフェース名です。
- allowedMethods - このクラスに対して呼び出すことができるメソッド。CORBA オブジェクトは，ここに示されるメソッド以外のメソッドを持ってもかまいません。ここに示されるメソッドは Web サービスで使用できます。
- objectName - オブジェクトの名前。これは，必須パラメータです。
- locateUsing - このパラメータは，このプロバイダでオブジェクトを検索するときのメカニズムを指定します。次の 3 つの値があります。

`osagent` - オブジェクトは osagent にあるとします。オブジェクトは，`bind()` メソッドで検索します。`poaName` も指定すると，`objectName` の検索範囲が，その POA の下になります。このパラメータのデフォルト値です。

`nameservice` - このオブジェクトはネーミングサービスにあるとします。オブジェクトは，ルートコンテキストの `resolve()` メソッドで検索します。objectName は，ルートコンテキストを起点とする絶対パスで指定したオブジェクトの名前です。

ior - 指定する objectName は，IOR であるとみなします。オブジェクトは，ORB の string_to_object() メソッドで取得します。IOR には標準形式を使用できます。たとえば，次のようになります。

```
corbaname::xxx
IOR:xxx
corbaloc::xxx
```

WSDD のサービス要素のサンプル

```
<service
    name="AServiceBankWebService.AccountManagerWebServiceService"
    provider="CPP:RPC " description="VisiBroker C++ web service">
    <parameter name="className"
        value="/usr/local/VisiBroker/servies/libaccount_manager.so"/>
        <parameter name="allowedMethods" value="openAndQueryBalance"/>
        <parameter name="objectName" value="BankManagerWebService" />
        <parameter name="locateUsing" value="osagent" />
        <parameter name="poaName" value="/bank_agent_poa">
    </parameter>
</service>
```

制限

Axis にあるいくつかの制限のため，現在のバージョンには次の制限が適用されます。

- IDL ファイルは，インターフェース定義を 1 つだけ持つことができます。これは，現在，Axis WSDL2WS ツールが WSDL 内の複数のポートタイプをサポートしていないためです。
- 各ブリッジがそれぞれ異なる共有ライブラリにバンドルされる必要があります。

SOAP/WSDL の互換性

SOAP バージョン 1.1 および WSDL バージョン 1.1 がサポートされています。

索引

記号

#pragma メカニズム 272
*_interface_name() メソッド 139
*_repository_id() メソッド 139
*object_to_string() メソッド 139
... 省略符 4
[ ] ブラケット 4
_duplicate() メソッド 137
_get_policy 141
_is_a() メソッド 139
_is_bound() メソッド 140
_is_local() メソッド 140
_is_remote() メソッド 140
_narrow() メソッド 83
_nil() メソッド 137
_ptr
  idl2cpp コンパイラによって生成 149
_ref_count() メソッド 138
_release() メソッド 138
_set_policy_override メソッド 141
_tie クラス 131
  idl2cpp コンパイラによって生成 150
  サンプル 132, 133
  デリゲータインプリメンテーション 131
  テンプレートクラス 132
_var クラス
  idl2cpp コンパイラによって生成 149
| 縦線 4

A

account.idl
  account_c.cc から生成されるファイル 17
  account_c.hh から生成されるファイル 17
  account_s.cc から生成されるファイル 17
  account_s.hh から生成されるファイル 17
AccountManager インターフェース
  DSI 309
activate() メソッド 403
Activator クラス
  ORB オブジェクトの非アクティブ化 403
  オブジェクトのアクティブ化の遅延 403, 405
ActiveObjectLifeCycleInterceptor 341
  クラス 340
Agent
  レポート 165
Agent インターフェース 169
agentaddr ファイル
  IP アドレスの指定 162
Any
  クラス 296
Any オブジェクト 290
Any 型
  DSI 309

B

backingStoreType 62
BAD_CONTEXT 例外 433
BAD_INV_ORDER 例外 433
BAD_OPERATION 例外 433
BAD_PARAM 例外 433
BAD_TYPECODE 例外 433
BiDirectional ポリシー 398
bind
  nsutil 185
  共通オブジェクトリファレンス 292
  プロセス 136
bind()
  osagent 157
bind_context
  nsutil 185
bind_new_context
  nsutil 185
BindInterceptor
  クラス 340
BOA
  VisiBroker における ... の使い方 401
  オブジェクトアクティベータ 401, 403
  下位互換サポート 403
  コードのコンパイル 401
  削除されたクラス 401
  サポートされるオプション 401
  ネーミングオブジェクト 402
  バインディング 165
BOA のオプション 401
BOA_init
  パッケージの変更 401
Borland Web サイト 4, 5
Borland 開発者サポート，連絡 4
Borland テクニカルサポート，連絡 4

C

C++
  compiler 26
  クラス 26
C++ コードの生成 26
ChainUntypedObjectWrapper 359
ClientRequestInterceptor 340
  クラス 315, 340
  実装 326
ClusterManager 199
ClusterManager インターフェース 201
Codec 318
  インターフェース 318
  クラス 318
CodecFactory 318
  インターフェース 318
  クラス 318
COMM_FAILURE 例外 433
ConnEventListener インターフェース 371
connID 371
ConnInfo 371
  connID 371
  ipaddress 371
  ポート 371
container
  サーバーマネージャ 234
Container クラス 235
CORBA
  C++ 言語マッピング仕様 33
  VisiBroker 準拠 11
  概要 7
  共通オブジェクトリクエストブローカーアーキテクチャ 7
  定義 7, 131
CORBA 例外 433

-corba_inc 引数 26
corbaloc URL 187
corbaname URL 187
CosNaming
  コマンドラインからの呼び出し 185
CosNaming 操作
  VisiNaming によるサポート 185
CreationImplDef クラス 276
  activation_policy プロパティ 276
  args プロパティ 276
  env プロパティ 276
  path_name プロパティ 276
CreationImplDef 構造体
  オブジェクトのアクティブ化 277
Current
  インターフェース 318
custom valuetype 392

