VisiBroker for Java
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［About］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 19 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 8
VisiBroker のスマートエージェント（osagent）アーキテクチャ 8
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 9
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 10
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
IDL を使用しないインターフェース定義 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
Java 開発環境 12
Java 2 Standard Edition 12
Java 実行時環境 13
GateKeeper の要件 13
Java 対応の Web ブラウザ 13
VisiBroker との相互運用性 13
ほかの ORB 製品との相互運用性 13
IDL から Java へのマッピング 14

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 16
IDL を使った Account インターフェースの記述 . . . 16
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.java 18
AccountManager オブジェクトへのバインド . . . 19
Account オブジェクトの取得 19
残高の取得 19
AccountManagerHelper.java 19
その他のメソッド 19
ステップ 4：サーバーの実装 20
サーバープログラム 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 21
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 22
アプリケーションの配布 22
環境変数 23
サポートサービスの有効性 23
vbj の使用 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 23
Java でのサーバーアプリケーションの実行 24

第 4 章
Java 対応プログラマツール 27
オプション 27
共通オプション 28
idl2ir 28
ir2idl 29
idl2java 29
java2idl 31
java2iiop 32
vbj 34
vbjc 35
クラスパスの指定 36
JVM の指定 36
idl2wsj 37

第 5 章
IDL から Java へのマッピング 39
名前 39
予約名 40
予約語 40
モジュール 40
基本型 41

IDL 拡張型 41
Holder クラス 42
Java の null 44
Boolean 45
Char 45
Octet 45
String 45
WString 45
整数型 45
浮動小数点型 45
Helper クラス 45
定数 46
インターフェース内の定数 47
インターフェース内にない定数 47
構造型 47
列挙体 47
構造体 49
共用体 50
シーケンス 51
配列 52
インターフェース 53
抽象インターフェース 54
ローカルインターフェース 54
パラメータの受け渡し 55
継承を使ったサーバーインプリメンテーション 56
デリゲーションを使ったサーバーインプリメンテーション 56
インターフェースのスコープ 57
例外のマッピング 57
ユーザー定義例外 58
システム例外 58
Any 型のマッピング 59
ネストされた型のマッピング 59
typedef のマッピング 59
単純な IDL 型 59
複雑な IDL 型 59

第 6 章

VisiBroker のプロパティ 61

IIOP を介した Java RMI のプロパティ 61
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 62
VisiBroker ORB のプロパティ 63
POA のプロパティ 68
サーバーマネージャのプロパティ 68
追加プロパティ 68
サーバー側のリソース使用量関連のプロパティ . . 68
クライアント側のリソース使用量関連のプロパティ 69
スマートエージェント関連のプロパティ 69
ロケーションサービスのプロパティ 69
イベントサービスのプロパティ 70
ネーミングサービス（VisiNaming）のプロパティ . . . 70
取り替え可能なバックストアプロパティ 73
すべてのアダプタに共通するデフォルトのプロパティ 73
JDBC アダプタのプロパティ 74
DataExpress アダプタのプロパティ 75
JNDI アダプタのプロパティ 75
VisiNaming サービスのセキュリティ関連プロパティ 76
OAD のプロパティ 77
インターフェースリポジトリのプロパティ 77
クライアント側 IIOP 接続のプロパティ 78
URL ネーミングのプロパティ 79
QoS 関連のプロパティ 79
クライアント側インプロセス接続のプロパティ 79
サーバー側サーバーエンジンのプロパティ 79
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 80
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 80
サーバー側スレッドプール IIOP_TP/IIOP_TP 接続のプロパティ 81
サーバー側スレッドプール BOA_TP/BOA_TP 接続のプロパティ 82

第 7 章

例外処理 83

CORBA モデルにおける例外 83
システム例外 83
SystemException クラス 84
完了状態の取得 84
システム例外のキャッチ 85
例外をシステム例外にダウンキャスト 85
特定のシステム例外のキャッチ 86
ユーザー例外 86
ユーザー例外の定義 86
例外を生成するためのオブジェクトの変更 87
ユーザー例外のキャッチ 87
ユーザー例外へのフィールドの追加 88

第 8 章

サーバーの基礎 89

概要 89
VisiBroker ORB の初期化 89
POA の作成 90
ルート POA へのリファレンスの取得 90
子 POA の作成 90
サーバントメソッドの実装 91
サーバントの作成およびアクティブ化 92
POA のアクティブ化 92
オブジェクトのアクティブ化 92
クライアント要求の待機 92
完全なサンプルコード 93

第 9 章

POA の使い方 95

ポータブルオブジェクトアダプタの概要 95
POA の用語 96
POA の作成と使用の手順 97
POA ポリシー 97
POA の作成 99
POA の命名規則 99
ルート POA の取得 100
POA ポリシーの設定 100
POA の作成およびアクティブ化 100
オブジェクトのアクティブ化 101
オブジェクトの明示的なアクティブ化 101
オンデマンドのオブジェクトのアクティブ化 101
オブジェクトの暗黙的なアクティブ化 102
デフォルトサーバントによるアクティブ化 102

オブジェクトの非アクティブ化 103
サーバントとサーバントマネージャの使い方 104
ServantActivators 105
ServantLocators 107
POA マネージャによる POA の管理 109
現在の状態の取得 110
停止状態 110
アクティブ状態 110
破棄状態 110
非アクティブ状態 111
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 111
サーバーエンジンと POA 112
POA とサーバーエンジンの関連付け 112
サーバーエンジンのエンドポイントのホストの定義 113
サーバー接続マネージャ 113
マネージャ 114
リスナー 114
ディスパッチャ 114
以上のプロパティを使用するタイミング 115
アダプタアクティベータ 117
要求の処理 117

第 10 章

スレッドと接続の管理 119

スレッドの使い方 119
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 120
スレッドポリシー 120
スレッドプールポリシー 121
セッションごとのスレッドポリシー 124
接続管理 125
ServerEngines 125
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 126
スレッドプールディスパッチポリシー 126
セッションごとのスレッドのディスパッチポリシー . 127
コーディングにおける留意点 128
接続管理プロパティの設定 128
適用できるプロパティの有効値 129
プロパティの変更の影響 129
動的に変更できるプロパティ 129
プロパティ値の変更が有効かどうかの確認 130
プロパティ値の変更による影響 130
VisiBroker for Java の高スケーラビリティ設定（Java NIO の使用） 130
ガベージコレクション 131
ORB ガベージコレクションの動作 131
ORB ガベージコレクションに関連するプロパティ 131

第 11 章

tie メカニズムの使い方 133

tie メカニズムのしくみ 133
サンプルプログラム 134
tie メカニズムを使用するサンプルプログラムの場所 134
サーバークラスの変更 134
AccountManager の変更 134
Account クラスの変更 135
tie サンプルのビルド 135

第 12 章

クライアントの基礎 137

VisiBroker ORB の初期化 137
オブジェクトへのバインド 138
バインド処理中に実行される動作 138
オブジェクトのオペレーションの呼び出し 139
オブジェクトリファレンスの操作 139
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 140
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
オブジェクトリファレンスをナローイングする . . . 141
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 142
QoS のインターフェース 142
org.omg.CORBA.Object 142
com.borland.vbroker.CORBA.Object（Borland） 142
org.omg.CORBA.PolicyManager 142
org.omg.CORBA.PolicyCurrent 143
com.borland.vbroker.QoSExt.DeferBindPolicy 143
com.borland.vbroker.QoSExt.ExclusiveConnectionPolicy 144
com.borland.vbroker.QoSExt::RelativeConnectionTimeoutPolicy 144
org.omg.Messaging.RebindPolicy 144
org.omg.CORBA.Messaging.RelativeRequestTimeoutPolicy 146
org.omg.CORBA.Messaging.RelativeRoundTripTimeoutPolicy 146
org.omg.CORBA.Messaging.SyncScopePolicy 147
例外 147
コードセットサポート 148
コードセットのタイプ 148
ネイティブコードセット 148
変換コードセット（CCS） 148
転送コードセット（TCS） 148
コードセットネゴシエーション 148
サポートされるコードセット 148
クライアントランタイムによるクライアント専用アプリケーションの配布 149
使い方 149

第 13 章

IDL の使い方 151

IDL の概要 151
IDL コンパイラでコードを生成する方法 152
IDL 仕様のサンプル 152
生成されるコードの概要 152
_<interface_name>Stub.java 152
<interface_name>.java 153
<interface_name>Helper.java 153
<interface_name>Holder.java 154
<interface_name>Operations.java 154
<interface_name>POA.java 155
<interface_name>POATie.java 155
IDL のインターフェース属性の定義 156
戻り値がない一方向メソッドの指定 156

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 . . . 156

第 14 章
スマートエージェントの使い方 159

スマートエージェントの概要 . . . 159
スマートエージェントの設定と同期に関する推奨事項 159
全般的なガイドライン . . . 160
負荷分散とフォールトトレランスのガイドライン 160
ロケーションサービスのガイドライン . . . 161
スマートエージェントを使用しない場合 . . . 161
スマートエージェントの検索 . . . 161
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 . . . 161
OAD との協力によるオブジェクトへの接続 . . . 162
スマートエージェント（osagent）の起動 . . . 162
詳細出力 . . . 163
エージェントを無効にする . . . 163
スマートエージェントの有効性の確認 . . . 163
クライアントの確認 . . . 164
VisiBroker ORB ドメインの操作 . . . 164
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 . . . 165
スマートエージェントが互いを検出する方法 . . . 165
マルチホームホストのしくみ . . . 166
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 166
ポイントツーポイント通信の使い方 . . . 167
実行時パラメータとしてのホストの指定 . . . 167
環境変数による IP アドレスの指定 . . . 168
agentaddr ファイルによるホストの指定 . . . 168
オブジェクトの有効性の確認 . . . 168
状態を保持しないオブジェクトのメソッドの呼び出し 169
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 . . . 169
OAD に登録されたオブジェクトの複製 . . . 169
ホスト間のオブジェクトの移行 . . . 169
状態を保持するオブジェクトの移行 . . . 169
インスタンス化されたオブジェクトの移行 . . . 170
OAD に登録されたオブジェクトの移行 . . . 170
すべてのオブジェクトとサービスのレポート . . . 170
オブジェクトへのバインド . . . 171

第 15 章
ロケーションサービスの使い方 173

ロケーションサービスの概要 . . . 173
ロケーションサービスのコンポーネント . . . 175
ロケーションサービスエージェントの概要 . . . 175
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 . . . 176
アクセス可能なすべてのインターフェースを検索する 176
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 . . . 176
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 . . . 176
トリガーの概要 . . . 177
トリガーメソッド . . . 177
トリガーの作成 . . . 177
トリガーによって検出される最初のインスタンス 178
エージェントの照会 . . . 178
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 . . . 178
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 . . . 179
トリガーハンドラの書き込みと登録 . . . 181

第 16 章
VisiNaming サービスの使い方 183

概要 . . . 183
名前空間の理解 . . . 184
ネーミングコンテキスト . . . 185
ネーミングコンテキストファクトリ . . . 185
Name と NameComponent . . . 186
名前の解決 . . . 186
文字列化された名前 . . . 186
単純な名前と複雑な名前 . . . 187
VisiNaming サービスの実行 . . . 187
VisiNaming サービスのインストール . . . 187
VisiNaming サービスの設定 . . . 187
VisiNaming サービスの起動 . . . 188
vbj コマンドによる VisiNaming サービスの起動 188
コマンドラインからの VisiNaming サービスの起動 . . . 188
nsutil の設定 . . . 188
nsutil の実行 . . . 189
nsutil を使用して VisiNaming サービスを閉じる . . . 190
VisiNaming サービスのブートストラップ . . . 190
resolve_initial_references の呼び出し . . . 190
-DSVCnameroot の使用 . . . 190
-DORBInitRef の使用 . . . 191
corbaloc URL の使用 . . . 191
corbaname URL の使用 . . . 191
-DORBDefaultInitRef . . . 191
-DORBDefaultInitRef での corbaloc URL の使用 191
-DORBDefaultInitRef での corbaname の使用 . 191
NamingContext . . . 192
NamingContextExt . . . 192
デフォルトネーミングコンテキスト . . . 193
デフォルトネーミングコンテキストの取得 . . . 193
ネーミングコンテキストファクトリの取得 . . . 193
VisiNaming サービスのプロパティ . . . 193
取り替え可能なバックストア . . . 197
バックストアの種類 . . . 197
インメモリアダプタ . . . 197
JDBC アダプタ . . . 198
DataExpress アダプタ . . . 198
JNDI アダプタ . . . 198
設定と使用 . . . 198
プロパティファイル . . . 199
JDBC アダプタのプロパティ . . . 199
DataExpress アダプタのプロパティ . . . 201
JNDI アダプタのプロパティ . . . 201
OpenLDAP の設定 . . . 202
キャッシング機能 . . . 202
キャッシング機能に関する重要事項 . . . 203
クラスタ . . . 203
クラスタリングの基準 . . . 204
Cluster と ClusterManager インターフェース . . . 204
Cluster インターフェースの IDL 仕様 . . . 204
ClusterMangager インターフェースの IDL 仕様 205

NamingContextExtExtended インターフェースの IDL 仕様 205
クラスタの作成 206
明示的クラスタと暗黙的クラスタ 206
負荷分散 207
オブジェクトのフェイルオーバー 207
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 207
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 208
VisiNaming サービスクラスタの設定 209
マスター／スレーブモードでの VisiNaming サービスの設定 210
多数のクライアントが接続する環境での起動 210
VisiNaming サービスフェデレーション 211
VisiNaming サービスのセキュリティ 212
ネーミングクライアント認証 212
SSL を使用するように VisiNaming を設定する . . . 212
メソッドレベル承認 213
import ステートメント 214
サンプルプログラム 215
名前のバインドの例 215
VisiNaming と JDataStore HA を組み合わせて使用する際の設定 216
プライマリミラーの DB を作成する 217
各リスニング接続について JdsServer を呼び出す . . 217
JDataStore HA を設定する 217
VisiNaming の明示的クラスタリングの例を実行する 218
VisiNaming のネーミングフェイルオーバーの例を実行する 219

第 17 章

イベントサービスの使い方 223

概要 223
プロキシコンシューマおよびプロキシサプライヤ . . 224
OMG コモンオブジェクトサービス仕様 224
通信モデル 225
プッシュモデル 225
プルモデル 226
イベントチャネルの使い方 226
イベントチャネルの作成 228
プッシュサプライヤおよびコンシューマのサンプル . . . 228
プッシュサプライヤ／コンシューマサンプル 228
プッシュモデルサンプルの実行 229
PullModel サンプルの実行 229
PullView サンプルの実行 229
PullSupply 229
PullSupply の実行 230
pull メソッドと try_pull メソッドのインプリメンテーション 230
PullSupply の main メソッド 231
PullConsume 231
PullConsume の実行 232
イベントサービスの起動 233
キューの長さの設定 233
インプロセスイベントチャネル 234
インプロセスイベントチャネルの使い方 235
Java EventLibrary クラス 235
Java サンプル 235
import ステートメント 235

第 18 章

VisiBroker サーバーマネージャの使用 237

サーバーマネージャの概要 237
サーバーでのサーバーマネージャの有効化 237
サーバーマネージャリファレンスの取得 238
コンテナの使用 238
Storage インターフェース 239
Container インターフェース 239
Container クラス 239
Container のメソッド（Java） 239
プロパティの操作とクエリーに関連するメソッド 239
オペレーションに関連するメソッド 240
子コンテナに関連するメソッド 240
ストレージに関連するメソッド 241
Storage インターフェース 241
Storage インターフェースのクラスとメソッド . . . 241
Storage クラス 241
Storage インターフェースのメソッド 241
サーバーマネージャに対するアクセスの制限 242
サーバーマネージャ IDL 242
サーバーマネージャの例 244
最上位コンテナへのリファレンスの取得 245
すべてのコンテナとそれらのプロパティの取得 . . . 245
プロパティの取得と設定，およびファイルへの保存 . 245
Container でのオペレーションの呼び出し 246
カスタムコンテナ 247

第 19 章

VisiBroker ネイティブメッセージングの使用 249

はじめに 249
2 フェーズ呼び出し（2PI） 249
ポーリング／プルモデルとコールバックモデル . . . 250
非ネイティブメッセージングと IDL の変形 250
ネイティブメッセージングソリューション 250
リクエストエージェント 251
ネイティブメッセージングの Current オブジェクト 251
コアオペレーション 252
StockManager サンプル 252
ポーリング／プルモデル 253
コールバックモデル 254
高度な項目 257
グループポーリング 257
応答受信者における Cookie と応答逆多重化 258
2 フェーズ呼び出しへの展開 259
応答ドロップ 261
コレクションの手動破棄 261
非抑制早期リターンモード 262
コールバックモデルでのポーラー生成の抑制 263
ネイティブメッセージングの API 仕様 263
RequestAgentEx インターフェース 263
create_request_proxy() 263
destroy_request() 264
RequestProxy インターフェース 264
the_receiver 264
poll() 265
destroy() 265
ローカルインターフェースの Current オブジェクト 266
suppress_mode() 266

wait_timeout 266
the_cookie 266
request_tag 266
the_poller 266
reply_not_available 267
ReplyRecipient インターフェース 268
reply_available() 268
コアオペレーションのセマンティクス 269
ネイティブメッセージングの相互運用性仕様 269
ネイティブメッセージングはネイティブ GIOP を使用 269
ネイティブメッセージングのサービスコンテキスト 270
NativeMessaging タグ付きコンポーネント 271
Borland ネイティブメッセージングの使用 271
リクエストエージェントとクライアントモデルの使用 271
Borland リクエストエージェントの起動 271
Borland リクエストエージェントの URL 271
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 272
Borland リクエストエージェントの vbroker プロパティ 272
vbroker.requestagent.maxThreads 272
vbroker.requestagent.maxOutstandingRequests 272
vbroker.requestagent.blockingTimeout . . . 272
vbroker.requestagent.router.ior 272
vbroker.requestagent.listener.port 272
vbroker.requestagent.requestTimeout 272
CORBA Messaging との相互運用性 273

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 275

オブジェクトとサーバーの自動アクティブ化 275
インプリメンテーションリポジトリデータの検索 . . 275
サーバーのアクティブ化 276
OAD の使い方 276
OAD の起動 276
OAD ユーティリティの使い方 277
インターフェース名をリポジトリ ID に変換する . . 277
oadutil リストによるオブジェクトの一覧表示 . . . 278
oadutil によるオブジェクトの登録 279
例：リポジトリ ID の指定 280
例：IDL インターフェース名の指定 280
OAD にリモートで登録する 280
スマートエージェントを使用しない OAD の使用 281
ネーミングサービスによる OAD の使用 281
オブジェクトの複数のインスタンスの区別 282
CreationImplDef クラスによるアクティブ化プロパティの設定 282
ORB インプリメンテーションを動的に変更する . . 282
OAD::reg_implementation による OAD 登録 . . 283
オブジェクトの作成と登録の例 283
OAD で渡す引数 284
オブジェクトの登録解除 284
oadutil ツールによるオブジェクトの登録解除 . . . 284
登録解除のサンプル 285
OAD からの登録解除の操作 285
インプリメンテーションリポジトリの内容の表示 . . 286
OAD の IDL インターフェース 286

第 21 章

インターフェースリポジトリの使い方 289

インターフェースリポジトリの概要 289
インターフェースリポジトリの内容 290
作成できるインターフェースリポジトリの数 290
irep を使ったインターフェースリポジトリの作成と表示 291
irep を使ったインターフェースリポジトリの作成 . .291
インターフェースリポジトリの内容の表示 292
idl2ir を使ったインターフェースリポジトリの更新 . . .292
インターフェースリポジトリの構造体の概要 292
インターフェースリポジトリ内のオブジェクトの識別 .293
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 293
継承元のインターフェース 294
インターフェースリポジトリへのアクセス 295
インターフェースリポジトリのサンプルプログラム . . .295