D

-D_VIS_INCLUDE_IR フラグ 287
DATA_CONVERSION 例外 433
DataExpress アダプタ 193
deactivate() メソッド 403
destroy
  nsutil 185
DII 10
  _request メソッドの使い方 294
  Any オブジェクト 290
  create_request メソッドの使い方 294
  DII 要求の作成 294
  DII 要求の初期化 295
  NamedValue インターフェース 296
  NamedValue クラス 296
  NVList オブジェクト 291
  Reply の受信オプション 290
  Request オブジェクト 290
  Request オブジェクトの使用 290
  Request クラス 293
  Request の送信オプション 290
  send_deferred メソッド 299
  send_oneway メソッド 299
  Typecode オブジェクト 290
  インターフェースリポジトリ 281, 301
  応答の受信 292
  概念 290
  概要 289
  機能の概要 10
  共通オブジェクトリファレンス 292
  クライアント 292
  クライアントの構築 289
  結果の受信 298
  欠点 289
  コンテキストの設定 295
  サンプル 292
  非同期要求 299
  複数の要求の受信 299, 300
  複数の要求の送信 299
  要求の作成 293
  要求の初期化 293
  要求の送信 291, 298
  要求の引数の設定 295
DSI
  AccountManager インターフェース 309
  Any 型 309
  BAD_OPERATION 例外 309
  DSI での入力処理 309
  DynamicImplementation クラスから派生 306
  ServerRequest クラス 308
  オブジェクトインプリメンテーションの動的な作成 306
  オブジェクトサーバーのコンパイル 306
  オブジェクトのアクティブ化 310
  オブジェクトの動的な作成 306
  概要 305
  クラスの派生 306
  サーバーオブジェクトの実装 309
  サンプル 306
  スコープ解決演算子 308
  入力パラメータ 309
  プロトコル間ブリッジ 305
  プロトコルのブリッジ 305
  戻り値 309
-DSTRICT プリプロセッサオプション 26
DynamicImplementation クラス 306
  … から派生するサンプル 306
DynAny
  アクセスと初期化 378
  値の初期化とアクセス 378
  概要 377
  作成 378
  タイプ 377
DynAny インターフェース 377
  current_component メソッド 379
  DynAnyFactory オブジェクト 378
  DynArray データ型 379
  DynEnum インターフェース 379
  DynSequence データ型 379
  DynStruct インターフェース 379, 380
  DynUnion インターフェース 379
  NameValuePair 380
  next メソッド 379
  rewind メソッド 379
  seek メソッド 379
  to_any メソッド 380
  構造データ型 379
  サンプル 377
  サンプル IDL 380
  サンプルアプリケーション 380
  サンプルクライアントアプリケーション 380
  サンプルサーバーアプリケーション 381
  制限 378
DynArray データ型 379
DynEnum インターフェース 379
DynSequence データ型 379
DynStruct インターフェース 379
DynUnion インターフェース 379

E

enableBiDir プロパティ 395
EventChannel 220
EventListener 371
  接続の実装 373
  登録 373
EventQueueManager インターフェース 371
-export 引数 26
-export_skel 引数 26
exportBiDi プロパティ 395

F

factory_name 185
FREE_MEM 例外 433

G

get_listeners 371

H

-hdr_suffix 引数 26

I

id フィールド
  NameComponent 182
IDL
  DynAny サンプル 380
  idl2cpp によって生成されるクライアントコード 148
  IR に含まれる情報 281
  Java へのマッピング 13, 17
  OAD インターフェース 279
  一方向メソッドの定義 151
  インターフェースの継承 152
  インターフェースリポジトリで表現される構造 282
  オブジェクトの指定 17
  から C++ 言語へのマッピング 33
  共用体 40
  コンパイラ 17
  サーバーマネージャ 234
  仕様のサンプル 148
  配列 44
  プリミティブデータ型 33
IDL 型
  valuetype 46
IDL コンパイラ 26, 29
IDL ファイル
  #pragma メカニズム 272
idl2cpp コンパイラ 17
  _op1 メソッド 150
  -corba_inc 26
  -export 26
  -export_skel 26
  -hdr_suffix 26
  -no_excep_spec 26
  op1 メソッド 148
  -type_code_info 26
  -version 26
  一方向メソッドの定義 151
  インターフェースの継承 152
  コードの生成 147
  生成される _ptr 149
  生成される _tie 150
  生成される _var 149
  生成されるクライアントコード 148
  属性メソッド 151
idl2cpp ツール 26
idl2ir
  コマンド情報 29, 30
  説明 29, 30
idl2ir コンパイラ 284
  コマンド情報 12
  説明 12
idl2ir ツール 29
IIOP
  双方向 ... サンプル 396, 398
  双方向 ... の使用 395
  双方向 ... の有効化 397
IMP_LIMIT 例外 433
impl_rep ファイル 269
importBiDi プロパティ 395
importBiDir 399
INITIALIZE 例外 433
Interceptor
  インターフェース 314
  クラス 314
InterfaceDef オブジェクト
  インターフェースリポジトリ 282
INTERNAL 例外 433
INTF_REPOS 例外 433
INV_FLAG 例外 433
INV_INDENT 例外 433
INV_OBJREF 例外 433
INVALID_TRANSACTION 例外 433
InvalidPolicy 例外 398
invoke() メソッド 305, 306
  実装のサンプル 306
IOR インターセプタ 313
IORCreationInterceptor 342
  クラス 340
IORInfoExt
  クラス 321
IORInterceptor
  インターフェース 317
  クラス 317
IP サブネットマスク
  localaddr ファイル 161
  スコープを指定するブロードキャストメッセージ 159
ipaddress 371
IR
  オブジェクト情報へのアクセス 286
  オブジェクトの識別 285
  継承元のインターフェース 286
  構造体 284
  サンプル 287
  説明 281
  内容 285
  保存できるオブジェクトの型 285
IR → インターフェースリポジトリ 10
ir2idl 30
  オプション 30
ir2idl ユーティリティ
  IR の内容の表示 283
irep ツール
  インターフェースリポジトリの作成 282, 283
  インターフェースリポジトリの表示 283
is_nil() メソッド 137

J

JDBC アダプタ 193
JDBC アダプタのプロパティ 62
jdbcDriver 62

K

kind フィールド
  NameComponent 182

L

list
  nsutil 185
localaddr ファイル
  インターフェースの用法の指定 161
loginPwd 62

M

MARSHAL 例外 433

maxQueueLength 231
method 141
minor() メソッド 83
ModuleDef オブジェクト
  インターフェースリポジトリ 282

N

name
  解像度 182
  定義 182
NameComponent
  id フィールド 182
  kind フィールド 182
  定義 182
NamedValue
  オブジェクト 295
NamingContext
  ファクトリ 181
  ブートストラップ 181
NamingContextExt 189
NamingContexts
  オブジェクトインプリメンテーションによる使用 181
  クライアントアプリケーションによる使用 181
  定義 181
new_context
  nsutil 185
nil リファレンス
  取得 137
  … のチェック 137
nmake
  … でコンパイル 21
nmake コンパイラ 21
-no_excep_spec 引数 26
NO_IMPLEMENT 例外 433
NO_MEMORY 例外 433
NO_PERMISSION 例外 433
NO_RESOURCES 例外 433
NO_RESPONSE 例外 433
nsutil 185
  bind 185
  bind_context 185
  bind_new_context 185
  destroy 185
  list 185
  new_context 185
  rebind 185
  rebind_context 185
  resolve 185
  shutdown 185
  unbind 185
null valuetype 388
null セマンティクス 391
NVList オブジェクト 291
NVList クラス 309
  ARG_IN パラメータ 309
  ARG_INOUT パラメータ 309
  ARG_OUT パラメータ 309
  引数リストの実装 295

O

OAD
  IDL のインターフェース 279
  impl_rep ファイル 269
  oadutil list 272
  osagent 156
  アクティブ化ポリシーの設定 277
  インターフェース名 271
  インプリメンテーションリポジトリ 269
  オブジェクトの登録 273, 277
  オブジェクトの登録解除 277
  オブジェクトのリスト 272
  概要 270
  起動 270
  スマートエージェント 156
  タイムアウトの指定 270
  登録されたオブジェクトの移行 164
  登録されたオブジェクトの複製 163
  登録情報 269
  登録情報の保存 272
  とスマートエージェント 270
  プログラミングインターフェース 279
  プロパティ 65
  リポジトリ ID 271
  渡される引数 277
OAD コマンド
  環境変数の設定 270
oadj
  レポート 165
oadutil
  OAD に登録されたオブジェクトの一覧表示 272
  インプリメンテーションの登録解除 278
oadutil list 272
oadutil ツール
  インプリメンテーションリポジトリの内容の表示 279
  オブジェクトインプリメンテーションの登録 269
OBJ_ADAPTOR 例外 433
OBJECT_NOT_EXIST 例外 433
ObjectWrapper 363
OMG 7
  イベントサービス 217
  コモンオブジェクトサービス仕様 219
  通知サービス 217
open() メソッド 309
OpenLDAP 197
OperationDef オブジェクト
  インターフェースリポジトリ 282
ORB
  resolve_initial_references 186
  インターセプタとオブジェクトラッパーを使ったカスタマイズ 11
  オブジェクトインプリメンテーション 272
  オブジェクトへのバインド 136
  機能 7
  初期化 87, 135
  相互運用性 12
  定義 165
  ドメイン 158
  バインド処理中のオブジェクトへの接続 136
  プロキシの作成 165
  プロパティ 53
orb.lib 26
ORBDefaultInitRef のプロパティ 187
ORBInitializer
  インターフェース 319
  実装 324
  登録 320, 323
ORBInitInfo
  インターフェース 319
  クラス 319
ORBInitRef 185
ORBInitRef のプロパティ 187
ORInfoExt
  インターフェース 321
osagent

bind() 157
オブジェクトの検索 155
オブジェクト名 402
可用性の確認 157
起動 156
クライアントの確認（ハートビート）157
詳細出力 156
スマートエージェント 153
スマートエージェントの起動 21
バインディング 165
別のエージェントの検出 159
無効化 156, 157
レポート 165
osagent ログファイル
オプション 157
OSAGENT_ADDR 環境変数 162
OSAGENT_LOCAL_FILE
環境変数 161
OSAgent（スマートエージェント）
VisiBroker のアーキテクチャ 9
osfind
コマンド情報 165

P

PDF マニュアル 3
PERSIST_STORE 例外 433
PICurrent
クラス 318
POA
BiDirectional ポリシー 398
ObjectID 94
POA の管理 107
POA マネージャ 94, 107
rootPOA 94, 97
servant 94
ServantLocators 105
アクティブオブジェクトマップ 94
アクティブ化 98
アダプタアクティベータ 94, 115
一時的オブジェクト 94
オブジェクトのアクティブ化 98, 102
オブジェクトの非アクティブ化 101
具現化 94
サーバーエンジン 110
サーバー接続マネージャ 112
サーバントの使用 102
サーバントマネージャ 94, 102
作成 87, 95, 97
双方向 IIOP の有効化 398
定義 93
ディスパッチャのプロパティ 109, 113
デフォルトサーバントによるアクティブ化 100
ポリシー 94, 95
要求の処理 116
リスナーのプロパティ 109, 113
リスナーポートプロパティ 114
霊化 94
POALifeCycleInterceptor 341
クラス 340
poolSize 62
ProxyPullConsumer 220
ProxyPullSupplier 220
ProxyPushConsumer 220
ProxyPushSupplier 220
PushConsumer
サンプル 222
実装 228
派生 225
PushConsumer インターフェース 228
PushModel クラス 222
PushSupplier
実装 222
PushSupplier インターフェース 222

Q

QoS 141
インターフェース 141
Quality of Service（Qos）
プロパティ 68

R

rebind
nsutil 185
rebind_context
nsutil 185
rebinds
スマートエージェントで有効 163
ref_data パラメータ 276
register_listener 371
release() メソッド 138
Reply の受信オプション 290
Repository クラス 286
Request オブジェクト 290
DII 290
Request クラス 293
Request の送信オプション 290
RequestInterceptor
実装 326
REQUIRE_AND_TRUST 399
resolve
nsutil 185
root NamingContext 181
rootPOA 97
RoundRobin
VisiNaming サービス 202
ネーミングサービス 202

S

SCM
双方向 IIOP 395
sequence 41
メモリ管理 43
ServerRequest クラス 308
ServerRequestInterceptor 342
インターセプトポイント 316
クラス 340
実装 326
ServiceInit クラス 344
ServiceLoader インターフェース 344
ServiceResolverInterceptor 343
shutdown
nsutil 185
SSL
VisiNaming 208
VisiNaming の設定 208
双方向 IIOP 399
ネーミングサービス 208
ネーミングサービスの設定 208
Storage インターフェース 236
サーバーマネージャ 234
string_alloc 34
string_free 34

string_to_object() メソッド 139
String_var
　クラス 34
SVCnameroot 185
SVCnameroot のプロパティ 187

T

TRANSACTION_MODE 例外 433
TRANSACTION_REQUIRED 例外 433
TRANSACTION_ROLLEDBACK 例外 433
TRANSACTION_UNAVAILABLE 例外 433
TRANSIENT 例外 433
truncatable valuetype 392
-type_code_info 引数 26
Typecode オブジェクト 290
TypeCode クラス 297

U

UDP プロトコル 155
unbind
　nsutil 185
UNKNOWN 例外 433
unregistered_listener 371
UntypedObjectWrapper
　post_method 359
　pre_method 359