第 22 章

動的起動インターフェースの使い方 299

動的起動インターフェースの概要 299
重要な DII の概念 300
Request オブジェクトの使用 300
Any 型を使った引数のカプセル化 301
要求の送信オプション 301
応答の受信オプション 302
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 302
DII を使用するサンプルプログラム 302
idl2java コンパイラの使い方 302
共通オブジェクトリファレンスの取得 303
要求の作成と初期化 303
Request インターフェース 303
DII 要求を作成および初期化する方法 304
create_request メソッドの使い方 304
_request メソッドの使い方 304
Request オブジェクトを作成するサンプルコード . .305
要求に引数を設定する方法 305
NVList クラスを使用して，引数リストを実装する .305
NamedValue クラスを使用して，入力引数と出力引数を設定する 306
Any クラスを使ってタイプセーフに引数を渡す . . .306
TypeCode クラスを使用して，引数または属性の型を表す 307
DII 要求の送信と結果の受信 308
要求を呼び出す 308
send_deferred メソッドを使用して，遅延 DII 要求を送信する 309
send_oneway メソッドを使用して，非同期 DII 要求を送信する 310
複数の要求を送信する 310
複数の要求を受信する 310
DII によるインターフェースリポジトリの使い方 310

第 23 章

動的スケルトンインターフェースの使い方 313

動的スケルトンインターフェースの概要 313
idl2java コンパイラの使い方 313

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 314
DSI を使用するサンプルプログラム 314
DynamicImplementation クラスの拡張 314
動的要求のオブジェクトを設計するサンプル 314
リポジトリ ID の指定 316
ServerRequest クラスについて 317
Account オブジェクトの実装 317
AccountManager オブジェクトの実装 317
入力パラメータの処理 318
戻り値の設定 318
サーバーインプリメンテーション 318

第 24 章 ポータブルインターセプタの使い方 321

ポータブルインターセプタの概要 321
インターセプタの種類 322
ポータブルインターセプタの種類 322
ポータブルインターセプタと情報インターフェース . . .322
Interceptor クラス 322
リクエストインターセプタ 323
ClientRequestInterceptor 323
クライアント側の規則 324
ServerRequestInterceptor 324
サーバー側の規則 325
IOR インターセプタ 325
ポータブルインターセプタ（PI）Current 326
Codec 326
CodecFactory 326
ポータブルインターセプタの作成 327
例を次に示します。PortableInterceptor の作成 .327
ポータブルインターセプタの登録 328
ORBInitializer の登録 328
例を次に示します。ORBInitializer の登録 329
VisiBroker によるポータブルインターセプタの拡張機能 330
POA スコープ付きサーバーリクエストインターセプタ 330
システム例外の挿入と抽出 330
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 330
ClientRequestInfo の制限 330
ServerRequestInfo の制限 331
ポータブルインターセプタのサンプル 331
例：client_server 331
サンプルの目的 331
必要なパッケージのインポート 332
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 .332
サーバー側インターセプタの ORBInitializer の実装 333
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 334
クライアントの ClientRequestInterceptor 実装 . .334
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 . . .335
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 335
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 335
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 335
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 338
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 338
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 338
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 . . .338
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 338
クライアントおよびサーバーアプリケーションの開発 . 341
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 341
サーバーアプリケーションのインプリメンテーション 342
コンパイルの手順 .343
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 343

第 25 章 VisiBroker インターセプタの使い方 345

インターセプタの概要 345
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 346
クライアントインターセプタ 346
BindInterceptor 346
ClientRequestInterceptor 347
サーバーインターセプタ 347
POALifeCycleInterceptor 347
ActiveObjectLifeCycleInterceptor 348
ServerRequestInterceptor 348
IORCreationInterceptor 348
サービスリゾルバインターセプタ 349
デフォルトのインターセプタクラス 349
VisiBroker ORB によるインターセプタの登録 . . .349
インターセプタオブジェクトの作成 350
インターセプタのロード 350
サンプルインターセプタ 350
サンプルコード 350
クライアント／サーバーインターセプタのサンプル .351
ServiceResolverInterceptor のサンプル 352
コードリスト 353
SampleServerLoader 353
SamplePOALifeCycleInterceptor 353
SampleServerInterceptor 354
SampleClientInterceptor 355
SampleClientLoader 356
SampleBindInterceptor 356
インターセプタ間の情報の受け渡し 357
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 357
インターセプタの呼び出しポイントの順序 357
クライアント側インターセプタ 358
サーバー側インターセプタ 358
POA 作成中の ORB イベントの順序 358
POA リファレンス作成中の ORB イベントの順序 . 359

第 26 章
オブジェクトラッパーの使い方 361

オブジェクトラッパーの概要 361
型付きと型なしのオブジェクトラッパー 361
idl2java の特殊な要件 362
Object ラッパーのサンプルアプリケーション 362
型なしオブジェクトラッパー 362
複数の型なしオブジェクトラッパーの使い方 363
pre_method 呼び出しの順序 363
post_method 呼び出しの順序 363
型なしオブジェクトラッパーの使い方 364
型なしオブジェクトラッパーファクトリの実装 . . . 364
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 364
pre_method メソッドと post_method メソッドに共通の引数 365
型なしオブジェクトラッパーファクトリの作成と登録 365
型なしオブジェクトラッパーの削除 367
型付きオブジェクトラッパー 367
複数の型付きオブジェクトラッパーの使い方 368
呼び出しの順序 369
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 369
型付きオブジェクトラッパーの使い方 369
型付きオブジェクトラッパーの実装 370
クライアント向け型付きオブジェクトラッパーの登録 370
サーバー向けの型付きオブジェクトラッパーの登録 371
型付きオブジェクトラッパーの削除 372
型なしラッパーと型付きラッパーの複合的な使い方 . . 372
型付きラッパーのコマンドライン引数 372
型付きラッパーのイニシャライザ 373
型なしラッパーのコマンドライン引数 374
型なしラッパーのイニシャライザ 374
サンプルアプリケーションの実行 375
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 375
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 376
型付きラッパーと型なしラッパーをオンにする . 376
共用クライアント／サーバーを実行する 376

第 27 章
イベントキュー 379

イベントタイプ 379
接続イベント 379
イベントリスナー 379
IDL 定義 379
ConnInfo 構造体 380
EventListener インターフェース 380
ConnEventListeners インターフェース 380
EventQueueManager インターフェース 381
EventQueueManager を返す方法 381
イベントキューのサンプルコード 381
EventListener の登録 381
EventListener の実装 382

第 28 章
IIOP を介した RMI の使い方 383

RMI over IIOP の概要 383
RMI-IIOP を使用する Java アプレットの設定 383
java2iiop と java2idl 383
java2iiop の使い方 .384
サポートされるインターフェース 384
java2iiop の実行 .384
Java クラスから IDL への逆マッピング 384
開発の完了 385
RMI-IIOP Bank サンプル 385
サポートされるデータ型 387
プリミティブデータ型のマッピング 387
複合データ型のマッピング 387
インターフェース 387
配列 387

第 29 章
動的に管理される型の使い方 389

DynAny インターフェースの概要 389
DynAny サンプル .389
DynAny 型 389
DynAny 使用上の制限 390
DynAny の作成 390
DynAny の値の初期化とアクセス 390
構造データ型 391
構造データ型内の複数のコンポーネント間の移動 . .391
DynEnum 391
DynStruct 391
DynUnion 391
DynSequence と DynArray 392
DynAny サンプル IDL 392
DynAny サンプルクライアントアプリケーション 392
DynAny サンプルサーバーアプリケーション 393

第 30 章
valuetype の使い方 399

valuetype について 399
valuetype IDL サンプルコード 399
具象 valuetype 399
valuetype の派生 400
共有セマンティクス 400
null セマンティクス 400
ファクトリ 400
抽象 valuetype 400
valuetype の実装 400
valuetype の定義 .401
IDL ファイルのコンパイル 401
valuetype 基底クラスの継承 401
Factory クラスの実装 402
ファクトリを VisiBroker ORB に登録 402
ファクトリの実装 402
ファクトリと valuetype 403
valuetype の登録 .403
ボックス化 valuetype 403
抽象インターフェース 404
custom valuetype .404
truncatable valuetype 405

第 31 章

URL ネーミングの使い方 407

URL ネーミングサービス 407
　URL ネーミングサービスの例 407
オブジェクトの登録 408
URL によるオブジェクトの検索 409

第 32 章

双方向通信 411

双方向 IIOP の使用 411
双方向 VisiBroker ORB のプロパティ 411
双方向サンプルについて 412
既存のアプリケーションで双方向 IIOP を有効にする . . 413
双方向 IIOP を明示的に有効にする 413
　一方向または双方向接続 414
　POA で双方向 IIOP を有効にする 414
セキュリティに関する注意 414

第 33 章

VisiBroker における BOA の使い方 417

VisiBroker を使った BOA コードのコンパイル 417
BOA オプションのサポート 417
BOA を使用する場合の制限事項 417
オブジェクトアクティベータの使い方 418
BOA の下でのネーミングオブジェクト 418
　オブジェクト名 418

第 34 章

オブジェクトアクティベータの使い方 419

オブジェクトのアクティブ化の遅延 419
Activator インターフェース 419
サービスのアクティブ化の使い方 420
　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 421
　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 421
　　odb.idl インターフェース 422
　　サービスアクティベータの実装 422
　　サービスアクティベータのインスタンス化 423
　　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 423

第 35 章

CORBA 例外 425

CORBA 例外の説明 425
OMG 指定のヒューリスティックな例外 429
OMG 指定のその他の例外 430

第 36 章

Web サービスの概要 433

　Web サービスアーキテクチャ 433
　標準 Web サービスアーキテクチャ 434
VisiBroker Web サービスアーキテクチャ 434
　Web サービス関連ファイル 434
　Web サービスランタイム 435
Web サービスとしての CORBA オブジェクトの公開 . . 437
　開発 438
　配布 439
SOAP/WSDL の互換性 439

索引 441

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

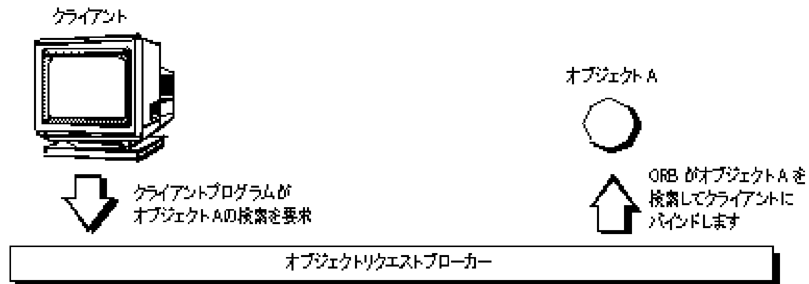
コモンオブジェクトリクエストブローカーキテクチャ（Common Object Request Broker Architecture，CORBA）を利用すると，記述された言語や存在する場所に関係なく，分散アプリケーションどうしの相互運用が可能になります。

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。

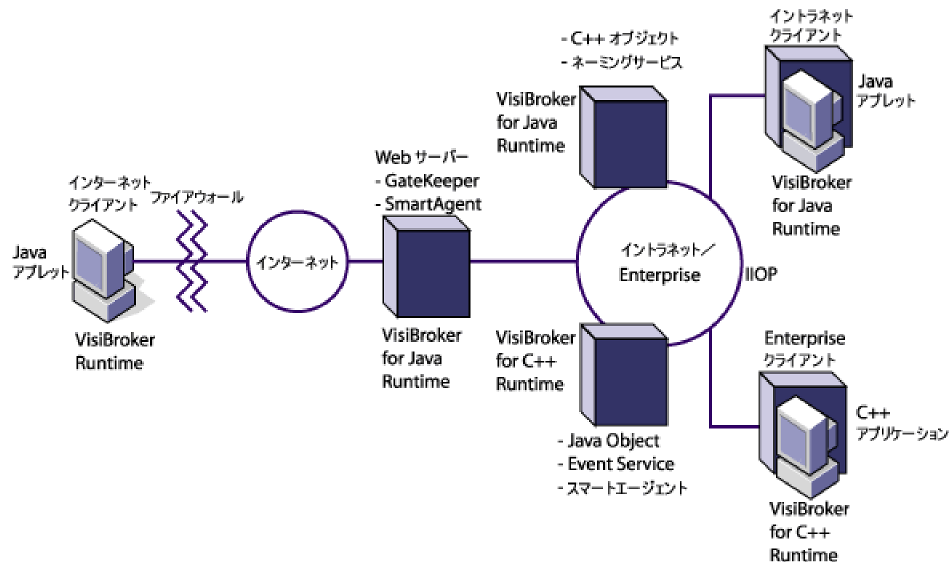
図 2.1 オブジェクトを操作するクライアントプログラム



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」と第 12 章「クライアントの基礎」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。

IDL を使用しないインターフェース定義

VisiBroker の `java2iiop` コンパイラを使用すると，インターフェース定義言語（IDL）のかわりに Java 言語でインターフェースを定義できます。既存の Java コードがあり，CORBA 分散オブジェクトと相互運用する場合や IDL を習得したくない場合は，`java2iiop` コンパイラを使用します。

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 ガイド』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。

Java 開発環境

VisiBroker は，Java 実行時環境で次のコンポーネントを使用します。

- 12 ページの「Java 2 Standard Edition」
- 13 ページの「Java 実行時環境」

Java 2 Standard Edition

VisiBroker ORB を使用するアプレットやアプリケーションの開発には，Borland JBuilder などの Java 開発環境が必要です。JavaSoft の Java 開発者キット（JDK）には，Java 実行時環境（JRE）も含まれています。

Sun Microsystems は, JavaSoft の JDK (Java 実行時環境を含む) を Solaris, Windows NT プラットフォーム向けに提供しています。JDK は，Sun Microsystems の Web サイト http://java.sun.com からダウンロードできます。

JDK は，IBM AIX，OS/2，SGI IRIX，および HP-UX にも移植済みです。これらのバージョンは，各ハードウェアベンダーの Web サイトからダウンロードできます。これらのプラットフォームで利用できる機能については，Sun Microsystems の JavaSoft Web サイト http://java.sun.com/products/jdk を参照してください。

Java 実行時環境

VisiBroker のサービスとツールを実行するには，すべてのエンドユーザーで Java 実行時環境が必要です。Java 実行時環境は Java アプリケーションを解釈して実行するエンジンで，通常 Java 開発環境にバンドルされています。詳細については，12 ページの「Java 2 Standard Edition」を参照してください。

GateKeeper の要件

VisiBroker Gatekeeper を使用するには，Servlet 2.1 API を使用する必要があります。これは，JavaServer Web Development Kit 1.0.1 から入手できます。

Java 対応の Web ブラウザ

アプレットは，Netscape Communicator，Netscape Navigator，Microsoft Internet Explorer など，Java 対応の Web ブラウザで実行できます。これらの Java 対応の Web ブラウザは，それぞれ次の URL から入手できます。

- http://www.netscape.com/
- http://microsoft.com/ie/

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 製品との相互運用性

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 Java へのマッピング

VisiBroker は，『OMG IDL/Java Language Mapping Specification』に準拠しています。idl2java コンパイラに実装されている VisiBroker の現在の IDL/Java 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 Java 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/Java 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_java.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。

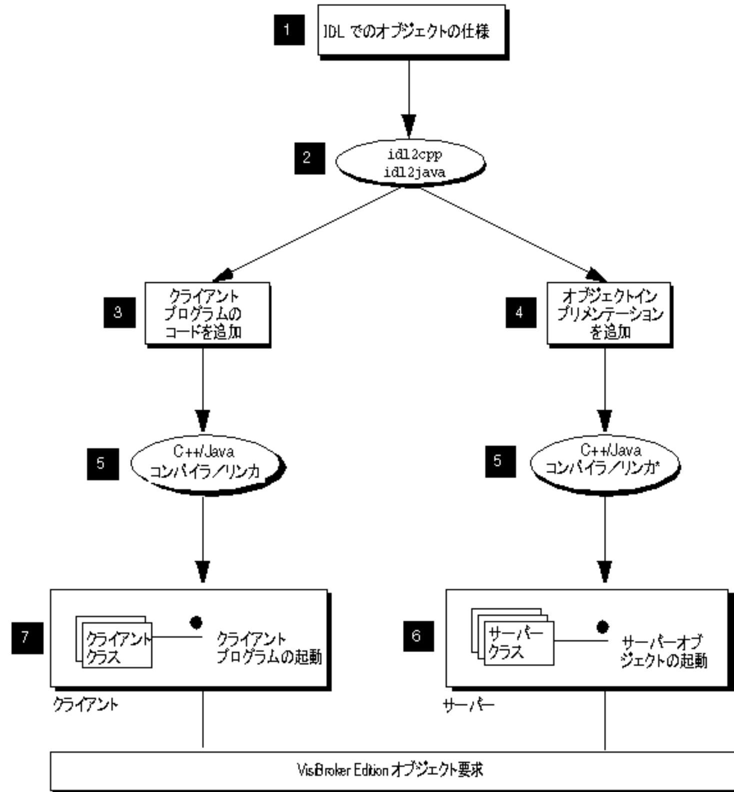
サーバーオブジェクトのコードのインプリメンテーションを完成するには，AccountPOA クラスと AccountManagerPOA クラスからクラスを派生し，それらのインターフェースのメソッドにインプリメンテーションを提供し，サーバーの main ルーチンを実装します。

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，idl2java 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，現在の残高を取得するためのメソッドを 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 idl2java コンパイラがクライアントプログラムの Java クラス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，すべてのメソッドの呼び出しで Java クラスを使用します。

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 Java コンパイラへの入力として使用して，クライアントとサーバーの実行可能なクラスを生成します。

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> idl2java Bank.idl
```

idl2java 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

Java

Java では，1 つのファイルに記述できるパブリックインターフェース，またはパブリッククラスは 1 つだけなので，IDL ファイルをコンパイルすると複数の .java ファイルが生成されます。これらのファイルは，Bank という名前で生成されたサブディレクトリに保存されます。Bank は，IDL で指定されているモジュール名で，生成されたファイルが属するパッケージです。生成された .java ファイルのリストは次のとおりです。

- _AccountManagerStub.java：クライアント側の AccountManager オブジェクトのスタブコードです。
- _AccountStub.java：クライアント側の Account オブジェクトのスタブコードです。
- Account.java：Account インターフェース宣言です。
- AccountHelper.java：AccountHelper クラスを宣言します。このクラスは，便利なユーティリティメソッドを定義します。
- AccountHolder.java：AccountHolder クラスを宣言します。このクラスは，Account オブジェクトを渡すためのホルダーを提供します。
- AccountManager.java：AccountManager インターフェース宣言です。
- AccountManagerHelper.java：AccountManagerHelper クラスを宣言します。このクラスは，便利なユーティリティメソッドを定義します。
- AccountManagerHolder.java：AccountManagerHolder クラスを宣言します。このクラスは，AccountManager オブジェクトを渡すためのホルダーを提供します。

- AccountManagerOperation.java：
  このインターフェースは，Bank.idl ファイルの AccountManager インターフェースで定義されているメソッドのシグニチャを宣言します。
- AccountManagerPOA.java：サーバー側の AccountManager オブジェクトインプリメンテーションの POA サーバントコード（インプリメンテーションベースコード）です。
- AccountManagerPOATie.java：tie メカニズムを使用して，サーバー側で AccountManager オブジェクトを実装するために使用されるクラスです。tie メカニズムについては，第 11 章「tie メカニズムの使い方」を参照してください。
- AccountOperations.java：このインターフェースは，Bank.idl ファイルの Account インターフェースで定義されているメソッドのシグニチャを宣言します。
- AccountPOA.java：サーバー側の Account オブジェクトインプリメンテーションの POA サーバントコード（インプリメンテーションベースコード）です。
- AccountPOATie.java：tie メカニズムを使用して，サーバー側で Account オブジェクトを実装するために使用されるクラスです。tie メカニズムについては，第 11 章「tie メカニズムの使い方」を参照してください。

ステップ 3：クライアントの実装

Client.java

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，idl2java コンパイラによって生成される Bank パッケージ内に存在します。

Client.java 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 クラスは，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. AccountManager オブジェクトの open を呼び出して，Account オブジェクトを取得する。
4. Account オブジェクトの balance を呼び出して，残高を取得する。