V

value 型
　抽象インターフェース 49
valuetype 46, 387
　CustomMarshal インターフェース 392
　Factory クラスの実装 390
　IDL ファイルのコンパイル 389
　isomorphic 388
　marshal メソッド 392
　null 388
　null セマンティクス 391
　truncatable 392
　unmarshal メソッド 392
　valuetype 基底クラスの継承 389
　値ボックス 49
　アンマーシャリング 392
　インプリメンテーションクラス 389
　概要 387
　カスタム 392
　基底クラス 389
　共有セマンティクス 391
　共用 388
　具象 388
　実装 388
　抽象 388
　抽象インターフェース 391
　定義 389
　登録 391
　派生 387
　ファクトリ 387, 388, 391
　ファクトリの実装 390
　ファクトリを ORB に登録 390
　ボックス化 391
　マーシャリング 392
valuetype の実装 388
vbmake
　… でコンパイル 21
vbroker.naming.backingStore 62
vbroker.naming.cache 197
vbroker.naming.enableSlave プロパティ 204
vbroker.naming.jdbcDriver 62
vbroker.naming.loginName 62
vbroker.naming.loginPwd 62
vbroker.naming.poolSize 62
vbroker.naming.propBindOn 202
vbroker.naming.serverAddresses プロパティ 204
vbroker.naming.serverClusterName プロパティ 204
vbroker.naming.serverNames プロパティ 204
vbroker.naming.slaveMode プロパティ 204
vbroker.naming.url 62
vbroker.orb.dynamicLibs プロパティ 344
vbroker.orb.enableBiDir プロパティ 395
vbroker.orb.enableServerManager プロパティ 237
vbroker.security.peerAuthenticationMode 399
vbroker.serverManager.enableOperations プロパティ 237
vbroker.serverManager.enableSetProperty プロパティ 237
vbroker.serverManager.name プロパティ 234
-version 引数 26
VisiBroker
　BOA の下位互換性 401
　CORBA 準拠 11
　機能 9
　サンプルアプリケーション 15
　説明 8
VisiBroker for C++
　ヘッダーファイルスイッチ 25
VisiBroker ORB
　初期化 135
VisiBroker インターセプタ
　サンプル 344
VisiBroker インターセプタ（インターセプタ）339
VisiBroker の機能 9
　IDL コンパイラ 10
　インターフェースリポジトリ（IFR）10
　インプリメンテーションのアクティブ化 9
　インプリメンテーションリポジトリ 10
　オブジェクトデータベースの統合 11
　オブジェクトとインプリメンテーションのアクティブ化 9
　オブジェクトのアクティブ化 9
　コンパイラ , IDL 10
　スマートエージェントのアーキテクチャ 9
　スマートエージェントへの IDL インターフェース 9
　スレッド管理 9
　接続管理 9
　動的起動 10
　マルチスレッド 9
　ロケーションサービス 9
VisiBroker の概要 1
VisiNaming
　OpenLDAP の設定 197
　SSL の使い方 208
　SSL 用のプロパティ（C++）208
　SSL 用のプロパティ（Java）208
　SSL を使用するように設定 208
　キャッシング機能 197
　ブートストラップ 208
　メソッドレベル承認 209
VisiNaming サービス
　nsutil ユーティリティ 185
　アダプタ 195
　インストール 183
　概要 179
　起動 183, 184

クライアント認証 207
クラスタ 199
クラスタの作成 201
サポートされている CosNaming 操作 185
サンプル 210
サンプルプログラム 210
シャットダウン 186
セキュリティ 207
設定 183
デフォルトネーミングコンテキスト 189
取り替え可能なバックストア 193
ブートストラップ 186
フェイルオーバー 203
フォールトトレランス 204
負荷分散 202
プロパティ 58, 190
プロパティファイル 195
マスター／スレーブモード 205
メソッドレベル承認 207
VISObjectWrapper
ChainUntypedObjectWrapper 359
UntypedObjectWrapper 359
VISObjectWrapper::UntypedObjectWrapperFactory 358
Visual C++ nmake コンパイラ 21

W

Web サイト
CORBA 仕様 11
Borland ニュースグループ 5
ボーランド社の更新されたソフトウェア 5
ボーランド社のマニュアル 5
Windows サービス
osagent 156
コンソールモード 156

あ

アクセッサ関数 40
アクティブ化 9
サービスのアクティブ化 404
値ボックス 49
アダプタ
DII 289
VisiNaming サービス 195
ネーミングサービス 195
アプリケーション
オブジェクトインターフェースの定義 17
クライアントプログラムの起動 22
サーバーオブジェクトの起動 21
実行 21
スマートエージェントの起動 21
スレッドプール 120
セッションごとのスレッド 123
双方向 IIOP の有効化 397
配布 22
アプリケーション開発コスト , 削減 7
アプリケーション開発コストの削減 7

い

移行 267, 268
OAD に登録されたオブジェクト 164
インスタンス化されたオブジェクト 164
オブジェクト 164
状態を持つオブジェクト 164
ホスト間のオブジェクト 164
一方向メソッド
定義 151
イベントキュー 371
ConnEventListener インターフェース 371
EventListener インターフェース 371
EventListener の登録 373
EventQueueManager インターフェース 371
イベントタイプ 371
イベントリスナー 371
概要 371
サンプルコード 373
接続 EventListener 373
接続イベント 371
イベントサービス
PushConsumer の派生 225
概要 217
キューの長さの設定 231
コンパイルとリンク 231
サンプル 222
通信モデル 219
プッシュコンシューマの実装 228
プッシュサプライヤの派生 222
プッシュモデル 220
プルモデル 220
イベントタイプ 371
接続タイプ 371
イベントチャネル 221
イベントリスナー 371
ConnInfo 371
インスタンス
オブジェクトリファレンスの … の判定 139
ロケーションサービスを使った検索 167
インターセプタ
ActiveObjectLifeCycleInterceptor 341
API クラス 340
BindInterceptor 340
ClientRequestInterceptor 340
IOR 313
IORCreationInterceptor 342
ORB によるインターセプタの登録 343
ORB のカスタマイズ 11
POALifeCycleInterceptor 341
ServerRequestInterceptor 342
ServiceResolverInterceptor 343
インターセプタオブジェクトの作成 344
インターフェース 340
概要 339
クライアント 340
クライアントインターセプタ 339
クライアント側のポータブルインターセプタ 351
サーバー 341
サーバーインターセプタ 339
サーバー側のポータブルインターセプタ 352
サンプルプログラム 344
使い方 339
データの受け渡し 351
ポータブルインターセプタの使用 351
マネージャ 340
ロード中 344
インターセプタインターフェース
ORB に登録 343
サンプル 344
インターセプタオブジェクト
作成 344
インターセプトポイント
ServerRequestInterceptor 316
要求インターセプトポイント 315, 316
呼び出しの順序 351
インターフェース

Codec 318
CodecFactory 318
ConnEventListeners 371
Current 318
EventListener 371
EventQueueManager 371
IDL での定義 17
Interceptor 314
IORInterceptor 317
NamingContextExt 189
ORBInitializer 319
ORBInitInfo 319
ORInfoExt 321
QoS 141
インターフェースリポジトリの説明 281
継承 152
継承する ... の指定 152
検索 287
属性 151
レポート 165
インターフェース定義言語（IDL）17
インターフェースの継承
指定 152
インターフェース名
OAD からのオブジェクトの登録解除 278
取得 139
定義 148
リポジトリ ID に変換 271
インターフェースリポジトリ 10
_get_interface() メソッド 282
idl2ir による記入 12, 29, 30
idl2ir を使った内容の更新 284
オブジェクト情報へのアクセス 286
オブジェクトの識別 285
機能の概要 10
継承元のインターフェース 286
構造体 284
作成 283
サンプル 287
説明 281
内容 282, 285
内容の表示 283
プロパティ 65
保存できるオブジェクトの型 285
インターフェースリポジトリ（IR）281
インプリメンテーション
OAD からの登録解除 277, 278
アクティブ化 9, 406
サポート 9
ステートレス，メソッドの呼び出し 163
スマートエージェントへの接続 153
セッションごとのスレッドの使い方 123
バインディング 165
フォールトトレランス 163
レポート 165
インプリメンテーションリポジトリ 10
impl_rep ファイル 269
OAD 272
OAD からの登録解除時の削除 277
OAD によるディレクトリの指定 270
OAD の使用 270
オブジェクトの登録解除 278
機能の概要 10
内容の一覧表示 279
保存された登録情報 269
インメモリアダプタ 193

え

永続的オブジェクト
ODA, 機能の概要 11
演算子
スコープ解決 308

お

オーバーライド
ポリシー 141
オブジェクト
CreationImplDef 構造体の使用 277
DSI を使った動的な作成 306
IDL での指定 17
OAD からの登録解除 277
OAD による接続 156
アクティブ化 405, 406
アクティブ化ポリシーの設定 277
インターフェースリポジトリの情報へのアクセス 286
実行可能ファイルのパス 277
ステートレス，メソッドの呼び出し 163
スマートエージェントへの接続 153
登録 276, 277
特性の動的な変更 276
ネットワーク上のオブジェクトのレポート 165
バインディング 165
複数のインスタンス 276
複製 163
無効 408
メソッドを呼び出す状態 163
リスト 272
ロケーションサービスを使った検索 167
オブジェクトアクティベーションデーモン →「OAD」9
オブジェクトアクティベーションデーモン（OAD）156
オブジェクトアクティベータ 403
オブジェクトインプリメンテーション
状態を保持するインプリメンテーション 163
動的な変更 276
フォールトトレランス 163
オブジェクト検索機能
ロケーションサービスによる拡張 9
オブジェクト指向のアプローチ
ソフトウェアコンポーネントの作成 7
オブジェクトデータベースアクティベータ
機能の概要 11
オブジェクトのアクティブ化 9, 276
deferred メソッドサンプル 405
OAD によって渡される引数 277
サービスアクティベータを使った ... の遅延 404
サービスのアクティブ化 404
サポート 9
遅延 403
オブジェクトのアクティブ化の遅延
サービスのアクティブ化 405
オブジェクトの移行 164
オブジェクトの登録
using oadutil 273
変更 276
オブジェクトの登録解除
OAD 277
using oadutil 278
オブジェクトの非アクティブ化 408
オブジェクトマネージメントグループ 7
オブジェクト名
取得 139
... によるバインド先の限定 136
オブジェクトラッパー

idl2cpp requirement 356
ORB のカスタマイズ 11
post_method 357
pre_method 357
概要 355
型付き 356, 361
型付き … の呼び出し順序 362
型付きラッパーと型なしラッパーの併用 366
型付きラッパーの削除 366
型付きラッパーの追加 364
型付きラッパーの派生 363
型なし 356
型なしのインストール 359
型なしの実装 358
型なしの使用 358
型なしファクトリ 358
型なしファクトリの削除 361
共用クライアント／サーバー 363
サンプルアプリケーションの実行 369
サンプルプログラム 356
説明 355
ファクトリの追加 360
複数の型付き … の使用 362
オブジェクトリクエストブローカー→「ORB」7
オブジェクトリファレンス
_is_a() メソッドの使用 139
nil リファレンスの取得 137
nil リファレンスのチェック 137
インスタンスの型の判定 139
インターフェース名の取得 139
永続的 402
オブジェクト名の取得 139
解放 138
型の判定 139
型の変換 140
サブタイプ 139
状態の判定 140
スーパータイプへの変換 141
操作 137
等価のインプリメンテーションのチェック 139
ナローイング 140, 141
場所の判定 140
ハッシュ値の取得 139
複製 137
メモリ管理 149
文字列への変換 139
リファレンスカウントの取得 138
リポジトリ ID の取得 139
ワイドニング 141
オプションと引数 26
オンラインヘルプトピック，アクセス 3

か

下位互換性
イベントサービス 217
開発者サポート，連絡 4
概要 1
VisiNaming サービス 179
型なしオブジェクトラッパー 356
可搬性
サーバー側の配布可能な 10
ガベージコレクション 129
可変長
構造体 39
環境変数
OAD 270
OSAGENT_ADDR 162
OSAGENT_LOCAL_FILE 161
管理コマンド
oadutil list 272
oadutil unreg 278
osfind 165
管理される型
配列 44
完了状態 83
システム例外の取得 83

き

記号
省略符 ... 4
縦線 | 4
ブラケット [ ] 4
起動機能の概要 10
キャスト
システム例外に 83
キャッシング機能 197
キューの長さ
設定値 231
共通オブジェクトのテスタ
DII 289
共通オブジェクトリクエストブローカー→「CORBA」7
共有セマンティクス 391

く

クライアント
DII の使用 292
ORB の初期化 135
サーバーへの一方向接続 398
サーバーへの双方向接続 398
サーバーマネージャの参照 234
実装 18
スレッドプールの使い方 120
セッションごとのスレッドの使い方 123
双方向 IIOP 395
動的起動インターフェースを使った … の構築 289
非同期情報の受信 395
クライアントインターセプタ 339
クライアント側 LIOP 接続
プロパティ 66
クライアント側インプロセス接続
プロパティ 68
クライアントスタブ
生成 17
クライアントとサーバー
実行 21, 23
クライアント認証
ネーミングサービス 207
クライアントリクエストインターセプタ
サンプル 327
クラス
_tie 131, 132, 150
_var 149
ActiveObjectLifeCycleInterceptor 340
Any 296
BindInterceptor 340
ClientRequestInterceptor 315, 340
Codec 318
CodecFactory 318
CreationImplDef 276
DynamicImplementation 306
Interceptor 314
IORCreationInterceptor 340
IORInterceptor 317