```
public class Client {
   public static void main(String[] args) {
      // ORB を初期化します。
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
      // マネージャの ID を取得します。
      byte[] managerId = "BankManager".getBytes();
      // 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 manager =
            Bank.AccountManagerHelper.bind(orb, "/bank_agent_poa", managerId);
      // 口座名またはデフォルトとして args[0] を使用します。
      String name = args.length > 0 ? args[0] : "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 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。
      System.out.println("The balance in " + name + "'s account is $" +
balance);
   }
}
```

AccountManager オブジェクトへのバインド

クライアントプログラムは，open(String name) メソッドを呼び出す前に，bind() メソッドを使用して，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() メソッドのインプリメンテーションは，idl2java コンパイラによって自動的に生成されます。bind() メソッドは，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，AccountManager オブジェクトへのオブジェクトリファレンスが返されます。

Account オブジェクトの取得

次に，クライアントプログラムは AccountManager オブジェクトの open() メソッドを呼び出して，指定された顧客名に対応する Account オブジェクトへのオブジェクトリファレンス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() メソッドを使って残高を取得できます。クライアント側の balance() メソッドは，実際には idl2java 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。

AccountManagerHelper.java

Java このファイルは Bank パッケージ内にあります。このファイルには AccountManagerHelper オブジェクトがあり，このオブジェクトを実装するサーバーにバインドするためのメソッドをいくつか定義しています。bind() クラスメソッドは，指定された POA マネージャとコンタクトし，オブジェクトをリゾルブします。このサンプルアプリケーションでは，bind メソッドのうち，オブジェクト名を受け取るバージョンを使用しますが，クライアントから特定のホストを指定したり，特別なバインドオプションを指定することもできます。Helper クラスの詳細については，『VisiBroker プログラマーズリファレンス』を参照してください。

```
package Bank;
public final class AccountManagerHelper {
   . . .
   public static Bank.AccountManager bind(org.omg.CORBA.ORB orb) {
      return bind(orb, null, null, null);
      . . .
   }
}
```

その他のメソッド

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ソッドがいくつかあります。

サンプルクライアントアプリケーションでそれらのメソッドとメンバー関数を使用することはほとんどありませんが，詳細については，『VisiBroker プログラマーズリファレンス』に記載されています。

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，idl2java コンパイラによって生成される Bank パッケージに格納されています。Server.java 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，C++ と Java のサーバー側プログラムのサンプルです。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。

```
public class Server {
   public static void main(String[] args) {
      try {
         // ORB を初期化します。
         org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
         // ルート POA へのリファレンスを取得します。
         POA rootPOA =
               POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
         // 永続的 POA のポリシーを作成します。
            org.omg.CORBA.Policy[] policies = {
            rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)
         };
         // 適切なポリシーで myPOA を作成します。
         POA myPOA = rootPOA.create_POA( "bank_agent_poa",
               rootPOA.the_POAManager(),
               policies );
         // サーバントを作成します。
         AccountManagerImpl managerServant = new AccountManagerImpl();
         // サーバントの ID を決定します。
         byte[] managerId = "BankManager".getBytes();
         // 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
         rootPOA.the_POAManager().activate();
         System.out.println(myPOA.servant_to_reference(managerServant) + " is
                  ready.");
         // 着信要求を待機します。
         orb.run();
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

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 vbmake.bat 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> vbmake
```

`vbmake` コマンドは，`idl2java` コンパイラを実行して各ファイルをコンパイルするバッチファイルです。

`vbmake` 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。

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> make java
```

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 vbj 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> vbj 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> vbj 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> vbj 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。

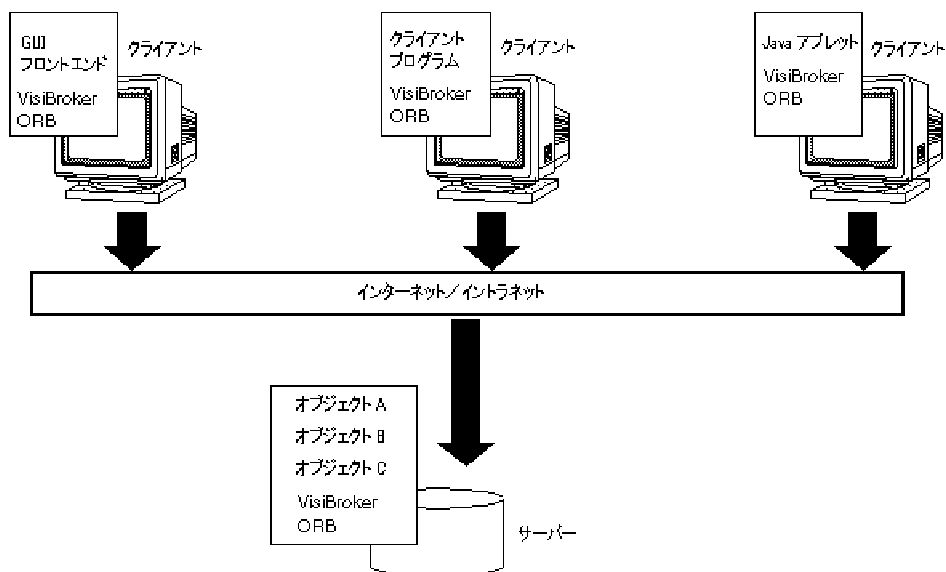
```
The balance in the account in $168.38.
```

VisiBroker を使ったアプリケーションの配布

VisiBroker は配布フェーズでも使用します。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フェーズです。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 3.2　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 Java 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。

- Java 実行時環境。
- `vbjorb.jar` ファイルにアーカイブされた VisiBroker Java パッケージ。
  これは，VisiBroker のインストール先の `lib` 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，Java 実行時環境と VisiBroker パッケージをインストールする必要があります。

環境変数

vbj 実行可能プログラムを使用すると，環境変数を自動的に設定します。

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。vbroker.agent.addr 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。24 ページの表 3.1「Java クライアントアプリケーションのコマンドライン引数」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。

UDP ポート番号は，vbroker.agent.port（Java）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。

vbj の使用

Java

vbj コマンドを使用すると，アプリケーションの動作を制御するためのコマンドライン引数を入力して，アプリケーションを起動できます。

```
vbj -Dvbroker.agent.port=10000 <class>
```

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，21 ページの「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。

クライアントは，vbj コマンドを使って起動するか，Java 対応の Web ブラウザ内から起動できます。

次の表は，Java 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。

表 3.1　Java 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
-DORBagentAddr=<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。
-DORBagentPort=<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 ORB ドメインが必要な場合に便利です。ポート番号を設定しなかった場合，デフォルトは 14000 になります。
-DORBmbufSize=<buffer_size>	VisiBroker がオペレーションリクエストの実行に使用する中間バッファのサイズを指定します。パフォーマンスの向上のため，VisiBroker ORB は，以前のバージョンの VisiBroker より複雑なバッファ管理を行っています。送信および受信のデフォルトのバッファサイズは 4（4 KB）です。送信または受信したデータのサイズがデフォルトより大きい場合は，新しいバッファが各要求／応答に割り当てられます。作成したアプリケーションが 4 KB より大きなデータを頻繁に送信する場合，このシステムプロパティを使用してより大きなバイト数をデフォルトのバッファサイズに指定すれば，バッファ管理機能を有効に活用できます。
-DORBtcpNoDelay=<false|true>	true に設定すると，すべてのネットワーク接続でデータが即座に送信されます。デフォルトは false です。この場合，ネットワーク接続はバッファが満たされるとすぐにデータを一括して送信します。
-DORBconnectionMax=<integer>	OAid TSession が選択されている場合に，オブジェクトインプリメンテーションで利用できる接続の最大数を指定します。値を指定しない場合，デフォルトは無制限になります。
-DORBconnectionMaxIdle=<integer>	ネットワーク接続がアイドルになり，VisiBroker によってシャットダウンされるまでの時間をミリ秒単位で指定します。デフォルトは 360 です。これは，接続がタイムアウトにならないことを意味します。インターネットアプリケーションの場合は，このオプションを設定する必要があります。

Java でのサーバーアプリケーションの実行

サーバーアプリケーションは，1 つ以上の VisiBroker ORB オブジェクトをクライアントアプリケーションに提供するアプリケーションです。サーバーアプリケーションは，vbj コマンドを使って起動するか，オブジェクトアクティベーションデーモン（oad）を使ってアクティブ化します。

次の表は，Java サーバー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。

表 3.2　Java サーバー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
-DOAipAddr <hostname|ip_address>	オブジェクトアダプタで使用されるホスト名または IP アドレスを指定します。このオプションは，ホストに複数のネットワークインターフェースがあり，BOA がこれらのインターフェースの 1 つだけに関連付けられている場合に使用します。このオプションを指定しない場合は，ホストのデフォルトのアドレスが使用されます。
-DOAport <port_number>	オブジェクトアダプタによって新しい接続の監視に使用されるポート番号を指定します。
-DOAid <TPool|TSession>	BOA が使用するスレッドポリシーを指定します。下位互換モードでない限り，デフォルトは TPool です。下位互換モードの場合，デフォルトは TSession です。
-DOAthreadMax <integer>	OAid に TPool が選択されている場合に，利用できるスレッドの最大数を指定します。指定しない場合，または 0 を指定した場合，スレッド数は無制限になります。正確には，スレッド数はシステムリソースによってのみ制限されます。

表 3.2 Java サーバーアプリケーションのコマンドライン引数（続き）

オプション	説明
-DOAthreadMin <integer>	スレッドプールで利用できるスレッドの最小数を指定します。指定しない場合のデフォルトは 0 です。このオプションは，OAid TPool が選択されている場合にしか指定できません。
-DOAthreadMaxIdle <integer>	スレッドがまったく要求に応えないまま存在できる時間を秒単位で指定します。指定された時間を過ぎてアイドル状態のスレッドは，システムに返されます。デフォルトでは 300 秒に設定されています。
-DOAconnectionMax <integer>>	OAid TSession が選択されている場合に，利用できるスレッドの最大数を指定します。指定しない場合のデフォルトは，無制限です。
-DOAconnectionMaxIdle <integer>	接続がトラフィックなしのアイドル状態を継続できる時間を指定します。この時間を過ぎてもアイドル状態が続いている接続は，VisiBroker によってシャットダウンされます。デフォルトでは 0 に設定されています。これは，接続が自動的にタイムアウトにならないことを意味します。インターネットアプリケーションの場合は，このオプションを設定する必要があります。

第 4 章

Java 対応プログラマツール

この章では，VisiBroker for Java が提供するプログラマツールについて説明します。コマンドの構文は，コマンド，コマンドの実行に必要な引数，およびコマンドラインオプションで構成されます。一部のコマンドには引数がありませんが，オプションが提供されています。

VisiBroker（バージョン 6.5 以降）は，いくつかの追加機能を VisiBroker for Java ツールとして提供しています。これらの機能を使用して，クラスパスや ORB プロパティの設定など，アプリケーションの設定を柔軟に行うことができます。VisiBroker では，設定ファイルベースのシステムを使用して設定を指定できます。さらに，VisiBroker バージョン 6.5 からは，これらすべてのツールがネイティブに構築された起動プログラムを使用して起動されます。これまでの UNIX ベースの起動プログラムはスクリプトベースで，提供される設定機能は非常に限定されていました。

オプション

VisiBroker for Java のプログラマツールは，どれも汎用のオプションとツールに固有のオプションの両方を持っています。各ツールに固有のオプションは，そのツールの節で説明します。このリストにあるオプションはすべてデフォルトで有効になり，ハイフン（-）の後に続きます。デフォルト値をオフにするには，先頭に -no_ を付けるかハイフンを削除します。たとえば，認識できない #pragma があった場合に警告を出す場合，デフォルト値は次のようになります。

```
warn_unrecognized_pragmas
```

デフォルトをオフにするには，次のオプションを使用します。

```
-no_warn_unrecognized_pragmas
```

すべてのプログラマツールで使用できるオプションを次に示します。

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。

表 4.1　プログラマツールの共通オプション

オプション	説明
-VBJdebug	VisiBroker for Java のデバッグ情報を出力します。
-J<java_option>	java_option ディレクトリを Java 仮想マシンに渡します。
-VBJversion	使用中の VisiBroker for Java のバージョンを出力します。
-VBJprop <property>=<value>	指定されたプロパティを VBJ 実行可能プログラムに渡します。
-VBJjavavm <vm-name>	Java VM のパスとフラグを指定します。指定しない場合は，デフォルト値 java が使用されます。
-VBJclasspath <classpath>	クラスパスを指定します。ここで入力した値は，CLASSPATH 環境変数より優先します。
-VBJaddJar <jarfile>	VM を実行する前に，<jarfile> を CLASSPATH に追加します。絶対パスが指定されていない場合，この jarfile は <launcher-location>/../lib からの相対パスとみなされます。
-VBJconfig <config-file-name>	起動プログラムによって使用される設定ファイルのパス。指定されない場合，デフォルトの場所は <install-dir>/bin/vbj.config（起動プログラム vbjc の場合は vbjc.config）になります。
-help|-h|-?|-usage	コマンドの使用方法を表示します。

idl2ir

このツールを使用すると，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。idl2ir コマンドを使用して実行されます。

構文

```
idl2ir [options] {filename}
```

例

```
idl2ir -irep my_repository -replace java_examples/bank/Bank.idl
```

説明

idl2ir コマンドは IDL ファイルを入力とし，インターフェースリポジトリサーバーにバインドして，filename 内の IDL 構造をリポジトリに記入します。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，古い項目が変更されます。

キーワード

キーワードには，次に挙げるオプションと処理される IDL 入力ファイルが含まれています。

オプション

idl2ir では，次のオプションを使用できます。

表 4.2　idl2ir のオプション

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 off 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 off 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。
-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。
-[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_]builtin (TypeCode|Principal)	組み込み型 ::TypeCode または ::Principal を作成します。デフォルトは on です。
-[no_]warn_unrecognized_pragmas	プラグマを認識できない場合に警告を表示します。デフォルトは on です。

表 4.2　idl2ir のオプション（続き）

オプション	説明
-[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_]preprocess	解析前に入力ファイルをプリプロセスします。デフォルトは on です。
-[no_]preprocess_only	プリプロセス後に入力ファイルの解析を停止します。デフォルトは off です。
-[no_]warn_all	すべての警告を同時にオンまたはオフにします。デフォルトは on です。
-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。
-deep	ディープ（シャローでない）マージを適用します。デフォルトは off です。
-replace	マージしないで，リポジトリ全体を置き換えます。デフォルトは off です。
file1 [file2]...	処理対象となる 1 つ以上のファイル。stdin の場合は “-”。
-h, -help, -usage, -?	使用方法を表示します。

ir2idl

このツールを使用すると，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。ir2idl コマンドを使用して実行されます。

構文

```
ir2idl [options] filename
```

例

```
ir2idl -irep my_repository -o my_file
```

説明 ir2idl コマンドは，IR にバインドし，その内容を IDL 形式で出力します。

キーワード キーワードには，次に挙げるオプションが含まれています。

オプション ir2idl では，次のオプションを使用できます。

表 4.3　ir2idl のオプション

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 on 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 Borland 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。

idl2java

このツールは，IDL ソースファイルから Java ソースコードを生成します。idl2java コマンドを使用して実行されます。

構文

```
idl2java [options] {filename}
```

例

```
idl2java -no_tie Bank.idl
```

説明 idl2java コマンドは Java ベースのプリプロセッサです。IDL ソースファイルをコンパイルし，作成したディレクトリ構造の中に IDL 宣言の Java マッピングを格納します。Java では，1 つのファイルに public のインターフェースまたはクラスを 1 つしか記述できない

ので，1 つの IDL ファイルは多数の Java ファイルにマップされるのが普通です。IDL のファイル名には，.idl 拡張子を付ける必要があります。

キーワード キーワードには，次に挙げるオプションと処理される IDL ソースファイルが含まれています。

オプション idl2java では，次のオプションを使用できます。

表 4.4 idl2java のオプション

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>	インクルードファイルのディレクトリのフルパスまたは相対パスを指定します。インクルードファイルの検索に使用されます。
-P, -no_line_directives	コードを生成するとき，行番号情報の生成を抑止します。デフォルトは off 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。
-compilerflags	Java コンパイラのフラグを指定します。最初の "-" はエスケープされ，コンマで区切られます。
-compiler <full name>	Java コンパイラクラス名の完全名を指定します。
-U, -undefine foo	プリプロセッサマクロ foo の定義を解除します。
-[no_]builtin (TypeCode|Principal)	組み込み型 ::TypeCode または ::Principal を作成します。デフォルトは on です。
-[no_]preprocess	解析前に入力ファイルをプリプロセスします。デフォルトは on です。
-[no_]preprocess_only	プリプロセス後に入力ファイルの解析を停止します。デフォルトは off です。
-[no_]warn_all	すべての警告を同時にオンまたはオフにします。デフォルトは off です。
file1 [file2]...	処理対象となる 1 つ以上のファイル。stdin の場合は "-"。
-[no_]copy_local_values	CORBA メソッドの連結呼び出しを実行するときに値をコピーします。デフォルトは off です。
-sealed <pkg> <dest_pkg>	このパッケージを sealed としてマークします。コードは，dest_pkg またはデフォルトの場所に生成されます。
-no_classloader_aware	クラスローダー対応の Java コードを生成します。デフォルトは on です。
-backcompat_compile	非推奨の jdk1.4.1 コンパイルオプションを使用します。デフォルトは off です。
-[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	プラグマを認識できない場合に警告を表示します。デフォルトは on です。
-[no_]back_compat_mapping	VisiBroker 3.x caffeine コンパイルと下位互換性のあるマッピングを使用します。
-[no_]boa	BOA 互換のコードを生成します。デフォルトは off です。
-[no_]comments	コード内のコメントの生成を抑止します。デフォルトは on です。
-[no_]examples	_example クラスの生成を抑止します。デフォルトは off です。
-gen_included_files	#include ファイルのコードを生成します。デフォルトは off 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。デフォルトは off です。
-[no_]obj_wrapper	オブジェクトラッパーをサポートするファイルを生成します。デフォルトは off です。
-root_dir <path>	生成されたファイルを置くディレクトリを指定します。
-[no_]servant	サーバント（サーバー側）コードを生成します。デフォルトは on です。
-tie	tie クラスを生成します。デフォルトは on です。
-[no_]warn_missing_define	前方参照宣言されたインターフェース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 on です。

表 4.4 idl2java のオプション（続き）

オプション	説明
-[no_]bind	Helper クラスを生成するときに，bind() メソッドの生成を抑止します。デフォルトは off です。
-[no_]compile	オンに設定された場合は，Java ファイルも自動的にコンパイルします。デフォルトは off です。
-dynamic_marshal	DSI/DII モデルを使用してマーシャリングします。デフォルトは off です。
-idl2package <IDL_name> <pkg>	指定された IDL のコンテナ型のデフォルトパッケージを上書きします。
-[no_]invoke_handler	EJB の起動ハンドラクラスを生成します。デフォルトは off です。
-[no_]narrow_compliance	ナローイングについて柔軟にコードを生成します (3.x 互換でなく)。デフォルトは on です。
-[no_]Object_methods	オブジェクトのすべてのメソッドを生成します。デフォルトは on です。
-package <pkg>	生成されるコードのルートパッケージを指定します。
-stream_marshal	ストリームモデルを使用してマーシャリングします。デフォルトは on です。
-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 off です。
-version	Borland VisiBroker のソフトウェアバージョン番号を表示します。
-map_keyword <kwd> <replacement>	使用しないキーワードとその代替キーワードを指定します。
-h, -help, -usage, -?	使用方法を表示します。

java2idl

このコマンドは，Java クラスファイル（Java バイトコード）から IDL を生成します。1 つまたは複数の Java クラス（バイトコード）を入力できます。クラス名を複数入力する場合は，クラス名とクラス名の間に必ずスペースを入れてください。

なんらかの Java リモートインターフェース定義の中で，org.omg.CORBA.IDLEntity を拡張するクラスを使用する場合は，次の準備が必要です。

- その型の IDL 定義を保持する IDL ファイル。org.omg.CORBA.IDLEntity インターフェースは，Java にマップされるすべての IDL データ型を示すシグニチャインターフェースです。
- オブジェクトマネージメントグループ (OMG) の「CORBA 2.6 IDL2Java Specification」に準拠するすべての関連（サポート）クラス。

なんらかの Java リモートインターフェース定義の中で，org.omg.CORBA.IDLEntity を拡張するクラスを使用する場合は，java2idl ツールのコマンドラインで -import <IDL files> 指示文を使用してください。

詳細は，http://www.omg.org/ で「CORBA 2.6 IDL2Java Specification」を参照してください。

メモ このコマンドを使用するには，JDK 1.3 以降をサポートする仮想マシンが必要です。

構文

```
java2idl [options] {filename}
```

例

```
java2idl -o final.idl Account Client Server
```

説明 Java バイトコードから IDL を生成するには，このコマンドを使用します。既存の Java バイトコードから IDL ファイルを作成し，そのファイルを C++，COBOL，Smalltalk などの他のプログラミング言語でも使用する場合に，このコマンドを利用します。

上のサンプルのようにオプション「-o」を使用することで，3 つの Java バイトコードファイル（Account，Client，Server）が final.idl というファイルに出力されます。デフォルトで出力は画面に表示されます。

キーワード キーワードには，次に挙げるオプションと処理される Java バイトコードファイルが含まれています。

オプション `java2idl` では，次のオプションを使用できます。

表 4.5　java2idl のオプション

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>	インクルードファイルのディレクトリのフルパスまたは相対パスを指定します。インクルードファイルの検索に使用されます。
-P, -no_line_directives	コードを生成するとき，行番号情報の生成を抑止します。デフォルトは off 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。
-U, -undefine foo	プリプロセッサマクロ foo 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	プラグマを認識できない場合に警告を表示します。デフォルトは on です。
-[no_]back_compat_mapping	VisiBroker 3.x caffeine コンパイルと下位互換性のあるマッピングを使用します。
-exported <pkg>	指定されたパッケージの型定義がエキスポートされます。
-[no_]export_all	すべてのパッケージの型定義をエキスポートします。デフォルトは off です。
-import <IDL file name>	追加の IDL 定義をロードします。
-imported <pkg> <IDL file name>	指定されたパッケージの型定義は，指定された IDL ファイルからインポートされるとみなす必要があり，コードを生成してはなりません。
-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 off です。
class1 [class2]...	処理対象の 1 つ以上の Java クラス。
-version	Borland VisiBroker のソフトウェアバージョン番号を表示します。
-h, -help, -usage, -?	使用方法を表示します。

java2iiop

このコマンドを使用すると，IDL のかわりに Java 言語を使用して，IDL インターフェースを定義できます。1 つまたは複数の Java クラスファイル（Java バイトコード）の名前を入力できます。クラス名を複数入力する場合は，クラス名とクラス名の間に必ずスペースを入れてください。完全なスコープ付きクラス名を使用します。

メモ このコマンドを使用するには，JDK 1.3 以降をサポートする Java 仮想マシンが必要です。

なんらかの Java リモートインターフェース定義の中で，`org.omg.CORBA.IDLEntity` を拡張するクラスを使用する場合は，次の準備が必要です。

- その型の IDL 定義を保持する IDL ファイル。`org.omg.CORBA.IDLEntity` インターフェースは，Java にマップされるすべての IDL データ型を示すシグニチャインターフェースです。
- オブジェクトマネージメントグループ (OMG) の「CORBA 2.6 IDL2Java Specification」に準拠するすべての関連（サポート）クラス。

なんらかの Java リモートインターフェース定義の中で，`org.omg.CORBA.IDLEntity` を拡張するクラスを使用する場合は，java2iiop ツールのコマンドラインで `-import <IDL files>` 指示文を使用してください。

詳細は，http://www.omg.org/ で「CORBA 2.6 IDL2Java Specification」を参照してください。

構文

```
java2iiop [options] {class name}
```

例

```
java2iiop -no_tie Account Client Server
```

説明 既存の Java バイトコードを書き換えて分散オブジェクトに利用する場合，また IDL の記述を敬遠する場合に，java2iiop を使用します。java2iiop を使用すると，必要なコンテナクラス，クライアントスタブ，およびサーバースケルトンを Java バイトコードから生成できます。

メモ java2iiop コンパイラは，CORBA インターフェースのオーバーロードされたメソッドをサポートしません。

キーワード キーワードには，次に挙げるオプションと処理される Java バイトコードファイルが含まれています。

オプション java2iiop では，次のオプションを使用できます。

表 4.6 java2iiop のオプション

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>	インクルードファイルのディレクトリのフルパスまたは相対パスを指定します。インクルードファイルの検索に使用されます。
-P, -no_line_directives	コードを生成するとき，行番号情報の生成を抑止します。デフォルトは off 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。
-U, -undefine foo	プリプロセッサマクロ foo 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	プラグマを認識できない場合に警告を表示します。デフォルトは on です。
-[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。デフォルトは off です。
-exported <pkg>	エキスポートされるパッケージの名前を指定します。
-[no_]export_all	すべてのパッケージをエキスポートします。デフォルトは off です。
-import <IDL file name>	追加の IDL 定義をロードします。
-imported <pkg> <idl_file_name>	インポートされるパッケージの名前を指定します。
-[no_]boa	BOA 互換のコードを生成します。デフォルトは off です。
-[no_]comments	コード内のコメントの生成を抑止します。デフォルトは on です。
-[no_]examples	_example クラスの生成を抑止します。デフォルトは off です。
-gen_included_files	#include ファイルのコードを生成します。デフォルトは off 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。デフォルトは off です。
-[no_]obj_wrapper	オブジェクトラッパーをサポートするファイルを生成します。デフォルトは off です。
-root_dir <path>	生成されたファイルを置くディレクトリを指定します。
-[no_]servant	サーバント（サーバー側）コードを生成します。デフォルトは on です。
-tie	_tie クラスを生成します。デフォルトは on です。
-[no_]warn_missing_define	前方参照宣言されたファイル名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 on です。
-[no_]bind	Helper クラスを生成するときに，bind() メソッドの生成を抑止します。デフォルトは on です。
-[no_]compile	Java ファイルを自動的に生成します。オンに設定された場合は，Java ファイルも自動的にコンパイルします。デフォルトは off です。
-compiler	使用する Java コンパイラを指定します。-compile オプションが設定されていない場合，このオプションは無視されます。

表 4.6 java2iiop のオプション（続き）

オプション	説明
-compilerflags "\-flag,arg[,.]	Java コンパイラに渡す Java コンパイラのフラグを指定します。最初の "-" はエスケープされ，コンマで区切られます。
-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。
-[no_]builtin (TypeCode| Principal)	組み込み型 ::TypeCode または ::Principal を作成します。デフォルトは on です。
-[no_]preprocess	解析前に入力ファイルをプリプロセスします。デフォルトは on です。
-[no_]preprocess_only	プリプロセス後に入力ファイルの解析を停止します。デフォルトは off です。
-[no_]warn_all	すべての警告を同時にオンまたはオフにします。デフォルトは off です。
-[no_]copy_local_values	CORBA メソッドの連結呼び出しを実行するときに値をコピーします。デフォルトは off です。
-no_classloader_aware	クラスローダー対応の Java コードを生成します。デフォルトは on です。
-backcompat_compile	非推奨の jdk1.4.1 コンパイルオプションを使用します。デフォルトは off です。
-[no_]idlentity_array_mapping	boxedRMI の boxedIDL に IDLEntity の配列をマップします。デフォルトは off です。
class1 [class2]...	処理対象の 1 つ以上の Java クラス。
-dynamic_marshal	DSI/DII モデルを使用してマーシャリングします。デフォルトは off です。
-idl2package <IDL name> <pkg>	指定された IDL のコンテナ型のデフォルトパッケージを上書きします。
-[no_]invoke_handler	EJB の起動ハンドラクラスを生成します。デフォルトは on です。
-[no_]narrow_compliance	柔軟にコードを生成します（3.x 互換でなく）。デフォルトは on です。
-[no_]Object_methods	string や equals など，java.lang.Object メソッドに定義されているすべてのメソッドを生成します。デフォルトは on です。
-package <pkg>	生成されるコードのルートパッケージを指定します。
-sealed <pkg> <destination_pkg>	指定されたパッケージ内のリモートインターフェースに対するスタブとスケルトンを org.omg.stub と目的のパッケージにそれぞれ生成します。
-stream_marshal	ストリームモデルを使用してマーシャリングします。デフォルトは on です。
-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 off です。
-version	Borland VisiBroker のソフトウェアバージョン番号を表示します。
-map_keyword <kwd> <replacement>	使用しないキーワードとその代替キーワードを指定します。
-h, -help, -usage, -?	使用方法を表示します。

vbj

このコマンドは，ローカルの Java インタープリタを起動します。

構文

```
vbj [options] [arguments normally sent to java VM] {class} [arg1 arg2 ...]
```

ここで

引数	説明
{class}	実行されるクラスの名前を指定します。
[arg1 arg2 ...]	そのクラスに渡される引数を指定します。

例

```
vbj Server
```

説明 Java アプリケーションには，他の言語で記述されたアプリケーションにはない一定の制約があります。vbj コマンドは，これらの制約の一部を回避するためのオプションを提供します。Borland VisiBroker アプリケーションを起動する場合は，このコマンドを使用することをお勧めします。vbj コマンドは次の動作を行います。

- コマンドラインオプションと設定ファイル定義に基づいて，CLASSPATH と引数を Java VM に渡します。
- カスタマイズされた設定ファイルを使用して，アプリケーションごとに起動動作をカスタマイズします。
- 起動プログラムと同じプロセスに JVM を埋め込みます。
- アプリケーションをデーモンとして実行します（Windows プラットフォームのみ）。

vbj では，次のオプションを使用できます。

表 4.7 vbj のオプション

引数	説明
`-debug, -VBJdebug`	起動プログラムのデバッグ出力をオンにします。
`-h, -help, -usage, -?`	起動プログラムのコマンドヘルプを出力します。
`-version`	現在実行している Borland VisiBroker for Java のバージョンを表示または出力します。
`-install <server-name>`	Windows NT/2000 サービスをインストールします。
`-remove <server-name>`	Windows NT/2000 サービスを削除します。
`-javahome <jvm-directory>`	Java VM のインストールディレクトリを指定します。

`-classicvm`
`-hotspotvm` / `-clientvm`
`-servervm`
	実行する VM タイプを選択します。-J フラグを使用して VM タイプを渡すこともできます。次に例を示します。
	`vbj -J-server Server`

`-classpath`
`-classpath/a`
`-classpath/p`
`-classpath/r`
`-VBJclasspath`
`-VBJaddJar`
	クラスパスを変更します。この引数の値は，環境内の既存のクラスパス設定の後に追加されるか（/a），前に追加されるか（/p），完全に置き換えられます（/r）。これらのクラスパス指定の最後の引数だけが適用されます。`-VBJclasspath` は `-classpath/p` と同等で，`-VBJaddJar` は `-classpath/a` と同等です。

`-verbose`	Java VM からの詳細出力をオンにします。
`-VBJconfig <config-file-name>`	デフォルトの設定ファイルのかわりに，別の設定ファイルを使用します。

`-jpda[:[{paused|running}]`
`[,address=[<host>:]<port#>]]`
	JPDA デバッグをオンにします。たとえば，次のようになります。
	`-jpda:running,address=23456`
	JPDA をオンにして JVM を開始します。これで，JPDA デバッガは，このアプリケーションにポート 23456 でアタッチして，アプリケーションをデバッグできます。また，起動プログラムの設定ファイル（たとえば，<install-dir>/bin/vbj.config）に次の行を置きます。
	`jpda running,address=23456`

`-javacmd`	同等の Java コマンドを出力します。これは，vbj 起動プログラムが不要で，アプリケーションが Java 起動プログラムから実行される場合に便利です。

vbjc

このコマンドは，VisiBroker のクラスをインポートする Java コードをコンパイルするために使用されます。このコマンドは次の作業を行います。

- コマンドラインオプションと設定ファイル定義に基づいて，Java VM に渡す CLASSPATH と引数を設定します。
- VisiBroker 標準の JAR ファイルを CLASSPATH に追加します。

- javac メインクラス com.sun.tools.javac.Main を起動します。

構文 vbjc [arguments normally passed to javac]

例 vbjc Server.java

vbjc コマンドは，次の表に示すコマンドラインオプションをサポートします。

表 4.8 vbjc のオプション

引数	説明
-VBJdebug	VisiBroker for Java デバッグ情報を表示または出力します。
-VBJversion	現在実行している Borland VisiBroker for Java のバージョンを表示または出力します。
-VBJjavavm <vmname>	使用する Java 仮想マシンのパスを指定します。デフォルトは java です。
-VBJclasspath <classpath>	クラスパスを指定します。CLASSPATH 環境変数より優先します。
-VBJaddJar <jarfile>	VM を実行する前に，<install-dir>/lib/<jarfile> を CLASSPATH に追加します。絶対パスが指定されていない場合，この jarfile は <launcher-location>/../lib からの相対パスとみなされます。
-VBJconfig <config-file-name>	起動プログラムによって使用される設定ファイルのパス。指定しない場合，デフォルトの場所は <install-dir>/bin/vbj.config|vbjc.config になります。
-help|-h|-?|-usage	コマンドの使用方法を表示します。
-VBJcompiler <class-name>	デフォルトの javac メインクラスを上書きします。

クラスパスの指定

次の情報がこの順に結合されます。

1. patches ディレクトリ（$VBROKERDIR/lib/patches/）内の JAR と ZIP ファイル。patches ディレクトリは，$VBROKERDIR/lib/ ディレクトリの下に自動的には作成されません。ユーザーが明示的に作成する必要があります。
2. -VBJclasspath，-classpath/p，または -classpath/r に指定されたクラスパス
3. 環境にエキスポートされた $CLASSPATH（-classpath/r が指定されていない場合）
4. -classpath/a に指定されたクラスパス
5. 起動プログラムに必要なデフォルトの JAR ファイル
6. VBJaddJar を使用して追加された JAR ファイル。絶対パスが指定されていない場合は，< 起動プログラムの場所 >/../lib ディレクトリにあるとみなされます。
7. 設定ファイルで addpath ディレクティブを使用して追加されたクラスパス
8. 設定ファイルで addjars ディレクティブを使用して追加された JAR ファイル
9. 現在のディレクトリ

マージされたクラスパスは，-Djava.class.path を使用して Java 仮想マシンに渡されます。

JVM の指定

デフォルトでは，JVM は次のように検索されます。

1. PATH で指定されたディレクトリを検索します。
2. 設定ファイルの javahome ディレクティブで指定された情報を使用します（vbj のデフォルトの設定ファイルは vbj.config）。

上の手順は，-VBJjavavm オプションまたは -javahome オプション（vbj のみ）を使用して上書きできます。-VBJjavavm では，VM の名前を指定するか，VM のフルパスを指定できます。オプション -javahome は，javahome 設定ファイルディレクティブと同じセマンティ

クスを持ちます。-VBJjavavm または -javahome オプションを使用しても VM が見つからない場合，デフォルトの JVM を検索するためにそれ以上の検索は行われず，プログラムはエラーになって終了します。

idl2wsj

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。
-wsdl_file_name	生成される WSDL ファイルの名前。デフォルトは IDL の名前です。
-wsdl_namespace	生成される WSDL の名前空間。デフォルトは IDL ファイルの名前です。
-gen_java_bridge	VisiBroker for Java ブリッジコードを生成します。デフォルトは OFF です。
-root_dir	生成ファイルが常駐するディレクトリ。

第 5 章

IDL から Java へのマッピング

ここでは，VisiBroker for Java の idl2java コンパイラによって実装されている現行の IDL ／ Java 言語マッピングの基本について説明します。VisiBroker for Java は，『OMG IDL/Java Language Mapping Specification』(OMG IDL ／ Java 言語マッピング仕様) に準拠しています。

次の事項に関する詳細な情報については，最新の OMG IDL ／ Java 言語マッピング仕様を参照してください。

- 擬似オブジェクトから Java へのマッピング
- サーバー側のマッピング
- Java ORB の可搬性のあるインターフェース

名前

一般に，IDL の名前と識別子は，そのまま Java の名前と識別子にマッピングされます。

マッピング先の Java コードで名前の競合が発生する場合は，マッピング先の名前の前にアンダースコア（_）を付加して，名前の競合が解決されます。

さらに，Java 言語の性質上，1 つの IDL の構成が複数の名前の異なる Java の構成にマッピングされる場合があります。追加して作成される名前には，その性質を説明するサフィックスが付きます。たとえば，IDL インターフェース AccountManager は，Java インターフェース AccountManager と追加 Java クラス AccountManagerOperations，AccountManagerHelper，および AccountManagerHolder にマッピングされています。

追加された名前がマッピング元のほかの IDL 名と競合するような例外的な場合には，その IDL 名の方に上記の解決規則が適用されます。つまり，名前を追加して使用する必要がある場合は，追加される名前の命名規則の方が優先されます。

たとえば fooHelper または fooHolder という名前のインターフェースは，foo という名前のインターフェースが存在するかどうかに関係なく，それぞれ _fooHelper または _fooHolder にマッピングされます。インターフェース fooHelper のヘルパークラスとホルダークラスの名前は，_fooHelperHelper と _fooHelperHolder になります。

Java の識別子にそのままマッピングされると，Java の予約語と競合してしまう IDL 名には，競合の解決規則が適用されます。

予約名

マッピングでは，それぞれ一定の目的で使用されるいくつかの名前が予約されています。それらの予約名をユーザー定義の IDL の型またはインターフェースに使用すると（それが IDL の名前としても正しいとして），先頭にアンダースコア（_）が付加されてマッピングされます。予約名は次のとおりです。

- Java クラス `<type>Helper`。ここで，`<type>` は IDL ユーザー定義型の名前です。
- Java クラス `<type>Holder`。ここで，`<type>` は IDL ユーザー定義型の名前です。ただし，`typedef` のような例外もあります。
- Java クラス `<basicJavaType>Holder`。ここで，`<basicJavaType>` は，IDL 基本データ型のいずれかによって使用される Java プリミティブデータ型のいずれかです。
- ネストされたスコープ付きの Java パッケージ名 `<interface>Package`。ここで，`<interface>` は IDL インターフェース名です。
- Java クラス `<interface> Operations`，`<interfaces> POA`，および `<interface>POATie`。ここで，<interface> は IDL インターフェース型の名前です。

予約語

マッピングでは，それぞれ一定の目的で使用されるいくつかの名前が予約されています。それらの予約名をユーザー定義の IDL の型またはインターフェースに使用すると（それが IDL の名前としても正しいとして），先頭にアンダースコア（_）が付加されてマッピングされます。Java 言語の予約キーワードは次のとおりです。

abstract	abstractBase	boolean	break
byte	case	catch	char
class	const	continue	default
do	double	else	extends
false	final	finally	float
for	goto	if	implements
import	instanceof	int	interface
local	long	native	new
null	package	private	protected
public	return	short	static
super	switch	synchronized	this
throw	throws	transient	true
try	void	volatile	while

モジュール

1 つの IDL モジュールは，同じ名前の 1 つの Java パッケージにマッピングされます。モジュール内のすべての IDL 型宣言は，生成されるパッケージ内で，対応する Java クラスまたはインターフェースの宣言にマッピングされます。

どのモジュールにも属さない IDL 宣言は，名前のない Java グローバルスコープにマッピングされます。

次のサンプルコードに，ある IDL モジュールの宣言に対して生成された Java コードを示します。

```
/* Example.idl より */
module Example { .... };
// 生成された Java
package Example;
. . .
```

基本型

次の表に，定義済みの IDL 型と Java 基本型とのマッピングを示します。

表 5.1　基本型のマッピング

IDL 型	Java 型
boolean	boolean
char	char
wchar	char
octet	byte
string	java.lang.String
wstring	java.lang.String
short	short
unsigned short	short
long	int
unsigned long	int
longlong	long
unsigned longlong	long
float	float
double	double

IDL の型とマッピング先の Java の型の間に不一致があった場合は，標準の CORBA 例外が生成されます。ほとんどの例外の原因は，次の 2 つに分類できます。

- Java の型の範囲が IDL の型の範囲より広いため。たとえば，Java の char は IDL の char の範囲を包含します。
- Java が符号なしの型をサポートしないため。符号なしの IDL 型の大きな値が Java の負の整数になっても正しく処理されるように開発者が配慮する必要があります。

以下の節で，追加の詳細情報について説明します。

IDL 拡張型

ここでは，VisiBroker for Java による IDL 型拡張のサポートについてまとめます。最初の表は早見表です。その次の 42 ページの表 5.3「新しい IDL 拡張型」で，新しい型のサポートについて説明します。

表 5.2　サポートされる IDL 拡張型の概要

型	Borland VisiBroker によるサポート
longlong	はい
unsigned longlong	はい
long double	いいえ [1]
wchar	はい [2]
wstring	はい [2]
fixed	いいえ [1]

[1]VisiBroker for Java は，OMG 標準インプリメンテーションの将来のリリースをサポートする予定です。

[2] ネットワークでは Unicode が使用されます。

表 5.3　新しい IDL 拡張型

新しい型	説明
longlong	64 ビットの符号付き整数（2 の補数表現）
unsigned longlong	64 ビットの符号なし整数（2 の補数表現）
long double	IEEE 標準 754-1985 の倍精度拡張浮動小数点
wchar	ワイド文字
wstring	ワイド文字列
fixed	固定小数点 10 進数（有効桁数 31）