NVList 309
NVList ARG_IN パラメータ 309
NVList ARG_INOUT パラメータ 309
NVList ARG_OUT パラメータ 309
ORBInitInfo 319
ORInfoExt 321
PICurrent 318
POALifeCycleInterceptor 340
ServerRequest 308
ServerRequestInterceptor 340
String_var 34
TypeCode 297
ネーミングコンテキスト 188
要求 293
リポジトリ 286
クラスタ 199
ネーミングサーバーで作成 201
クラステンプレート
生成 150
グローバルなスコープを持つオブジェクト
スマートエージェントの登録 153

け

継承
インターフェース 152
インプリメンテーションから 131

こ

構造体
可変長 39
固定長 38
メモリ管理 39
コード
BOA のコンパイル 401
nmake でビルド 21
vbmake でビルド 21
ビルド 20
コード生成 17
固定長
構造体 38
コマンド
idl2ir 29, 30
コマンド，規約 4
コンパイラ
IDL, 機能の概要 10
nmake 21
vbmake 21
コンパイル
BOA コード 401

さ

サーバー
GateKeeper なしのコールバック 395
アクティブ化ポリシーの設定 277
クライアントへの一方向接続 398
クライアントへの接続の開始 395
クライアントへの非同期情報の送信 395
クライアント要求の受け取り 87
クライアント要求の待機 90
サーバーマネージャ 234
実装 19
スレッドにおける留意点 126
セットアップ 87
双方向 IIOP 395
サーバーインターセプタ 339
サーバーエンジン
POA 111
サーバー側サーバーエンジン
プロパティ 69, 74
サーバー側スレッドセッション BOA_TS 接続
プロパティ 70
サーバー側スレッドセッション IIOP_TS 接続
プロパティ 69
サーバー側スレッドプール BOA_TP 接続
プロパティ 72
サーバー側スレッドプール IIOP_TP 接続
プロパティ 70
サーバー側の配布可能な
可搬性 10
サーバーサーバント
生成 17
サーバー接続マネージャ
POA 112
サーバーマネージャ
Container インターフェース 235
Container のメソッド（C++）235
IDL 定義 237
Storage インターフェース 234, 236
アクセス可能性 237
概要 233
カスタムコンテナ 241
コンテナ 234
サンプル 239
はじめに 233
プロパティ 56
有効化 233
リファレンスの取得 234
サーバーマネージャ IDL 234
サーバーリクエストインターセプタ
POA スコープ付き 321
サンプル 327, 332
サービス
ネットワーク上のサービスのレポート 165
サービスアクティベータ
実装 406
サービスのアクティブ化
オブジェクトのアクティブ化の遅延 404
サービスアクティブ化オブジェクトの非アクティブ化 408
サービスアクティベータの実装 406
サンプル 405
... の遅延の実装 405
作業スレッド 118
作成
ソフトウェアコンポーネント 7
サブネットマスク 159, 161
サプライヤ
EventChannel への接続 221
サプライヤ／コンシューマ通信モデル 217
サポート
インプリメンテーションとオブジェクトのアクティブ化のサポート 9
サポート，連絡 4
サンプル
_tie クラス 132, 133
DII の使用 292
DSI 306
DynAny IDL 380
IR 287
oadutil unreg ユーティリティ 278
odb 405
VisiBroker インターセプタ 344
VisiNaming サービス 210

アクティブ化 405
インターセプタ 344
インターフェースリポジトリ 287
オブジェクトのアクティブ化 405, 406, 408
オブジェクトのアクティブ化における deferred メソッド 405
オブジェクトラッパー 356
サーバーマネージャ 239
スマートエージェントの localaddr ファイル 161
双方向 IIOP 396
ネーミングサービス 210
プッシュコンシューマ 222
プッシュサプライヤ 222
ポータブルインターセプタ 322
リクエストインターセプタ 327
サンプルアプリケーション
IDL での Account インターフェースの記述 17
VisiBroker の使用 15
アプリケーションの配布 22
オブジェクトインターフェースの定義 17
開発手順 15
クライアントスタブの生成 17
クライアントの実装 18
コンパイル 21
サーバーサーバント 17
サーバーの起動 21
サーバーの実装 19
サンプルの実行 21
サンプルのビルド 20
サンプルアプリケーションの実行
クライアントプログラムの起動 22
サンプルプログラム
VisiNaming サービス 210
ネーミングサービス 210

し

システム例外
BAD_CONTEXT 433
BAD_INV_ORDER 433
BAD_OPERATION 433
BAD_PARAM 433
BAD_QOS 433
BAD_TYPECODE 433
COMM_FAILURE 433
CompletionStatus 値 83
CORBA 定義 81
DATA_CONVERSION 433
FREE_MEM 433
IMP_LIMIT 433
INITIALIZE 433
INTERNAL 433
INTF_REPOS 433
INV_FLAG 433
INV_INDENT 433
INV_OBJREF 433
INVALID_TRANSACTION 433
MARSHAL 433
NO_IMPLEMENT 433
NO_MEMORY 433
NO_PERMISSION 433
NO_RESOURCES 433
NO_RESPONSE 433
OBJ_ADAPTOR 433
OBJECT_NOT_EXIST 433
PERSIST_STORE 433
SystemException クラス 81
TRANSACTION_MODE 433
TRANSACTION_REQUIRED 433
TRANSACTION_ROLLEDBACK 433
TRANSACTION_UNAVAILABLE 433
TRANSIENT 433
UNKNOWN 433
完了状態の取得 83
キャッチ 84
処理 83
ポータブルインターセプタ 321
マイナーコードの取得と設定 83
例外のナローイング 84
システム例外の処理 83
実装
サーバー 19
指定
IP アドレス 162
状態完了
システム例外の取得 83
承認
VisiNaming のメソッドレベル 209
VisiNaming メソッドレベル 207
ネーミングサービスのメソッドレベル 207, 209

す

スケルトン 17
スコープ解決演算子 308
スタブ
ルーチン 17
ステートレスオブジェクト，メソッドの呼び出し 163
スマートエージェント
bind() 157
OAD 156, 270
OAD によるオブジェクトへの接続 156
osagent 153
OSAGENT_ADDR 環境変数 162
OSAGENT_LOCAL_FILE ファイル 161
インターフェースの用法の指定 161
オブジェクトの自動再登録 157
オブジェクトのフォールトトレランス 163
オブジェクト名 402
概要 153
可用性 157
起動 156
機能の概要 9
クライアントの確認（ハートビート）157
検索 155
異なるネットワーク上の接続 159
最善の方法 155
削除されるオブジェクト 277
詳細出力 156
通信 155
ネーミングサービスの負荷分散 202
バインディング 165
複数のインスタンスの起動 155
複数のドメインでの実行 158
プロパティ 51
別のエージェントの検出 159
ポイントツーポイント通信
通信 162
ほかのエージェントとの協力 155
マルチホームホスト 160
無効化 156, 157
ロケーションサービス 155, 167
スマートエージェント（OSAgent）
アーキテクチャ 9
スレッド
ガベージコレクション 129