Holder クラス

Holder クラスは，OUT および INOUT のパラメータ受け渡しモードをサポートし，org.omg.CORBA package パッケージ内のすべての IDL 基本データ型に使用できます。Holder クラスは，typedefs で定義された型を除き，すべての名前付きユーザー定義型に対して生成されます。詳細については，Java API リファレンスの VisiBroker API で，org.omg.CORBA パッケージを参照してください。

IDL ユーザー定義型の場合は，その型に対応する Java の名前の末尾に Holder を付けて，ホルダークラスの名前が作成されます。

IDL 基本型の場合は，そのデータ型に対応する Java の型の（先頭の文字を大文字にした）名前の末尾に Holder を付けて，ホルダークラスの名前が作成されます。たとえば，IntHolder です。

各ホルダークラスは，1 つのデフォルトコンストラクタ，および public インスタンスメンバー value を持ちます。このメンバーは型付きの値です。デフォルトコンストラクタは，value 値フィールドをその型のデフォルト値に設定します。デフォルト値は Java 言語で定義されています。

- boolean の場合は false
- 値の場合は null
- 数値と char 型の場合は 0
- 文字列の場合は null
- オブジェクトリファレンスの場合は null

可搬性のあるスタブとスケルトンをサポートするため，ユーザー定義型の Holder クラスは，org.omg.CORBA.portable.Streamable インターフェースも実装します。

次のサンプルコードに，基本型のホルダークラスの定義を示します。これらのサンプルは，org.omg.CORBA パッケージにあります。

```
// Java
package org.omg.CORBA;
final public class ShortHolder implements Streamable {
   public short value;
   public ShortHolder() {}
   public ShortHolder(short initial) {
      value = initial;
   }
   ...//Streamable インターフェースのインプリメンテーション
}
final public class IntHolder implements Streamable {
   public int value;
   public IntHolder() {}
```

```
   public IntHolder(int initial) {
      value = initial;
   }
   ...//Streamable インターフェースのインプリメンテーション
}
final public class LongHolder implements Streamable {
   public long value;
   public LongHolder() {}
   public LongHolder(long initial) {
      value = initial;
   }
   ...//Streamable インターフェースのインプリメンテーション
}
final public class ByteHolder implements Streamable {
   public byte value;
   public ByteHolder() {}
   public ByteHolder(byte initial) {
      value = initial;
   }
   ...//Streamable インターフェースのインプリメンテーション
}
final public class FloatHolder implements Streamable {
   public float value;
   public FloatHolder() {}
   public FloatHolder(float initial) {
      value = initial;
   }
   ...//Streamable インターフェースのインプリメンテーション
}
final public class DoubleHolder implements Streamable {
   public double value;
   public DoubleHolder() {}
   public DoubleHolder(double initial) {
      value = initial;
   }
   ...//Streamable インターフェースのインプリメンテーション
}
final public class CharHolder implements Streamable {
   public char value;
   public CharHolder() {}
   public CharHolder(char initial) {
      value = initial;
   }
   ...//Streamable インターフェースのインプリメンテーション
}
final public class BooleanHolder implements Streamable {
   public boolean value;
   public BooleanHolder() {}
   public BooleanHolder(boolean initial) {
      value = initial;
   }
   ...//Streamable インターフェースのインプリメンテーション
}
final public class StringHolder implements Streamable {
   public java.lang.String value;
   public StringHolder() {}
   public StringHolder(java.lang.String initial) {
      value = initial;
   }
   ...//Streamable インターフェースのインプリメンテーション
}
final public class ObjectHolder implements Streamable {
   public org.omg.CORBA.Object value;
```

```
   public ObjectHolder() {}
   public ObjectHolder(org.omg.CORBA.Object initial) {
      value = initial;
   }
   ...//Streamable インターフェースのインプリメンテーション
}
final public class ValueBaseHolder implements Streamable {
   public java.io.Serializable value;
   public ValueBaseHolder() {}
   public ValueBaseHolder(java.io.Serializable initial) {
      value = initial;
   }
   ...//Streamable インターフェースのインプリメンテーション
}
final public class AnyHolder implements Streamable {
   public Any value;
   public AnyHolder() {}
   public AnyHolder(Any initial) {
      value = initial;
   }
   ...//Streamable インターフェースのインプリメンテーション
}
final public class TypeCodeHolder implements Streamable {
   public TypeCode value;
   public typeCodeHolder() {}
   public TypeCodeHolder(TypeCode initial) {
      value = initial;
   }
   ...//Streamable インターフェースのインプリメンテーション
}
final public class PrincipalHolder implements Streamable {
   public Principal value;
   public PrincipalHolder() {}
   public PrincipalHolder(Principal initial) {
      value = initial;
   }
   ...//Streamable インターフェースのインプリメンテーション
}
```

次のサンプルコードは，ユーザー定義型の Holder クラス `<foo>` を示しています。

```
// Java
final public class <foo>Holder
   implements org.omg.CORBA.portable.Streamable {
   public <foo> value;
   public <foo>Holder() {}
   public <foo>Holder(<foo> initial) {}
   public void _read(org.omg.CORBA.portable.InputStream i)
      {...}
   public void _write(org.omg.CORBA.portable.OutputStream o)
      {...}
   public org.omg.CORBA.TypeCode _type() {...}
   }
```

Java の null

Java の `null` は，null の CORBA オブジェクトリファレンスと null の valuetype（再帰的な valuetype を含む）を表現するためにだけ使用されます。たとえば，空文字列を表すには，`null` ではなく，長さが 0 の文字列を使用する必要があります。これは，valuetype を除く，配列とすべての構造型にも当てはまります。構造体として `null` を渡そうとすると，`NullPointerException` が生成されます。

Boolean

IDL 型 boolean は，Java 型 boolean にマッピングされます。IDL 定数 TRUE と FALSE は，Java 定数 true と false にマッピングされます。

Char

IDL の char が 8 ビットで文字セットの要素を表現するのに対して，Java の char は符号なしの 16 ビットで Unicode 文字を表現します。タイプセーフを確保するため，メソッドの呼び出しでパラメータがマーシャリングされるとき，Java CORBA ランタイムは，IDL の char からマッピングされたすべての Java の char の範囲の有効性を確認します。char が文字セットに定義されている範囲内にない場合は，CORBA::DATA_CONVERSION 例外が生成されます。

IDL の wchar は，Java の char 型にマッピングされます。

Octet

IDL 型 octet は 8 ビットで，Java 型 byte にマッピングされます。

String

IDL の string 型は，固定長と可変長のどちらのバリアントも，Java の java.lang.String 型にマッピングされます。マーシャリング時に，文字列の長さのチェックだけでなく，文字列内の文字の範囲のチェックも行われます。

WString

Unicode 文字列の表現に使用される IDL の wstring 型は，Java の java.lang.String 型にマッピングされます。マーシャリング時に，文字列の長さがチェックされます。

整数型

IDL short と unsigned short は，Java 型 short にマッピングされます。IDL long と unsigned long は，Java 型 int にマッピングされます。

メモ　Java では符号なしの型がサポートされないので，符号なしの大きな値が Java の負の整数として正しく処理されるようにしてください。

浮動小数点型

IDL の浮動小数点数 float と double は，それらに対応するデータ型を保持する Java クラスにマッピングされます。

Helper クラス

どの IDL ユーザー定義型にも，生成された型名にサフィックス Helper が付いた追加ヘルパー Java クラスが生成されます。その型の操作に必要な static メソッドがいくつか提供されます。

- その型を Any として挿入または抽出するメソッド
- リポジトリ ID を取得するメソッド

- タイプコードを取得するメソッド
- その型をストリームから読み取るおよびストリームへ書き込むメソッド

<typename> という名前の IDL ユーザー定義型に対して, 次の Java コードが生成されます。IDL インターフェースからマッピングされたヘルパークラスには, narrow オペレーションがあります。

```
// 生成された Java ヘルパー
public class <typename>Helper {
   public static void insert(org.omg.CORBA.Any a, <typename> t);
   public static <typename> extract(org.omg.CORBA.Any a);
   public static org.omg.CORBA.TypeCode type();
   public static String id();
   public static <typename> read( org.omg.CORBA.portable.InputStream istream);
         {...}
   public static void write(
            org.omg.CORBA.portable.OutputStream ostream, <typename> value)
         {...}
   // インターフェースのヘルパーのみ
   public static <typename> narrow(org.omg.CORBA.Object obj);
```

次のサンプルコードは, named type から Java ヘルパークラスへのマッピングを示しています。

```
// IDL - 名前付きの型
struct st {long f1, String f2};
// 生成された Java
public class stHelper {
   public static void insert(org.omg.CORBA.Any any,
      st s) {...}
   public static st extract(org.omg.CORBA.Any a) {...}
   public static org.omg.CORBA.TypeCode type() {...}
   public static String id() {...}
   public static st read(org.omg.CORBA.InputStream is) {...}
   public static void write(org.omg.CORBA.OutputStream os,
       st s) {...}
}
```

次のサンプルコードは, typedef sequence から Java ヘルパークラスへのマッピングを示しています。

```
// IDL - typedef シーケンス
typedef sequence <long> IntSeq;
// 生成された Java ヘルパー
public class IntSeqHelper {
   public static void insert(org.omg.CORBA.Any any,
                           int[] seq);
   public static int[] extract(org.omg.CORBA.Any a){...}
   public static org.omg.CORBA.TypeCode type(){...}
   public static String id(){...}
   public static int[] read(
                  org.omg.CORBA.portable.InputStream is)
      {...}
   public static void write(
                  org.omg.CORBA.portable.OutputStream os,
                  int[] seq)
      {...}
}
```

定数

定数のマッピングは, その定数が持つスコープに依存します。

インターフェース内の定数

IDL インターフェース内の定数は，その IDL インターフェースに対応する Java インターフェース Operations クラス内の public static final フィールドにマッピングされます。

次のサンプルコードは，モジュール内の IDL 定数から Java クラスへのマッピングを示しています。

```
/* Example.idl より */
module Example {
   interface Foo {
      const long aLongerOne = -321;
   };
};

// Foo.java
package Example;
public interface Foo extends com.borland.vbroker.CORBA.Object,
      Example.FooOperations,
      org.omg.CORBA.portable.IDLEntity {
}
// FooOperations.java
package Example;
public interface FooOperations {
   public final static int aLongerOne = (int)-321;
}
```

インターフェース内にない定数

IDL モジュール内で宣言された定数は, value という名前の public static final フィールドを持ち，その定数と同じ名前の public インターフェースにマッピングされます。そのフィールドが定数値を保持します。

メモ そのクラスがほかの Java コードで使用される場合，通常，Java コンパイラはその値をインラインします。

次のサンプルコードは，モジュール内の IDL 定数から Java クラスへのマッピングを示しています。

```
/* Example.idl より */
module Example {
   const long aLongOne = -123;
};
// 生成された Java
package Example;
public interface aLongOne {
   public final static int value = (int) -123;
}
```

構造型

IDL 構造型には，enum，struct，union，sequence，および array があります。型 sequence と array は，ともに Java array 型にマッピングされます。IDL の構造型 enum，struct，および union は，その IDL 型のセマンティクスを実装する Java クラスにマッピングされます。生成される Java クラスの名前は，元の IDL 型と同じです。

列挙体

IDL の enum は，同じ名前の Java final class にマッピングされます。このクラスは，value メソッド，1 つのラベルごとに 2 つの static データメンバー，整数変換メソッド，および

private コンストラクタを宣言します。次のサンプルコードは，IDL enum から Java final クラスへのマッピングを示しています。

```
// 生成された Java
public final class <enum_name> {
   //enum 内のラベルごとに 1 組
   public static final int _<label> = <value>;
   public static final <enum_name> <label> =
      new <enum_name>(_<label>);

   public int value() {...}
   // 指定された値から列挙値を取得します。
   public static <enum_name> from_int(int value);
   // コンストラクタ
   protected <enum_name>(int) {...}
}
```

一方は，IDL の enum ラベルと同じ名前を持つ public static final メンバーです。もう一方は，先頭にアンダースコア（_）が付き，switch ステートメントで使用されます。

value メソッドは整数値を返します。値は 0 から始まって順に割り当てられます。enum のラベル名が value でも，Java の value() メソッドと競合することはありません。

enum のインスタンスは 1 つだけです。そのため，ポインタ等価テストは正しく機能します。つまり，java.lang.Object による equals() および hash() のインプリメンテーションは，デフォルトのまま列挙体のシングルトンオブジェクトについても正しく動作します。

enum の Java クラスは，追加のメソッド from_int() を持ちます。このメソッドは，指定された値の enum を返します。

enum には，ホルダークラスも生成されます。次のサンプルにあるように，ホルダークラスの名前は，列挙体をマッピングした Java クラスの名前の先頭に Holder を付けたものです。

```
public class <enum_name>Holder implements
      org.omg.CORBA.portable.Streamable {
   public <enum_name> value;
   public <enum_name>Holder() {}
   public <enum_name>Holder(<enum_name> initial) {...}
   public void _read(org.omg.CORBA.portable.InputStream i)
      {...}
   public void _write(org.omg.CORBA.portable.OutputStream o)
      {...}
   public org.omg.CORBA.TypeCode _type() {...}
}
```

このサンプルコードは，enum に対する IDL から Java へのマッピングを示しています。

```
// IDL

module Example {
   enum EnumType { first, second, third };
};
// 生成された Java
public final class EnumType
      implements org.omg.CORBA.portable.IDLEntity {
   public static final int _first = 0;
   public static final int _second = 1;
   public static final int _third = 2;
   public static final EnumType first = new EnumType(_first);
   public static final EnumType second = new EnumType(_second);
   public static final EnumType third = new EnumType(_third);
   protected EnumType (final int _vis_value) { . . . }
   public int value () { . . . }
   public static EnumType from_int (final int _vis_value) { . . . }
   public java.lang.String toString() { . . . }
}
public final class EnumTypeHolder
      implements org.omg.CORBA.portable.Streamable {
   public OtherExample.EnumType value;
```

```
    public EnumTypeHolder () { . . . }
    public EnumTypeHolder (final OtherExample.EnumType _vis_value) { . . . }
    public void _read (final org.omg.CORBA.portable.InputStream input) { . . . }
    public void _write (final org.omg.CORBA.portable.OutputStream output) { . .
 . }
    public org.omg.CORBA.TypeCode _type () { . . . }
    public boolean equals (java.lang.Object o) {. . .}
 }
```

構造体

IDL の struct は，同じ名前の Java final クラスにマッピングされます。このクラスは，IDL で指定されたフィールドに対応するインスタンス変数，およびすべての値のコンストラクタを提供します。構造体のフィールドを後で初期化するための null コンストラクタも提供されます。struct には，Holder クラスも生成されます。次のサンプルにあるように，ホルダークラスの名前は struct をマッピングした Java クラスの名前の先頭に Holder を付けたものです。

```
final public class <class>Holder implements
      org.omg.CORBA.portable.Streamable {
   public <class> value;
   public <class>Holder() {}
   public <class>Holder(<class> initial) {...}
   public void _read(org.omg.CORBA.portable.InputStream i)
      {...}
   public void _write(org.omg.CORBA.portable.OutputStream o)
      {...}
   public org.omg.CORBA.TypeCode _type() {...}
}
```

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

```
/* Example.idl より */
module Example {
   struct StructType {
      long field1;
      string field2;
   };

};
// 生成された Java
public final class StructType
      implements org.omg.CORBA.portable.IDLEntity {
   public int field1;
   public java.lang.String field2;
   public StructType () { . . . }
   public StructType (final int field1,
   final java.lang.String field2) { . . . }
   public java.lang.String toString() { . . . }
   public boolean equals (java.lang.Object o) {...}
public final class StructTypeHolder implements
org.omg.CORBA.portable.Streamable {
   public Example.StructType value;
   public StructTypeHolder () { . . . }
   public StructTypeHolder (final Example.StructType _vis_value)
      { . . . }
   public void _read (final org.omg.CORBA.portable.InputStream input)
      { . . . }
   public void _write (final org.omg.CORBA.portable.OutputStream output)
      { . . . }
   public org.omg.CORBA.TypeCode _type () { . . . }
}
```

共用体

IDL の union は，同じ名前の Java final クラスにマッピングされます。このクラスは次の要素を提供します。

- デフォルトコンストラクタ
- 共用体のディスクリミネータに対する discriminator() という名前のアクセッサメソッド
- ブランチごとのアクセッサメソッド
- ブランチごとのモディファイアメソッド
- 複数のケースラベルを持つブランチごとのモディファイアメソッド
- 必要に応じて，デフォルトのモディファイアメソッド

マッピングされた共用体またはフィールドの名前に競合がある場合は，通常の名前の競合の解決規則が使用されます。つまり，ディスクリミネータにアンダースコアが付きます。

ブランチアクセッサメソッドとモディファイアメソッドはオーバーロードされ，ブランチと同じ名前が付けられます。必要なブランチが設定されていない場合，アクセッサメソッドは CORBA::BAD_OPERATION 例外を生成します。

1 つのブランチと一致するケースラベルが複数ある場合は，そのブランチの単純なモディファイアメソッドがディスクリミナントを最初のケースラベルの値に設定します。さらに，明示的なディスクリミネータパラメータを受け取る追加のモディファイアメソッドが生成されます。

ブランチが default ケースラベルと一致する場合，モディファイアメソッドは，ディスクリミナントをほかのどのケースラベルとも一致しない値に設定します。

ディスクリミナントのすべての値に対応するケースラベルが完全に揃っている場合に，default ケースラベルを持つ共用体を指定することは不正です。Java コードジェネレータ（IDL コンパイラなどのツール）はこれを検出し，不正なコードを生成しません。

明示的な default ケースラベルがなく，ケースラベルがディスクリミナントの値に完全には対応していない場合は，デフォルトのメソッド _default() が作成されます。このメソッドは，共用体の値を範囲外の値に設定します。

共用体には，ホルダークラスも生成されます。次のサンプルにあるように，ホルダークラスの名前は，共用体をマッピングした Java クラスの名前の先頭に Holder を付けたものです。

このサンプルコードは，union に対する IDL から Java へのマッピングを示しています。

```
final public class <union_class>Holder
        implements org.omg.CORBA.portable.Streamable {
   public <union_class> value;
   public <union_class>Holder() {}
   public <union_class>Holder(<union_class> initial) {...}
   public void _read(org.omg.CORBA.portable.InputStream i)
      {...}
   public void _write(org.omg.CORBA.portable.OutputStream o)
      {...}
   public org.omg.CORBA.TypeCode _type() {...}
}
```

次のサンプルコードは，IDL union から Java へのマッピングを示しています。

```
/* Example.idl より */
module Example {
   enum EnumType { first, second, third, fourth, fifth, sixth };
   union UnionType switch (EnumType) {
      case first: long win;
      case second: short place;
      case third:
      case fourth: octet show;
      default: boolean other;
      };
```

```
};
// 生成された Java
final public class UnionType {
   // コンストラクタ
   public UnionType() {...}
   // ディスクリミネータのアクセッサ
   public int discriminator() { ... }
   //win
   public int       win() { ... }
   public void      win(int value) { ... }
   //place
   public short place() { ... }
   public void place(short value) { ... }
   // 表示
   public byte show() { ... }
   public void    show(byte value) { ... }
   public void          show(int discriminator, byte value) { ... }
   //other
   public boolean          other() {...}
   public void       other(boolean value) { ... }
   public java.lang.String to String () { . . .}
   public boolean equals (java.lang.Object o) { . . .}
}
final public class UnionTypeHolder {
      implements org.omg.CORBA.portable.Streamable {
   public UnionType value;
   public UnionTypeHolder() {}
   public UnionTypeHolder(UnionType initial) {...}
   public void _read(org.omg.CORBA.portable.InputStream i)
      {...}
   public void _write(org.omg.CORBA.portable.OutputStream o)
      {...}
   public org.omg.CORBA.TypeCode_type() {...}
}
```

シーケンス

1 つの IDL `sequence` は，同じ名前の 1 つの Java 配列にマッピングされます。このマッピングでは，`sequence` 型が必要なところで，シーケンスの要素をマッピングした型の配列が 1 つ使用されます。

シーケンスには，ホルダークラスも生成されます。次のサンプルにあるように，ホルダークラスの名前は，シーケンスをマッピングした Java クラスの名前の先頭に `Holder` を付けたものです。

```
final public class <sequence_class>Holder {
   public <sequence_element_type>[] value;
   public <sequence_class>Holder() {};
   public <sequence_class>Holder(
               <sequence_element_type>[] initial) {...};
   public void _read(org.omg.CORBA.portable.InputStream i)
      {...}
   public void _write(org.omg.CORBA.portable.OutputStream o)
      {...}
   public org.omg.CORBA.TypeCode _type() {...}
}
```

次のサンプルコードは，IDL シーケンスから Java へのマッピングを示しています。

```
// IDL
typedef sequence<long>UnboundedData;
typedef sequence<long, 42>BoundedData;

// 生成された Java
final public class UnboundedDataHolder
      implements org.omg.CORBA.portable.Streamable {
```

```
   public int[] value;
   public UnboundedDataHolder() {};
   public UnboundedDataHolder(final int[] initial) { . . . };
   public void _read(org.omg.CORBA.portable.InputStream i)
      { . . . }
   public void _write(org.omg.CORBA.portable.OutputStream o)
      { . . . }
   public org.omg.CORBA.TypeCode _type() { . . . }
}
final public class BoundedDataHolder
      implements org.omg.CORBA.portable.Streamable {
   public int[] value;
   public BoundedDataHolder() {};
   public BoundedDataHolder(final int[] initial) { . . . };
   public void _read(org.omg.CORBA.portable.InputStream i)
      { . . . }
   public void _write(org.omg.CORBA.portable.OutputStream o)
      { . . . }
   public org.omg.CORBA.TypeCode _type() { . . . }
}
```

配列

IDL の配列は，IDL の固定長シーケンスと同じ方法でマッピングされます。このマッピングでは，配列型が必要なところで，配列要素をマッピングした型の配列が 1 つ使用されます。Java では，Java 添え字演算子がマッピングされた配列に適用されます。IDL 定数で配列の範囲を指定することにより，その配列の長さを Java でも使用できます。この定数は，定数の規則にしたがってマッピングされます。

配列には，ホルダークラスも生成されます。次のサンプルにあるように，ホルダークラスの名前は，配列をマッピングした Java クラスの名前の先頭に `Holder` を付けたものです。

```
final public class <array_class>Holder
      implements org.omg.CORBA.portable.Streamable {
   public <array_element_type>[] value;
   public <array_class>Holder() {}
   public <array_class>Holder(
               <array_element_type>[] initial) {...}
   public void _read(org.omg.CORBA.portable.InputStream i)
      {...}
   public void _write(org.omg.CORBA.portable.OutputStream o)
      {...}
   public org.omg.CORBA.TypeCode _type() {...}
}
```

次のサンプルコードは配列のマッピングを示しています。

```
// IDL
const long ArrayBound = 42;
typedef long larray[ArrayBound];
// 生成された Java
final public class larrayHolder
         implements org.omg.CORBA.portable.Streamable {
   public int[] value;
   public larrayHolder() {}
   public larrayHolder(int[] initial) {...}
   public void _read(org.omg.CORBA.portable.InputStream i)
      {...}
   public void _write(org.omg.CORBA.portable.OutputStream o)
      {...}
   public org.omg.CORBA.TypeCode_type() {...}
}
```

インターフェース

IDL インターフェースは，次の 2 つの public Java インターフェースにマッピングされます。

- IDL インターフェースで宣言されているオペレーションと定数を保持する Operations インターフェース。
- すべての基本インターフェースオペレーション, このインターフェースオペレーション, および org.omg.CORBA.object を拡張する CORBA Object 宣言。

さらに，インターフェース名の末尾にサフィックス Helper が付加されたヘルパー Java クラスがあります。この Java インターフェースは，マッピングされたベース org.omg.CORBA.Object インターフェースを拡張します。

Java インターフェースには，マッピングされたオペレーションのシグニチャがあります。メソッドは，このインターフェースのオブジェクトリファレンスを使って起動できます。

ヘルパークラスは，org.omg.CORBA.Object のインスタンスをより具体的な型のオブジェクトリファレンスにナローイングするための静的なナローメソッドを宣言します。そのオブジェクトリファレンスが要求された型をサポートしていないためにナローイングに失敗した場合は，IDL 例外 CORBA::BAD_PARAM 例外が生成されます。その他のエラーの場合は，別のシステム例外が生成されます。null のナローイングは常に成功し, null 値が返されます。

特殊な「nil」オブジェクトリファレンスはありません。オブジェクトリファレンスを渡すところでは，Java の null を自由に渡すことができます。

属性は，1 つの Java アクセッサメソッドとモディファイアメソッドの組にマッピングされます。これらのメソッドは IDL の属性と同じ名前を持ち，オーバーロードされます。IDL の読み取り専用の属性には，モディファイアメソッドがありません。

インターフェースには，ホルダークラスも生成されます。次のサンプルにあるように，ホルダークラスの名前は，インターフェースをマッピングした Java クラスの名前の先頭に Holder を付けたものです。

```
final public class <interface_class>Holder
      implements org.omg.CORBA.portable.Streamable {
   public <interface_class> value;
   public <interface_class>Holder() {}
   public <interface_class>Holder(
               <interface_class> initial) {
      value = initial;

   public void _read(org.omg.CORBA.portable.InputStream i)
      {...}
   public void _write(org.omg.CORBA.portable.OutputStream o)
      {...}
   public org.omg.CORBA.TypeCode _type() {...}
}
```

次のサンプルコードは，IDL インターフェースから Java へのマッピングを示しています。

```
/* Example.idl より */
module Example {
   interface Foo {
      long method(in long arg) raises(AnException);
      attribute long assignable;
      readonly attribute long nonassignable;
   };
};
// 生成された Java
package Example;
public interface Foo extends com.borland.vbroker.CORBA.Object,
      Example.FooOperations,
      org.omg.CORBA.portable.IDLEntity {
}
public interface FooOperations {
   public int method (int arg) throws Example.AnException;
```

```
      public int assignable ();
      public void assignable (int assignable);
      public int nonassignable ();
   }
   public final class FooHelper {
      // ... その他の標準メソッド
      public static Foo narrow(org.omg.CORBA.Object obj)
         { . . . }
      public static Example.Foo bind (org.omg.CORBA.ORB orb,
            java.lang.String name,
            java.lang.String host,
            com.borland.vbroker.CORBA.BindOptions _options) { . . . }
      public static Example.Foo bind (org.omg.CORBA.ORB orb,
            java.lang.String fullPoaName, byte[] oid) { . . . }
      public static Example.Foo bind (org.omg.CORBA.ORB orb,
            java.lang.String fullPoaName, byte[] oid,
            java.lang.String host,
            com.borland.vbroker.CORBA.BindOptions _options) { . . . }

      public Foo read (org.omg.CORBA.portable.InputStream in) { . . . }
      public void write (org.omg.CORBA.portable.OutputStream out, Foo foo) { . . .
   }
   public Foo extract (org.omg.CORBA.Any any) { . . . }
   public void insert (org.omg.CORBA.Any any, Foo foo) { . . . }
   }
   public final class FooHolder
         implements org.omg.CORBA.portable.Streamable {
      public Foo value;
      public FooHolder() {}
      public FooHolder(final Foo initial) { . . . }
      public void _read(org.omg.CORBA.portable.InputStream i)
         { . . . }
      public void _write(org.omg.CORBA.portable.OutputStream o)
         { . . . }
      public org.omg.CORBA.TypeCode_type() { . . . }
   }
```

抽象インターフェース

IDL 抽象インターフェースは，単一の public Java インターフェースへ IDL インターフェースと同じ名前でマッピングされます。マッピングの規則は，IDL 非抽象インターフェースで Java オペレーションインターフェースを生成するのと類似しています。ただし，このインターフェースはシグニチャインターフェースとして動作するため，`org.omg.CORBA.protable.IDLEntity` を拡張できます。マッピングされた Java インターフェースは IDL インターフェースと同じ名前を持ち，指定タイプのインターフェースがほかのインターフェースで使用された場合はメソッド宣言のシグニチャタイプとして使用されます。これには，マッピングされたオペレーションシグニチャであるメソッドが含まれています。

`holder` クラスが非抽象インターフェースとして生成されます。詳細については，42 ページの「Holder クラス」を参照してください。

標準規則に準じて，`helper` クラスも生成されます。詳細については，45 ページの「Helper クラス」を参照してください。

ローカルインターフェース

IDL ローカルインターフェースは，`org.omg.CORBA.LocalInterface` がマークしたローカルインターフェースを除き，非ローカルインターフェースと類似した方法でマッピングされます。ローカルインターフェースはマーシャリングされないことがあり，インプリメンテーションは特殊ベースの `org.omg.CORBA.LocalObject` を拡張して生成されたシグニチャインターフェースを実装する必要があります。Java マッピングの場合，`LocalObject` クラスを

ローカルインターフェースのインプリメンテーション基底クラスとして使用します。ローカルインターフェースインプリメンテーションのインスタンス作成は，新しい Java 演算子を使った通常の Java オブジェクト作成と同じです。

holder クラスが非ローカルインターフェースとして生成されます。詳細については，42 ページの「Holder クラス」を参照してください。

標準規則に準じて，helper クラスも生成されます。詳細については，45 ページの「Helper クラス」を参照してください。

VisiBroker ORB インプリメンテーションは，ローカルオブジェクトをマーシャリングして CORBA::MARSHAL 例外を生成しようとする試みを検知します。

パラメータの受け渡し

IDL in パラメータは，標準の Java の実際のパラメータにマッピングされます。IDL オペレーションの結果は，対応する Java メソッドの結果として返されます。

IDL の out および inout パラメータは，Java のパラメータ受け渡しメカニズムに直接マッピングできません。これをマッピングするには，IDL の基本型とユーザー定義型のすべてについて追加のホルダークラスを定義し，Java で上記のパラメータモードを実装するために使用します。クライアントは，IDL の out と inout の各パラメータを渡す（値渡し）ために，適切な Java ホルダークラスのインスタンスを提供します。ホルダーのインスタンスの内容（インスタンス自体ではなく）は，呼び出しによって変更されます。クライアントは，呼び出しが戻った後で，変更された内容を使用します。

このサンプルコードは，Java の実際のパラメータへマッピングする IN パラメータを示しています。

```
/* Example.idl より */
module Example {
   interface Modes {
      long operation(in long inArg, out long outArg, inout long inoutArg);
   };
};
// 生成された Java コード
package Example;
public interface Modes extends com.borland.vbroker.CORBA.Object,
      Example.ModesOperations,
      org.omg.CORBA.portable.IDLEntity {
}
public interface ModesOperations {
   public int operation (int inArg,
         org.omg.CORBA.IntHolder outArg,
         org.omg.CORBA.IntHolder inoutArg);
}
```

上のサンプルの場合，結果は通常の結果として返され，in の実パラメータだけが通常の値です。しかし，out と inout のパラメータには，適切なホルダーを構築する必要があります。ホルダーは次のように使用します。

```
// ユーザーの Java コード
// ターゲットオブジェクトを選択します。
Example.Modes target = ...;
// 実パラメータの値を得ます。
int inArg = 57;

// out パラメータの受け取りを準備します。
IntHolder outHolder =          new IntHolder();
// inout パラメータの入力側を設定します。
IntHolder inoutHolder = new IntHolder(131);
// 呼び出しを行います。
int result =target.operation(inArg, outHolder, inoutHolder);
// outHolder の値を使用します。
... outHolder.value ...
```

```
// inoutHolder の値を使用します。
... inoutHolder.value ...
```

呼び出しを行う前に，実パラメータとなるホルダーのインスタンスに inout パラメータの入力値を設定する必要があります。inout ホルダーに値を設定するには，値から新しいホルダーを構築するか，適切な型の既存のホルダーに値を代入します。呼び出しを行った後で，クライアントは，outHolder.value を使って out パラメータの値にアクセスし，inoutHolder.value を使って inout パラメータの出力値にアクセスします。IDL オペレーションの戻り値は，呼び出しの結果として使用できます。

継承を使ったサーバーインプリメンテーション

サーバーを実装するには，継承を使用する方法が最も簡単です。サーバーオブジェクトとオブジェクトリファレンスはたいへん似ており，同じように動作し，正確に同じコンテキストで使用できるからです。サーバーオブジェクトがそのクライアントと同じプロセスに存在する場合，メソッドの呼び出しは，どのような種類のトランスポート，インダイレクション，およびデリゲーションも伴わない通常の Java の関数呼び出しになります。

各 IDL インターフェースは，その IDL インターフェースの Java バージョンを実装する Java POA 抽象クラスにマッピングされます。

メモ POA クラスは，本来の IDL インターフェースを拡張しません。したがって，POA は CORBA オブジェクトではありません。POA は CORBA サーバントであり，実際の CORBA オブジェクトを作成するために使用されます。POA クラスの詳細については，Java API リファレンスの VisiBroker API で，org.omg.PortableServer パッケージを参照してください。POA の詳細については，95 ページの「POA の使い方」を参照してください。

次のサンプルコードに示すように，次に <interface>POA クラスを拡張することにより，ユーザー定義のサーバークラスが ORB にリンクされます。

メモ POA クラス自体は，抽象クラスなので，インスタンス化できません。このクラスをインスタンス化するには，それに宣言されている IDL インターフェースオペレーションを各自の実装で実装する必要があります。

このサンプルコードは，継承を使った Java のサーバーインプリメンテーションを表示します。

```
/* Bank.idl より */
module Bank {
   interface Account {
   };
};
// 生成された Java
package Bank;
public abstract class AccountPOA extends org.omg.PortableServer.Servant
implements
      org.omg.CORBA.portable.InvokeHandler,
      Bank.AccountOperations { . . . }
// インプリメンテーションを ORB にリンクします。
public class AccountImpl extends Bank.AccountPOA { . . . }
```

デリゲーションを使ったサーバーインプリメンテーション

継承を使ったサーバーのインプリメンテーションには，不利な点が 1 つあります。サーバークラスは POA スケルトンクラスを拡張するため，インプリメンテーションの継承をほかの目的で使用できません。Java は，単一の継承だけをサポートするからです。サーバークラスが別の目的で 1 つの継承リンクを使用する必要がある場合は，デリゲーションの手法を使用する必要があります。

デリゲーションを使ってサーバークラスを実装する場合は，次の追加コードが生成されます。

- 各インターフェースが Tie クラスにマッピングされます。このクラスは POA スケルトンを拡張し，デリゲーションコードを提供します。

- 各インターフェースが Operations インターフェースにもマッピングされます。このインターフェースは，Tie クラスがデリゲートするオブジェクトの型を定義するために使用されます。

デリゲートされるインプリメンテーションは，Operation インターフェースを実装し，Tie クラスのインスタンス内に格納される必要があります。Tie クラスのコンストラクタを使用して，Operation インターフェースのインスタンスを Tie オブジェクトに格納できます。次のサンプルコードにデリゲーションの方法を示します。

```
/* Bank.idl より */
module Bank {
   interface AccountManager {
      Account open(in string name);
   };
};
// 生成された Java
package Bank;
public interface AccountManagerOperations {
   public Example.Account open(java.lang.String name);
}
// 生成された Java
package Bank;
public class AccountManagerPOATie extends AccountManagerPOA {
   public AccountManagerPOATie (final Bank.AccountManagerOperations _delegate)
      { . . . }
   public AccountManagerPOATie (final Bank.AccountManagerOperations _delegate,
         final org.omg.PortableServer.POA _poa) { . . . }
   public Bank.AccountManagerOperations _delegate () { . . . }
   public void _delegate (final Bank.AccountManagerOperations delegate) { . . .
}
   public org.omg.PortableServer.POA _default_POA () { . . . }
   public float open () { . . . }
}
// インプリメンテーションを ORB にリンクします。
   classAccountImpl implements AccountManager Operations
public class Server {
   public static main(String args) {
      // ...
      AccountManagerPOAtie managerServant = new AccountManagerPOATie(new
AccountManagerImpl());
      // ...
}
```

インターフェースのスコープ

OMG の IDL から Java へのマッピング仕様では，複数のデリゲーションを 1 つのインターフェーススコープ内でネストすることができません。また，パッケージとインターフェースが同じ名前を持つこともできません。したがって，インターフェースのスコープは，「Package」サフィックスが付いた同じ名前のパッケージにマッピングされます。

例外のマッピング

IDL の例外は，構造体とよく似た方法でマッピングされます。IDL の例外は，例外のフィールドに対応するインスタンス変数とコンストラクタを提供する Java クラスにマッピングされます。

CORBA のシステム例外は，チェックされない例外です。java.lang.RuntimeException から（間接的に）継承されます。

ユーザー定義の例外は，チェックされる例外です。java.lang.Exception から（間接的に）継承されます。

ユーザー定義例外

ユーザー定義の例外は，`org.omg.CORBA.UserException` を拡張する Java final クラスにマッピングされるか，そうでない場合は，Helper クラスと Holder クラスの生成も含め，IDL `struct` 型とまったく同じようにマッピングされます。

例外が，ネストされた IDL スコープ内（つまり，インターフェース内）で定義されている場合，その Java クラス名は，特別なスコープ内で定義されます。そうでない場合，その Java クラス名は，その例外が属する IDL モジュールに対応する Java パッケージのスコープ内で定義されます。

このサンプルコードは，ユーザー定義例外に対する IDL から Java へのマッピングを示しています。

```
// IDL
module Example {
   exception AnException {
      string reason;
   };
};
// 生成された Java
package Example;
public final class AnException extends org.omg.CORBA.UserException {
   public java.lang.String extra;
   public AnException () { . . . }
   public AnException (java.lang.String extra) { . . . }
   public AnException (java.lang.String _reason, java.lang.String extra) { . .
. }
   public java.lang.String to String () { . . . }
public boolean equals (java.lang.Object o) { . . . }
}
public final class AnExceptionHolder implements
      org.omg.CORBA.portable.Streamable {
   public Example.AnException value;
   public AnExceptionHolder () { }
   public AnExceptionHolder (final Example.AnException _vis_value) { . . . }
   public void _read (final org.omg.CORBA.portable.InputStream input) { . . . }
   public void _write (final org.omg.CORBA.portable.OutputStream output) { . .
. }
   public org.omg.CORBA.TypeCode _type () { . . . }
}
```

システム例外

標準の IDL システム例外は，`org.omg.CORBA.SystemException` を拡張する Java final クラスにマッピングされ，IDL のメジャーおよびマイナー例外コードのほか，その例外の原因を記述した文字列にもアクセスできます。`org.omg.CORBA.SystemException` には，public コンストラクタがありません。これを拡張したクラスだけをインスタンス化できます。

標準 IDL 例外の Java クラス名は IDL 名と同じで，`org.omg.CORBA` パッケージ内で宣言されます。デフォルトコンストラクタは，マイナーコードには 0，完了コードには `COMPLETED_NO`，原因文字列には空文字列（""）を提供します。原因を受け取り，ほかのフィールドにはデフォルト値を使用するコンストラクタ，および 3 つのパラメータすべての指定を要求するコンストラクタもあります。

Any 型のマッピング

IDL の Any 型は，Java の org.omg.CORBA.Any クラスにマッピングされます。このクラスは，定義済みの型のインスタンスを挿入および抽出するために必要なメソッドをすべて持っています。抽出処理で型の不一致があると，CORBA::BAD_OPERATION 例外が生成されます。

また，可搬性のあるスタブとスケルトンに効率のよいインターフェースを提供するために，ホルダークラスを受け取る挿入と抽出のメソッドが定義されています。各 IDL プリミティブ型に挿入と抽出のメソッドが定義されているほか，非プリミティブの IDL 型を処理するために，汎用のストリーム可能な型にも挿入と抽出のメソッドが定義されています。

挿入処理は，指定された値を設定し，必要な場合は Any の型を再設定します。

type() アクセッサを介してタイプコードを設定すると，その値が削除されます。値が設定される前に抽出しようとすると，CORBA::BAD_OPERATION 例外が生成されます。このオペレーションは，主に IDL out パラメータの型を正しく設定するために用意されています。

ネストされた型のマッピング

IDL では，型宣言をインターフェース内でネストできます。Java では，クラスをインターフェース内でネストすることはできません。このため，Java クラスにマッピングされ，1 つのインターフェースのスコープ内で宣言されている複数の IDL 型は，Java にマッピングされたとき，専用のスコープパッケージ内に置かれます。

このような型宣言を持つ IDL インターフェースからはスコープパッケージが生成され，マッピングされる Java クラス宣言がその中に入ります。スコープパッケージの名前は，IDL 型の名前の末尾に Package を付けて作成されます。