作業スレッド 118, 119, 122
スレッドの使い方 117
スレッドプールポリシー 119
スレッドポリシー 118
セッションごとのスレッドポリシー 122
使い方 117
ディスパッチのポリシーとプロパティ 125
同期ブロックの使用 126
プロパティ 128
マルチスレッド , 機能概要 9
リスナースレッド 118
スレッド管理 9
スレッドプールディスパッチポリシー 125
スレッドポリシー 118

せ

生成
_var クラス 149
String_var クラス 34
製品のバージョン 12, 30
セキュリティ
VisiNaming サービス 207
VisiNaming サービスクライアント認証 207
VisiNaming サービスのメソッドレベル承認 207
双方向 IIOP 399
ネーミングサービス 207
ネーミングサービスクライアント認証 207
ネーミングサービスのメソッドレベル承認 207
セキュリティ（C++）
VisiNaming での有効化 208
ネーミングサービスでの有効化 208
セキュリティ（Java）
VisiNaming での有効化 208
ネーミングサービスでの有効化 208
セッションごとのスレッドのインプリメンテーション 123
セッションごとのスレッドのディスパッチポリシー 126
接続
ガベージコレクション 129
管理 , 機能概要 9
クライアントアプリケーションをオブジェクトに … 7
異なるローカルネットワーク上のスマートエージェント 159
ポイントツーポイント通信 162
接続管理 9, 124
プロパティ 127

そ

相互運用性 12
ORB 相互運用性 12
VisiBroker for C++ との 12
VisiBroker for Java との 12
ほかの ORB 製品との 13
双方向 IIOP 395
InvalidPolicy 例外 398
POA 398
一方向接続 398
既存のアプリケーションでの有効化 397
サンプル 396, 398
セキュリティ 399
双方向 SCM 395, 399
双方向のプロパティ 395
ソフトウェアの更新 5

た

タイプ
Any 309
DynAny 377
IDL のプリミティブデータ型 33
sequence 41
valuetype 46
インスタンスの判定 139
オブジェクトリファレンスの … の判定 139
共用体 40
サブタイプの判定 139
システム例外の判定 83
プリミティブ 33
文字列 34
タイプコード
インターフェースリポジトリで表現される 282
タイプセーフ
配列 44
単純な名前 183

ち

遅延
オブジェクトのアクティブ化 404
抽象 valuetype 388
抽象インターフェース 391

つ

ツール
CORBA サービス 12
idl2cpp 17
idl2ir 12, 29, 30
oadutil 273
oadutil unreg 278
osfind 165
管理 12
プログラミング 12

て

ディスクリミナント 40
ディスパッチのポリシーとプロパティ 125
ディスパッチポリシー
スレッドプール 125
セッションごとのスレッド 126
ディスパッチャのプロパティ 113
データ型
sequence 41
共用体 40
テクニカルサポート，連絡 4
デバッグログのプロパティ 74
デフォルトネーミングコンテキスト
取得 189
デフォルトのファクトリ 391

と

動的起動インターフェース →「DII」289
動的スケルトンインターフェース 10
機能の概要 10
動的スケルトンインターフェース →「DSI」305
登録
OAD インプリメンテーションリポジトリ 269
スマートエージェント 153
ドメイン
複数の … の実行 158
トリガー 169, 171
作成 171
取り替え可能なバックストア

種類 193
設定 194
プロパティファイル 194

な

名前
オブジェクトへの名前のバインド 179
単純 183
複雑 183
文字列化 182
名前空間 179
名前の解決 182
ナローイング
例外をシステム例外に 84

に

ニュースグループ 5
入力 , インターフェースリポジトリ 29
入力パラメータ
DSI での処理 309
認証
VisiNaming クライアント 207
双方向 IIOP 399
ネーミングサービスクライアント 207

ね

ネイティブメッセージング 243
ネーミングコンテキスト
クラス 188
デフォルト 189
ネーミングサービス
SSL の使い方 208
SSL 用のプロパティ（C++）208
SSL 用のプロパティ（Java）208
SSL を使用するように設定 208
アダプタ 195
インストール 183
起動 183, 184
キャッシング機能 197
クライアント認証 207
クラスタ 199
クラスタの作成 201
サポートされている CosNaming 操作 185
サンプル 210
サンプルプログラム 210
シャットダウン 186
セキュリティ 207
セキュリティの有効化（C++）208
セキュリティの有効化（Java）208
設定 183
デフォルトネーミングコンテキスト 189
取り替え可能なバックストア 193
ブートストラップ 186, 208
フェイルオーバー 203
フォールトトレランス 204
負荷分散 202
プロパティ 58, 190
プロパティファイル 195
メソッドレベル承認 207, 209
ネットワーク
オブジェクトとサービスのレポート 165

は

配布
オブジェクトインターフェースの定義 17
説明 22
配列 44
管理される型 44
タイプセーフ 44
メモリ管理 45
配列スライス
多次元配列のパラメータの受け渡し 44
バインディング
ORB のタスク 165
バインドされたオブジェクト
場所と状態の判定 140
バインド処理
_bind() で実行されるアクション 136
オブジェクトへのバインド 136
確立されたオブジェクトへの接続 136
作成されるプロキシオブジェクト 136
場所
オブジェクトリファレンスの ... の判定 140
バックストア 193
パフォーマンスの向上 197
パラメータの受け渡し
多次元配列 44

ひ

引数
-corba_inc 26
-export 26
-export_skel 26
-hdr_suffix 26
-no_excep_spec 26
-type_code_info 26
-version 26
非同期通信 395
ヒューリスティックな例外 437

ふ

ファイル
idl コンパイラによる生成 17
impl_rep 269
localaddr 161
コンパイルによる生成 17
ファイル拡張子 17
ファクトリ 388
valuetype 390
実装 390
デフォルト 391
ファクトリクラス 390
フェイルオーバー
VisiNaming サービス 203
ネーミングサービス 203
フォールトトレランス 9
OAD に登録されたオブジェクトの複製 163
VisiNaming サービス 204
ネーミングサービス 204
負荷分散
VisiNaming サービス 202
ネーミングサービス 202
ホスト間のオブジェクトの移行 164
ロケーションサービスの使用 168
複雑な名前 183
プッシュコンシューマ 220
プッシュサプライヤ 220
サンプル 222
実装 222
派生 222