このサンプルコードは，特定のネストされた型に対する IDL から Java へのマッピングを示しています。

```
// IDL
module Example {
   interface Foo {
      exception e1 {};
   };
}
// 生成された Java
package Example.FooPackage;
final public class e1 extends org.omg.CORBA.UserException {...}
```

typedef のマッピング

Java には typedef 構造がありません。

単純な IDL 型

単純な Java の型にマッピングされる IDL の型は，Java ではサブクラス化されません。つまり，単純型の型宣言を持つ typedef は，常に元の型（マッピング元の型）にマッピングされます。単純型の場合は，すべての typedef に Helper クラスが生成されます。

複雑な IDL 型

配列とシーケンス以外の typedef は，typedef でない IDL の単純型またはユーザー定義型が見つかるまで，元の型に戻すことはできません。

シーケンスと配列の typedef には，Holder クラスが生成されます。

このサンプルコードは，複雑な IDL typedef のマッピングを示しています。

```
// IDL
struct EmpName {
   string firstName;
   string lastName;
};
typedef EmpName EmpRec;
// 生成された Java
         // EmpName に関する通常の構造体のマッピング
         // EmpRec に関する通常のヘルパークラスのマッピング
final public class EmpName {
   ...
}
public class EmpRecHelper {
   ...
}
```

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。

IIOP を介した Java RMI のプロパティ

表 6.1 IIOP を介した Java RMI のプロパティ

プロパティ	デフォルト値	説明
vbroker.rmi.supportRTSC	false	このプロパティは，クライアントとサーバーが異なる（更新された）バージョンのクラスを使用する場合に，それらの間の SendingContextRuntime サービスコンテキストの交換を有効または無効にします。クライアントとサーバーの JDK が異なるバージョンである場合，アプリケーションでは，このプロパティを true に設定する必要があります。この値は，VBJ が外部の ORB とやり取りする場合にも使用する必要があります。これにより，コードベースデータが交換され，更新されたクラスのマーシャリング／アンマーシャリングが例外の発生なく正常に行われるようになります。
javax.rmi.CORBA.StubClass	com.inprise.vbroker.rmi.CORBA.StubImpl	Stub 基底クラスのインプリメンテーションの名前を指定します。すべての RMI-IIOP スタブがこのクラスを継承します。
javax.rmi.CORBA.UtilClass	com.inprise.vbroker.rmi.CORBA.UtilImpl	Utility クラスのインプリメンテーションの名前を指定します。このクラスは，スタブと tie が共通のオペレーションを実行するためのメソッドを提供します。
javax.rmi.CORBA.PortableRemoteObjectClass	com.inprise.vbroker.rmi.CORBA.PortableRemoteObjectImpl	RMI-IIOP サーバーインプリメンテーションオブジェクトが javax.rmi.PortableRemoteObject を継承することを指定します。または，単に RMI-IIOP リモートインターフェースを実装し，次に exportObject メソッドを使用して自分自身をサーバーオブジェクトとして登録します。
java.rmi.server.codebase	<not set>	サーバーが未知のクラスを見つけることができる場所を指定します。使用できる値は，セミコロン（;）で区切られた URL です。
java.rmi.server.useCodebaseOnly	false	サーバーが未知のクラスを検索できるかどうかを指定します。true に設定すると，クライアントがサーバーにリモートクラスの場所を送信した場合でも，サーバーはリモートクラスを検索できません。

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 6.2　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.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.enableCache	true	ORBagentCache	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.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。

プロパティ	デフォルト値	以前のプロパティ	説明
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。
vborker.agent.clientPort	0	N/A	ORB が OSAgent と通信するためのポート範囲の下限。有効な値は 1025 ～ 65535 です。デフォルト値 0 は，ランダムなポートが選択されることを意味します。
vbroker.agent.clientPortRange	0	N/A	ORB が OSAgent と通信するための [clientPort, clientPort+clientPortRange] の範囲内のポート。このプロパティは，clientPort が 0 より大きい場合にのみ有効です。有効な値は 1025 ～ 65535 です。

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 6.3 VisiBroker ORB のプロパティ

プロパティ	デフォルト値	説明
vbroker.orb.propOrdering	CMD_PROPS:SYS_PROPS:FILE_PROPS:ORB_PROPS:DEF_PROPS	このプロパティを使用すると，ORB のプロパティマネージャによって設定されているプロパティのデフォルトの優先順位をオーバーライドできます。デフォルトの優先順位（降順）は次のとおりです。 **1** **CMD_PROPS**：コマンドライン引数（orb.init() 呼び出しの最初の引数を介して指定されます)。 **2** **SYS_PROPS**：システムまたは JVM のプロパティ。-VBJprop，-J などで指定されるプロパティを含みます。 **3** **FILE_PROPS**：ORBpropStorage プロパティによって指定されるファイルに含まれるプロパティ。 **4** **ORB_PROPS**：orb.init() 呼び出しの第 2 引数を介して設定されるプロパティ。 **5** **DEF_PROPS**：ORB のデフォルトのプロパティ。
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.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.enableKeyId	false	true に設定した場合は，クライアント要求でキー ID を使用できます。
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.keyIdCacheMax	16384	サーバー内のオブジェクトキー ID キャッシュの最大サイズを指定します。

表 6.3　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.keyIdCacheMin	64	サーバー内のオブジェクトキー ID キャッシュの最小サイズを指定します。
vbroker.orb.initRef	null	初期リファレンスを指定します。
vbroker.orb.defaultInitRef	null	デフォルトの初期リファレンスを指定します。
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。
vbroker.orb.alwaysSecure	false	true に設定した場合は，クライアントが必ずサーバーにセキュリティのかかった接続を行う必要があることを示します。
vbroker.orb.alwaysTunnel	false	true に設定した場合は，クライアントが必ずサーバーに HTTP トンネル（IIOP ラッパー）接続を行うことを示します。
vbroker.orb.autoLocateStubs	false	オブジェクトリファレンスを読み取るときに，スタブの検索機能をオンにします。これは，共通オブジェクトや渡された正式なクラス引数のスタブではなく，オブジェクトのリポジトリ ID を基に read_Object を使用して行われます。
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。

表 6.3　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.bids.bar	該当なし	このプロパティは，特定のビッドを指定できないようにします。たとえば，inprocess に設定すると，インプロセスビッドが無効になります。これは，最適化された連結呼び出しが必要でない場合に便利です。現在のところ，インプロセスビッドだけを禁止できます。
vbroker.orb.defAddrMode	0（キー）	クライアント VisiBroker ORB が使用するデフォルトのアドレッシングモード。0 に設定されている場合，アドレスモードは Key になります。1 に設定されている場合はアドレスモードが Profile になります。2 に設定されている場合のアドレスモードは IOR になります。
vbroker.orb.bufferCacheTimeout	6000	メッセージチャンクをキャッシュしてから破棄されるまでの時間を指定します。
vbroker.orb.bufferDebug	false	true に設定した場合は，内部バッファマネージャがデバッグ情報を表示します。
vbroker.orb.corbaloc.resolveHosts	false	このプロパティが true に設定されている場合，ORB は，corbaloc URL で指定されたホスト名の解決を試みます。false の場合，アドレスの解決は行われません。
vbroker.orb.debug	false	true に設定した場合は，ORB がデバッグ情報を表示できます。
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.enableBiDir	なし	双方向接続は，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。このプロパティの値は，server，client，both，または none です。詳細は，『GateKeeper ガイド』の「GateKeeper の双方向サポートによるコールバック」を参照してください。
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.fragmentSize	0	GIOP メッセージのフラグメントサイズを指定します。GIOP メッセージのチャンクサイズの倍数を指定する必要があります。このプロパティに 0 を割り当てると，フラグメンテーションがオフになります。
vbroker.orb.streamChunkSize	4096	GIOP メッセージのチャンクサイズを指定します。2 の累乗を指定する必要があります。
vbroker.orb.gcTimeout	30	使用されていない重要なリソースが消去されるまでの時間を秒単位で指定します。
vbroker.orb.logger.appName	VBJ-Application	ログに表示されるアプリケーション名を指定します。
vbroker.orb.logger.catalog	com.inprise.vbroker.Logging.ORBMsgs	ログが有効の場合に，ORB が使用するメッセージカタログを指定します。

表 6.3　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.logger.output	stdout	ログの出力先を指定します。標準出力またはファイル名を指定できます。
vbroker.orb.logLevel	emerg	ログに記録されるメッセージのログレベルを指定します。デフォルト値の emerg は，システムを使用できないか，異常な状態の場合に，システムがメッセージを記録することを指定します。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，エラーではないとしても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)：開発者が理解できるデバッグ状態です。
vbroker.orb.sendLocate	false	このプロパティは，true，false，onbind，または always のいずれかの値をとります。true に設定した場合，IIOP 1.2 ターゲットへの呼び出しを行う前に，システムが検索要求を送信することを強制します。onbind の場合は，ピアが GIOP を認識しているかどうかを判定するために，接続が開かれると検索要求メッセージが送信されます。always という値は，呼び出しの前と接続を開いたときの両方で検索要求を送信するというタスクを実行するように ORB に指示します。
vbroker.orb.shutdownTimeout	0	アプリケーションが ORB.shudown オペレーションのタイムアウトを設定できるようにします。このプロパティは，ORB.shutdown が長時間終了しない場合に便利です。このプロセスは，シャットダウンが終了せず，タイムアウトになると終了されます。デフォルト値 0 は，プロセスが終了されないことを意味します。
vbroker.orb.systemLibs.applet	com.inprise.vbroker.IIOP.Init, com.inprise.vbroker.LIOP.Init, com.inprise.vbroker.qos.Init, com.inprise.vbroker. URLNaming.Init, com.inprise.vbroker.HIOP.Init, com.inprise.vbroker. firewall.Init, com.inprise.vbroker. dynamic.Init, com.inprise.vbroker. naming.Init, com.inprise.vbroker.IOP.Init, com.inprise.vbroker. CONV_FRAME.Init, com.inprise.vbroker.rmi. CORBA.Init, com.inprise.vbroker. PortableInterceptor.Init, com.borland.vbroker.notify.Init, com.borland.vbroker.CosTime.Init	アプレットにロードされるシステムライブラリのリストを指定します。

表 6.3　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.systemLibs.application	com.inprise.vbroker.IIOP.Init, com.inprise.vbroker.LIOP.Init, com.inprise.vbroker.qos.Init, com.inprise.vbroker.ds.Init, com.inprise.vbroker. URLNaming.Init, com.inprise.vbroker. dynamic.Init, com.inprise.vbroker.ir.Init, com.inprise.vbroker. naming.Init, com.inprise.vbroker. ServerManager.Init, com.inprise.vbroker.IOP.Init, com.inprise.vbroker. CONV_FRAME.Init, com.inprise.vbroker.rmi. CORBA.Init, com.inprise.vbroker. PortableInterceptor.Init, com.borland.vbroker. notify.Init, com.borlvbroker.CosTime.Init	アプリケーションにロードされるシステムライブラリのリストを指定します。
vbroker.orb.tcIndirection	true	タイプコードを書き込むときのインダイレクションをオフにします。このプロパティは，他のベンダーの ORB と互換性を保つ場合に必要になる可能性があります。false に設定した場合は，再帰的なタイプコードをマーシャリングできなくなります。
vbroker.orb.warn	0	0，1，または 2 を指定して，出力する警告メッセージのレベルを示します。
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。

POA のプロパティ

表 6.4　POA のプロパティ

プロパティ	デフォルト値	説明
vbroker.poa.logLevel	emerg	ログに記録されるメッセージのログレベルを指定します。デフォルト値の emerg は，システムを使用できないか，異常な状態の場合に，システムがメッセージを記録することを指定します。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，エラーではないとしても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)：開発者が理解できるデバッグ状態です。

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 6.5　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.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。

プロパティ	説明
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.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。

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 6.6　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，208 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.factoryIorFile	N/A	このプロパティにファイル名を示す値が指定されている場合，ネーミングサービスは，そのファイルにコンテキストファクトリの IOR を保存します。その IOR ファイルは，ネーミングサービスをリモートにシャットダウンするために，nsutil ユーティリティによって使用されます。
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。 • 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。

表 6.6　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=9909 17734, key_string=%00VB%01%00%00%00%02 /%00%20%20%00%00%00% 04%00%00%00%00%00%00%01%f9;%104 f],codebase=null]

詳細は，203 ページの「クラスタ」を参照してください。

表 6.7　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。

表 6.7　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.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，208 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 6.8　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	208 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.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。

表 6.8　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.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。

プロパティ	デフォルト値	説明
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，202 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。
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。

プロパティ	デフォルト値	説明
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。

設定値	説明
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，213 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。
vbroker.oad.vbj	vbj	VisiBroker for Java 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。

クライアント側 IIOP 接続のプロパティ

次の表は，VisiBroker for Java 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.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。
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.ccm.waitForCompletion	false	このプロパティを true に設定すると，アプリケーションがすべての応答を受信するまで待機してから，ORB が接続を閉じるように指定できます。デフォルト値 false は，ORB が応答もまったく待機しないことを示します。
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.clientPort	0（任意のポート）	ORB が接続を開いたときに使用されるクライアントポートを指定します。指定できる値は 0 ～ 65535 です。このプロパティを使用する場合は，vbroker.ce.iiop.clientPortRange プロパティを使用して範囲を指定する必要があります。
vbroker.ce.iiop.clientPortRange	0	ORB が接続を開いたときに使用されるクライアントポートの範囲を指定します。vbroker.ce.iiop.clientPort プロパティで指定されたポートから開始します。指定できる値は 0 ～ 65535 です。
vbroker.ce.iiop.host	なし	このプロパティは，マルチホームマシンから接続を開く場合に使用するクライアントアドレスを宣言します。指定しない場合は，デフォルトのアドレスが使用されます。

URL ネーミングのプロパティ

次の表は，URL ネーミング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.URLNaming.allowUserInteraction	true	true に設定した場合，URL ネーミングサービスは，ユーザーとの対話用にグラフィカルユーザーインターフェース（GUI）を起動します。
vbroker.URLNaming.debug	false	true に設定した場合は，URL ネーミングサービスがデバッグ情報を表示することを指定します。

QoS 関連のプロパティ

プロパティ	デフォルト値	説明
vbroker.orb.qos.relativeRTT	0	このプロパティを使用すると，RelativeRoundtripTimeoutPolicy をミリ秒単位で設定できます。これは ORB レベルで有効で，他のレベルでは，プログラムによってオーバーライドできます。デフォルト値 0 は，タイムアウトがないことを意味します。
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。
vbroker.orb.qos.connectionTimeout	0（制限なし）	このプロパティにより，明示的にコードを記述する必要がなくなり，RelativeConnectionTimeoutPolicy Qos ポリシーを ORB レベルで設定しやすくなります。接続タイムアウト値は，ミリ秒単位で指定する必要があります。

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.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.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。

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，80 ページの「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」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。
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.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.portRange	0	このプロパティは，listener.port が 0 より大きい場合にのみ有効です。ポートが使用中なのでバインドできない場合，リスナーは，［port, port+portRange］の範囲内のポートにバインドするように試みます。その範囲内のポートがどれも利用できない場合は，COMM_FAILURE 例外が生成されます。
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.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.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，81 ページの「サーバー側スレッドプール IIOP_TP/IIOP_TP 接続のプロパティ」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。

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 35 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（続き）

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 35 章「CORBA 例外」を参照してください。

SystemException クラス

```
public abstract class org.omg.CORBA.SystemException extends
java.lang.RuntimeException {
   protected SystemException(java.lang.String reason,
         int minor, CompletionStatus completed) { . . . }
   public String toString() { . . . }
   public CompletionStatus completed;
   public int minor;
}
```

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;
};
```

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。

```
public class Client {
    public static void main(String[] args) {
        try {
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
            byte[] managerId = "BankManager".getBytes();
            Bank.AccountManager manager =
                    Bank.AccountManagerHelper.bind(orb, "/bank_agent_poa",
managerId);
            String name = args.length > 0 ? args[0] : "Jack B. Quick";
            Bank.Account account = manager.open(name);
            float balance = account.balance();
            System.out.println("The balance in " + name + "'s account is $" +
balance);
        } catch (Exception e) {
            System.err.println(e);
        }
    }
}
```

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>vbj Client
org.omg.CORBA.OBJECT_NOT_EXIST:
Could not locate the following POA:
poa name : /bank_agent_poa
minor code: 0 completed: 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。

```
public class Client {
    public static void main(String[] args) {
        try {
            // ORB を初期化します。
            org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
            // 口座にバインドします。
            Account account = AccountHelper.bind(orb, "/bank_poa",
                    "BankAccount".getBytes());
            // 口座の残高を取得します。
            float balance = account.balance();
            // 残高を出力します。
            System.out.println("The account balance is $" + balance);
            catch(Exception e) {
            if (e instanceof org.omg.CORBA.SystemException) {
                System.err.println("System Exception occurred:");
            } else {
                System.err.println("Not a system exception");
            }
            System.err.println(e);
        }
```

```
      }
   }
```

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:
in thread "main" org.omg.CORBA.OBJECT_NOT_EXIST minor code: 0 completed: No
```

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。

```
public class Client {
   public static void main(String[] args) {
      try {
         org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
         byte[] managerId = "BankManager".getBytes();
         Bank.AccountManager manager =
               Bank.AccountManagerHelper.bind(orb, "/bank_agent_poa",
managerId);
         String name = args.length > 0 ? args[0] : "Jack B. Quick";
         Bank.Account account = manager.open(name);
         float balance = account.balance();
         System.out.println("The balance in " + name + "'s account is $" +
balance);
      } catch(org.omg.CORBA.SystemException e) {
         System.err.println("System Exception occurred:");
         System.err.println(e);
      }
   }
}
```

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。idl2java 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。

```
public abstract class UserException extends java.lang.Exception {
   protected UserException();
   protected UserException(String reason);
}
```

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);
   };
};
```

idl2java コンパイラは，AccountFrozen 例外クラスに次のコードを生成します。

```
package Bank;
public interface Account extends com.inprise.vbroker.CORBA.Object,
      Bank.AccountOperations, org.omg.CORBA.portable.IDLEntity {
}
package Bank;
public interface AccountOperations {
   public float balance () throws Bank.AccountPackage.AccountFrozen;
}
package Bank.AccountPackage;
public final class AccountFrozen extends org.omg.CORBA.UserException {
   public AccountFrozen () { . . . }
   public AccountFrozen (java.lang.String _reason) { . . . }
   public synchronized java.lang.String toString() { . . . }
}
```

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。

```
public class AccountImpl extends Bank.AccountPOA {
   public AccountImpl(float balance) {
      _balance = balance;
   }
   public float balance() throws AccountFrozen {
      if (_balance < 50) {
         throws AccountFrozen();
      } else {
         return _balance;
   }
   private float _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。

```
public class Client {
   public static void main(String[] args) {
      try {
         // ORB を初期化します。
         org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
         // 口座にバインドします。
         Account account = AccountHelper.bind(orb, "/bank_poa",
                        "BankAccount".getBytes());
         // 口座の残高を取得します。
         float balance = account.balance();
         // 残高を出力します。
         System.out.println("The account balance is $" + balance);
      }
      // AccountFrozen 例外をチェックします。
      catch(Account.AccountFrozen e) {
         System.err.println("AccountFrozen returned:");
         System.err.println(e);
      }
      // システムエラーをチェックします。
      catch(org.omg.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
org.ogm.CORBA.ORB orb=org.omg.CORBA.ORB.init(args,null);
```

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。

```
//2. ルート POA へのリファレンスを取得します。
org.omg.CORBA.Object obj = orb.resolve_initial_reference("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。
POA rootPoa = org.omg.PortableServer.POAHelper.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。

```
// 永続的 POA のポリシーを作成します。
org.omg.CORBA.Policy[] policies = {
   rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)
};
// 適切なポリシーで myPOA を作成します。
POA myPOA = rootPOA.create_POA( "bank_agent_poa", rootPOA.the_POAManager(),
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。

AccountManagerPOA.java が作成され，次のように，サーバー側のオブジェクトインプリメンテーション AccountManager のスケルトンコード(インプリメンテーションベースコード)として機能します。

```
import org.omg.PortableServer.*;
import java.util.*;
public class AccountManagerImpl extends Bank.AccountManagerPOA {
   public synchronized Bank.Account open(String name) {
      // account ディレクトリ内で口座を検索します。
      Bank.Account account = (Bank.Account) _accounts.get(name);
      // ディクショナリに口座が 1 つもない場合は，作成します。
      if(account == null) {
         // 0 ～ 1000 ドルの範囲で口座に残高を設定します。
         float balance = Math.abs(_random.nextInt()) % 100000 / 100f;
         // 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。
            account = Bank.AccountHelper.narrow(_default_POA().
            servant_to_reference(accountServant));
         } catch (Exception e) {
            e.printStackTrace();
         }
         // 新しい口座を出力します。
         System.out.println("Created " + name + "'s account: " + account);
         // 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 account;
   }
   private Dictionary _accounts = new Hashtable();
   private Random _random = new Random();
}
```