プッシュモデル 220
ブリッジ
DII 289
プリミティブデータ型 33
プリンシパル
IDL インターフェース 46
プルコンシューマ 220
プルサプライヤ 220
プルモデル 220
ブロードキャストアドレス 161
ブロードキャストメッセージ 155
プロキシオブジェクト
バインディング 165
バインド処理中に作成 136
プロキシコンシューマ 218
プロキシサプライヤ 218
プログラマツール 26
idl2cpp 26
idl2ir 29
ir2idl 30
一般情報 26
プロセス
bind 136
プロパティ
DataExpress アダプタ 62
enableBiDir 395
JDBC アダプタ 62
JNDI アダプタ 62
listener 113
OAD 65
ORB 53
ORBDefaultInitRef 187
ORBInitRef 187
POA ディスパッチャ 109
POA リスナー 109
QoS 68
SVCnameroot 187
vbroker.naming.cache 197
vbroker.naming.enableSlave 204
vbroker.naming.propBindOn 202
vbroker.naming.serverAddresses 204
vbroker.naming.serverClusterName 204
vbroker.naming.serverNames 204
vbroker.naming.slaveMode 204
vbroker.orb.dynamicLibs 344
vbroker.orb.enableBiDir 395
vbroker.orb.enableServerManager 237
vbroker.serverManager.enableOperations 237
vbroker.serverManager.enableSetProperty 237
vbroker.serverManager.name 234
VisiBroker BiDirectional 395
VisiNaming サービス 58, 190
インターフェースリポジトリ 65
クライアント側 LIOP 接続 66
クライアント側インプロセス接続 68
サーバー側サーバーエンジン 69, 74
サーバー側スレッドセッション BOA_TS 接続 70
サーバー側スレッドセッション IIOP_TS 接続 69
サーバー側スレッドプール BOA_TP 接続 72
サーバー側スレッドプール IIOP_TP 接続 70
サーバーマネージャ 56
スマートエージェント 51
スレッド管理 128
接続管理の設定 127
ディスパッチャ 113
デバッグログ 74
ネーミングサービス 58, 190
ロケーションサービス 58
プロパティファイル
VisiNaming サービス 195
分散アプリケーション
開発手順 15

へ

ヘッダーファイル
C++ のスイッチ 25
ヘルプトピック，アクセス 3

ほ

ポインタ型
_ptr 定義 149
ポイントツーポイント通信 162
ポータブルインターセプタ
Current 318
Interceptor 314
IOR インターセプタ 313, 317
PICurrent 318
POA スコープ付きサーバー要求 321
ServerRequestInterceptor 316
インターセプトポイント 316
概要 313
拡張機能 321
作成 318
サンプル 322
システム例外 321
タイプ 313
登録 319
要求インターセプトポイント 315
リクエストインターセプタ 313, 315
ポータブルオブジェクトアダプタ
ポリシー 95
ポータブルオブジェクトアダプタ（POA）
定義 93
ポート番号
listener 114
ボックス化 valuetype 391
ポリシー 141
POA 95
有効な 141
ポリシーオーバーライド 141

ま

マイナーコード
システム例外の取得と設定 83
マッピング
IDL から Java 13
IDL モジュールから C++ 名前空間 37
抽象インターフェース 49
マニュアル 2
.pdf 形式 3
Borland セキュリティガイド 2
VisiBroker for .NET 開発者ガイド 2
VisiBroker for C++ API リファレンス 2
VisiBroker for C++ 開発者ガイド 2
VisiBroker for Java 開発者ガイド 2
VisiBroker GateKeeper ガイド 3
VisiBroker VisiNotify ガイド 2
VisiBroker VisiTelcoLog ガイド 3
VisiBroker VisiTime ガイド 2
VisiBroker VisiTransact ガイド 2
VisiBroker インストールガイド 2
Web 5
Web での更新 3

使用されている表記規則のタイプ 4
使用されているプラットフォームの表記規則 4
ヘルプトピックの表示 3
マルチスレッド 117
機能の概要 9
マルチホームホスト 160
インターフェースの用法の指定 161

み

ミューテータ関数 40

む

無効
サービスアクティブ化オブジェクトインプリメンテーション 408
無効化
スマートエージェント 156

め

メイクファイル
Solaris のサンプル 21
メソッド
*_interface_name() 139
*_object_name() 139
*_repository_id() 139
*object_to_string() 139
_duplicate() 137
_get_policy 141
_is_a() 139
_is_bound() 140
_is_local() 140
_is_remote() 140
_narrow() 83
_nil() 137
_ref_count() 138
_release() 138
_set_policy_override メソッド 141
activate() 403
boa.obj_is_ready() 306
deactivate() 403
invoke() 305, 306
invoke() の実装のサンプル 306
is_nil() 137
minor() 83
release() 138
string_to_object() 139
一方向 ... の定義 151
状態を保持するオブジェクト 163
ステートレスオブジェクト，呼び出し 163
生成 150
メソッドレベル承認
ネーミングサービス 207
メッセージ
ブロードキャスト 155
メモリ管理
オブジェクトリファレンス 149
構造体 39
シーケンス 43
配列 45

も

モジュール
IDL モジュールから C++ 名前空間へのマッピング 37
文字列
オブジェクトリファレンスへの変換 139
タイプ 34
文字列 , 動的な割り当てと解放 34
文字列化
object_to_string() メソッドの使用 139
文字列化された名前 182

ゆ

有効なポリシー 141
ユーザー例外
UserException クラス 85
キャッチするためのオブジェクトの変更 86
定義 85
フィールドの追加 86
フィールドへの追加 86
例外を生成するためのオブジェクトの変更 85
ユーザー例外の生成 85
ユーザー例外へのフィールドの追加 86
ユーティリティ
idl2ir 284
irep 282
osagent 21

よ

様態
オブジェクトリファレンスの ... の判定 140

り

リアルタイム CORBA 拡張 411
リクエストインターセプタ 313, 315
POA スコープ付きサーバー要求 321
ServerRequestInterceptor 316
インターセプトポイント 315, 316
サンプル 327, 332
リスナースレッド 118
リスナーのプロパティ 113
リファレンスカウント 138
インクリメント 137
取得 138
リファレンスデータ 276
リポジトリ ID
取得 139, 271
リンクエラー 26

れ

例外
CORBA 433
CORBA 定義のシステム例外 81
CORBA の概要 81
InvalidPolicy 398
システム
SystemException クラス 81
システム例外にキャスト 83
システム例外にナローイング 84
処理 83
生成 85
ヒューリスティック 437
ユーザー例外のキャッチ 86
ユーザー例外へのフィールドの追加 86
例外の完了状態 83
例外のキャッチ
オブジェクトの変更 86
システム例外 84
ユーザー例外 86

例外を生成するためのオブジェクトの変更 85

ろ

ログのプロパティ , デバッグ 74
ロケーションサービス 167
  Agent インターフェース 169
  エージェントのコンポーネント 169
  機能の概要 9
  高度なオブジェクト検索機能 9
  スマートエージェント 155
  トリガー 169, 171
  プロパティ 58