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 = new AccountManagerImpl();
// サーバントの ID を決定します。
byte[] managerId = "BankManager".getBytes();
// 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します。
PortableServer::POAManager_var mgr=rootPoa ->the_POAManager();
mgr->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.java
import org.omg.PortableServer.*;
public class Server {
public static void main(String[] args) {
   try {
      // ORB を初期化します。
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
      // ルート POA へのリファレンスを取得します。
      POA rootPOA =
POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
      // 永続的 POA のポリシーを作成します。
      org.omg.CORBA.Policy[] policies = {
            rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)
      };
      // 適切なポリシーで myPOA を作成します。
      POA myPOA = rootPOA.create_POA( "bank_agent_poa",
rootPOA.the_POAManager(),
            policies );
      // サーバントを作成します。
      AccountManagerImpl managerServant = new AccountManagerImpl();
      // サーバントの ID を決定します。
      byte[] managerId = "BankManager".getBytes();
      // 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
      rootPOA.the_POAManager().activate();
      System.out.println(myPOA.servant_to_reference(managerServant) + " is
ready.");
      // 着信要求を待機します。
      orb.run();
   } catch (Exception e) {
      e.printStackTrace();
   }
}
```

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。

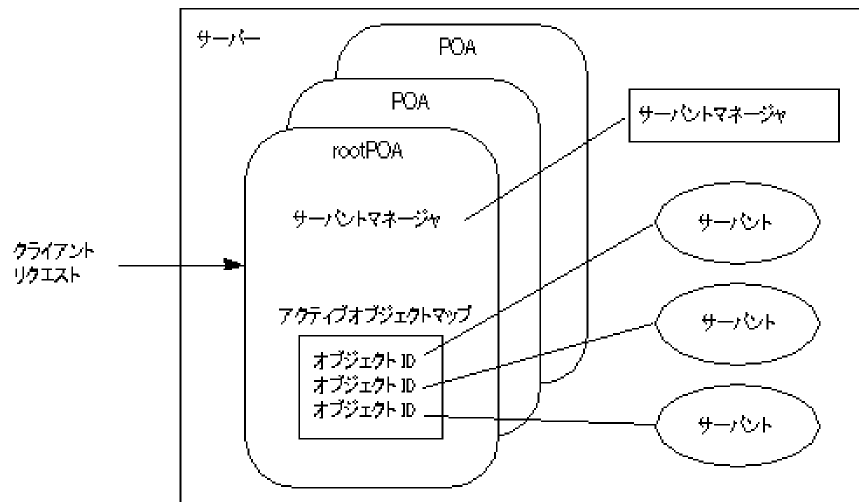
POA は，オブジェクトのインプリメンテーションと VisiBroker ORB を仲介します。POA は，仲介役として，要求をサーバントに送ります。その結果，サーバントが実行され，必要に応じて子 POA が作成されます。

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。
霊化	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，101 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，104 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。

- `org.omg.PortableServer.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。

```
org.omg.PortableServer.POA myPOA1 = rootPOA.create_POA("A/B",
      poaManager,
      policies);
org.omg.PortableServer.POA myPOA2 = myPOA1.create_POA("¥t",
      poaManager,
      policies);
```

クライアントは，次のようにバインドします。

```
org.omg.CORBA.Object manager = ((com.inprise.vbroker.orb.ORB) orb).bind("/A¥¥/
B/¥t",
      managerId,
      null,
      null);
```

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。
org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
// ルート POA へのリファレンスを取得します。
org.omg.PortableServer.POA rootPOA =
      POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
```

メモ resolve_initial_references メソッドは，org.omg.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，org.omg.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，97 ページの「POA ポリシー」を参照してください。

```
org.omg.CORBA.Policy[] policies = {
   rootPOA.create_lifespan_policy(LifespanPolicyValue.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，109 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。

```
// 永続的 POA のポリシーを作成します。
org.omg.CORBA.Policy[] policies = {
      rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)};
// 適切なポリシーで myPOA を作成します。
org.omg.PortableServer.POA myPOA =
      rootPOA.create_POA( "bank_agent_poa", rootPOA.the_POAManager(), 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。

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。
オンデマンドのアクティブ化	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。
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。
デフォルトサーバント	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。

```
// 口座マネージャサーバントを作成します。
Servant managerServant = new AccountManagerImpl(rootPoa);
// 新しく作成したサーバントをアクティブ化します。
testPoa.activate_object_with_id("BankManager".getBytes(), managerServant);
// POA をアクティブ化します。
testPoa.the_POAManager().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 メソッド
- POA.servant_to_id メソッド
- _this() サーバントメソッド

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。

```
import org.omg.PortableServer.*;
public class Server {
   public static void main(String[] args) {
      try {
         // ORB を初期化します。
         org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
         // ルート POA へのリファレンスを取得します。
         POA rootPOA =
POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
         // 永続的 POA のポリシーを作成します。
         org.omg.CORBA.Policy[] policies = {
            rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT),
            rootPOA.create_request_processing_policy(
                  RequestProcessingPolicyValue.USE_DEFAULT_SERVANT
            )

rootPOA.create_id_uniqueness_policy(IdUniquenessPolicyValue.MULTIPLE_ID)
         };
         // 適切なポリシーで myPOA を作成します。
         POA myPOA = rootPOA.create_POA( "bank_default_servant_poa",
               rootPOA.the_POAManager(),
               policies );
         // サーバントを作成します。
         AccountManagerImpl managerServant = new AccountManagerImpl();
```

```
         // 独自の POA にデフォルトサーバントを設定します。
         myPOA.set_servant(managerServant);
         org.omg.CORBA.Object ref;
         // POA マネージャをアクティブ化
         rootPOA.the_POAManager().activate();
         // 参照を生成し，それを書き出します。各当座預金口座と普通預金口座に参
             照が 1 つずつ
         // あります。ここではサーバントを作成しておらず，
         // 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
         try {
            ref =
myPOA.create_reference_with_id("CheckingAccountManager".getBytes(),
                  "IDL:Bank/AccountManager:1.0");
            // 当座のオブジェクト ID を書き出します。
            java.io.PrintWriter pw = new java.io.PrintWriter(
                  new java.io.FileWriter("cref.dat") );
            pw.println(orb.object_to_string(ref));
            pw.close();
            ref =
myPOA.create_reference_with_id("SavingsAccountManager".getBytes(),
                  "IDL:Bank/AccountManager:1.0");
            // 普通のオブジェクト ID を書き出します。
            pw = new java.io.PrintWriter( new java.io.FileWriter("sref.dat") );
            pw.println(orb.object_to_string(ref));
            pw.close();
         } catch ( java.io.IOException e ) {
            System.out.println("Error writing the IOR to file ");
            return;
         }
         System.out.println("Bank Manager is ready.");
         // 着信要求を待機します。
         orb.run();
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

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。

```
import org.omg.PortableServer.*;
public class AccountManagerActivator extends _ServantActivatorLocalBase {
   public Servant incarnate (byte[] oid, POA adapter) throws ForwardRequest {
      Servant servant;
      String accountType = new String(oid);
      System.out.println("¥nAccountManagerActivator.incarnate called with ID = "
            + accountType + "¥n");
      // AccountType に基づいて，当座または普通のサーバントを作成します。
      if ( accountType.equalsIgnoreCase("SavingsAccountManager"))
         servant = (Servant )new SavingsAccountManagerImpl();
      else
         servant =(Servant)new CheckingAccountManagerImpl();
         new DeactivateThread(oid, adapter).start();
         return servant;
```

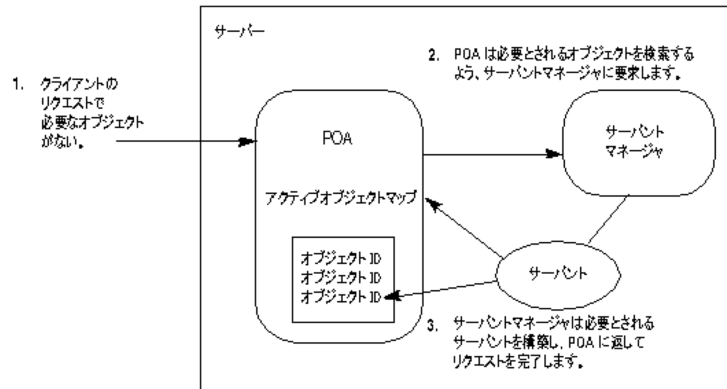
```
   }
   public void etherealize (byte[] oid,
         POA adapter,
         Servant serv,
         boolean cleanup_in_progress,
         boolean remaining_activations) {
      System.out.println("¥nAccountManagerActivator.etherealize called with ID ="
            + new String(oid) + "¥n");
      }
}
class DeactivateThread extends Thread {
   byte[] _oid;
   POA _adapter;
   public DeactivateThread(byte[] oid, POA adapter) {
      _oid = oid;
      _adapter = adapter;
   }
   public void run() {
      try {
         Thread.currentThread().sleep(15000);
         System.out.println("¥nDeactivating the object with ID = " +
         new String(_oid) + "¥n");
         _adapter.deactivate_object(_oid);
      } catch (Exception e) {
         e.printStackTrace();
   }
}
}
```

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，97 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，103 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。

```
import org.omg.PortableServer.*;
public class Server {
```

```
   public static void main(String[] args) {
      try {
         // ORB を初期化します。
         org.omg.CORBA.ORB orb = org.omg.CORBA.FORB.init(args,null);
         // ルート POA へのリファレンスを取得します。
         POA rootPOA =
                  POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
         // 独自の POA のポリシーを作成します。PERSISTENT 存続期間ポリシーと
         // USE_SERVANT_MANAGER 要求処理ポリシーが必要です
         org.omg.CORBA.Policy[] policies = {
            rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT),

   rootPOA.create_request_processing_policy(RequestProcessingPolicyValue.
               USE_SERVANT_MANAGER)
         };
         // 適切なポリシーで myPOA を作成します。
         POA myPOA = rootPOA.create_POA( "bank_servant_activator_poa",
               rootPOA.the_POAManager(),
               policies );
         // サーバントアクティベータサーバントを作成し，そのリファレンスを取得します。
         ServantActivator sa = new AccountManagerActivator();
         // 独自の POA にサーバントアクティベータを設定します。
         myPOA.set_servant_manager(sa);
         org.omg.CORBA.Object ref;
         // POA マネージャをアクティブ化
         rootPOA.the_POAManager().activate();
         // 参照を生成し，それを書き出します。各当座預金口座と
普通預金口座に参
         // 照が 1 つずつあります。ここではサーバントを作成しておらず,
         // 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
         try {
            ref =
myPOA.create_reference_with_id("CheckingAccountManager".getBytes(),
                  "IDL:Bank/AccountManager:1.0");
            // 当座のオブジェクト ID を書き出します。
            java.io.PrintWriter pw =
                  new java.io.PrintWriter( new java.io.FileWriter("cref.dat")
);
            pw.println(orb.object_to_string(ref));
            pw.close();
            ref =

myPOA.create_reference_with_id("SavingsAccountManager".getBytes(),
                  "IDL:Bank/AccountManager:1.0");
            // 普通のオブジェクト ID を書き出します。
            pw = new java.io.PrintWriter( new java.io.FileWriter("sref.dat") );
            pw.println(orb.object_to_string(ref));
            pw.close();
         } catch ( java.io.IOException e ) {
            System.out.println("Error writing the IOR to file ");
            return;
         }
         System.out.println("Bank Manager is ready.");

// 着信要求を待機します。
         orb.run();
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

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。

```
import org.omg.PortableServer.*;
public class AccountManagerActivator extends _ ServantActivatorLocalBase {
   public Servant incarnate (byte[] oid, POA adapter) throws ForwardRequest {
      Servant servant;
      String accountType = new String(oid);
      System.out.println("¥nAccountManagerActivator.incarnate called with ID =
                       " + accountType + "¥n");
      // AccountType に基づいて，当座または普通のサーバントを作成します。
      if ( accountType.equalsIgnoreCase("SavingsAccountManager"))
         servant = (Servant )new SavingsAccountManagerImpl();
      else
         servant =(Servant)new CheckingAccountManagerImpl();
      new DeactivateThread(oid, adapter).start();
      return servant;
   }

   public void etherealize (byte[] oid,
         POA adapter,
         Servant serv,
         boolean cleanup_in_progress,
         boolean remaining_activations) {
      System.out.println("¥nAccountManagerActivator.etherealize called with ID
=
                     " + new String(oid) + "¥n");
   }
}
class DeactivateThread extends Thread {
   byte[] _oid;
   POA _adapter;
   public DeactivateThread(byte[] oid, POA adapter) {
      _oid = oid;
      _adapter = adapter;
   }

 public void run() {

    try {
    Thread.currentThread().sleep(15000);
    System.out.println("¥nDeactivating the object with ID =
    " + new String(_oid) + "¥n");
    _adapter.deactivate_object(_oid);
    } catch (Exception e) {
       e.printStackTrace();
    }
}
 }
```

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。

```
import org.omg.PortableServer.*;
public class Server {
   public static void main(String[] args) {
      try {
         // ORB を初期化します。
         org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
         // ルート POA へのリファレンスを取得します。
         POA rootPOA =
               POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
         // 独自の POA のポリシーを作成します。PERSISTENT 存続期間ポリシー，
         // USE_SERVANT_MANAGER 要求処理ポリシー，および NON_RETAIN
         // サーバント管理ポリシーが必要です。このサーバント管理ポリシーに
          // サーバントアクティベータではなくサーバントロケータが使用されます
         org.omg.CORBA.Policy[] policies = {
            rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT),

rootPOA.create_servant_retention_policy(ServantRetentionPolicyValue.
               NON_RETAIN),

rootPOA.create_request_processing_policy(RequestProcessingPolicyValue.
               USE_SERVANT_MANAGER)
         };
         // 適切なポリシーで myPOA を作成します。
         POA myPOA = rootPOA.create_POA( "bank_servant_locator_poa",
             rootPOA.the_POAManager(),
             policies );
         // サーバントロケータサーバントを作成し，そのリファレンスを取得します。
         ServantLocator sl = new AccountManagerLocator();
         // 独自の POA にサーバントロケータを設定します。
         myPOA.set_servant_manager(sl);
         org.omg.CORBA.Object ref ;
         // POA マネージャをアクティブ化
         rootPOA.the_POAManager().activate();
         // 参照を生成し，それを書き出します。各当座預金口座と普通預金口座に
         // 参照が 1 つずつあります。ここではサーバントを作成しておらず，
         // 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
         try {
            ref =

myPOA.create_reference_with_id("CheckingAccountManager".getBytes(),
               "IDL:Bank/AccountManager:1.0");
            // 当座のオブジェクト ID を書き出します。
            java.io.PrintWriter pw =
            new java.io.PrintWriter( new java.io.FileWriter("cref.dat") );
            pw.println(orb.object_to_string(ref));
            pw.close();
            ref =

myPOA.create_reference_with_id("SavingsAccountManager".getBytes(),
               "IDL:Bank/AccountManager:1.0");
            // 普通のオブジェクト ID を書き出します。
            pw = new java.io.PrintWriter( new java.io.FileWriter("sref.dat") );
```

```
            pw.println(orb.object_to_string(ref));
            pw.close();
         } catch ( java.io.IOException e ) {
            System.out.println("Error writing the IOR to file ");
            return;
         }
         System.out.println("BankManager is ready.");
         // 着信要求を待機します。
         orb.run();
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

次に，この例でのサーバントマネージャを示します。

```
import org.omg.PortableServer.*;
import org.omg.PortableServer.ServantLocatorPackage.CookieHolder;
public class AccountManagerLocator extends _ServantLocatorLocalBase {
   public Servant preinvoke (byte[] oid,POA adapter,
         java.lang.String operation,
         CookieHolder the_cookie) throws ForwardRequest {
      String accountType = new String(oid);
      System.out.println("¥nAccountManagerLocator.preinvoke called with ID = " +
            accountType + "¥n");
      if ( accountType.equalsIgnoreCase("SavingsAccountManager"))
         return new SavingsAccountManagerImpl();
      return new CheckingAccountManagerImpl();
   }
   public void postinvoke (byte[] oid,
         POA adapter,
         java.lang.String operation,
         java.lang.Object the_cookie,
         Servant the_servant) {
      System.out.println("¥nAccountManagerLocator.postinvoke called with ID = " +
            new String(oid) + "¥n");
   }
}
```

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。

```
POA myPOA = rootPOA.create_POA( "bank_agent_poa",
      rootPOA.the_POAManager(),
      policies );
POA myPOA = rootPOA.create_POA( "bank_agent_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。

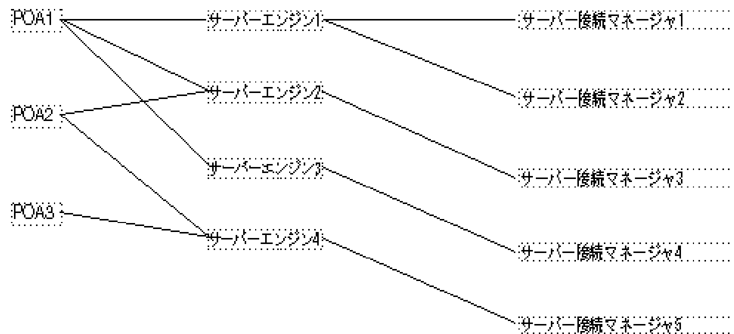
抽象 ServerEngine は，次の項目を制御できます。

- 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

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。
org.omg.CORBA.Any seAny = orb.create_any();
org.omg.CORBA.StringSequenceHelper.insert(seAny, new String[]{"MySE"});
org.omg.CORBA.Policy sePolicy =
orb.create_policy(com.inprise.vbroker.PortableServerExt.SERVER_ENGINE_POLICY_TY
PE.value,
     seAny);
```

```
// POA のポリシーを作成します。
org.omg.CORBA.Policy[] policies = {
    rootPOA.create_lifespan_policy(LifeSpanPolicyValue.PERSISTENT),
    sePolicy
};

// ポリシー付きで POA を作成します。
POA myPOA = rootPOA.create_POA("bank_se_policy_poa", rootPOA.the_POAManager(),
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
```

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。

VisiBroker for Java は，Socket 型と，Socket 型の一種で Java NIO パッケージを使用する Socket_nio だけをサポートします。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。

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。

ガベージコレクション時間は，次のプロパティを使って指定します。

```
vbroker.orb.gcTimeout=<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。

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。

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。

また，VisiBroker では，GIOP のバージョンを指定するためのプロパティもサポートされています。

```
vbroker.se.<se-name>.scm.<scm-name>.listener.giopVersion=<version>
```

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。

Java NIO パッケージを使用するように設定されている場合，VisiBroker for Java は，冷却時間プロパティをサポートします。詳細については，130 ページの「VisiBroker for Java の高スケーラビリティ設定（Java NIO の使用）」を参照してください。

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。

```
// Server.java
import org.omg.PortableServer.*;
public class Server {
   public static void main(String[] args) {
      try {

      // ORB を初期化します。
     org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
      // プロパティマネージャを取得します。
      com.inprise.vbroker.properties.PropertyManager pm =
            ((com.inprise.vbroker.orb.ORB)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", 100);
      pm.addProperty("vbroker.se.mySe.scm.scmlist.manager.connectionMaxIdle",
         300);
      pm.addProperty("vbroker.se.mySe.scm.scmlist.listener.type", "IIOP");
      pm.addProperty("vbroker.se.mySe.scm.scmlist.listener.port", 55000);
      pm.addProperty("vbroker.se.mySe.scm.scmlist.listener.proxyPort", 0);
```

```
      pm.addProperty("vbroker.se.mySe.scm.scmlist.dispatcher.type",
         "ThreadPool");
      pm.addProperty("vbroker.se.mySe.scm.scmlist.dispatcher.threadMax", 100);
      pm.addProperty("vbroker.se.mySe.scm.scmlist.dispatcher.threadMin", 5);
      pm.addProperty("vbroker.se.mySe.scm.scmlist.dispatcher.threadMaxIdle",
         300);
      // ルート POA へのリファレンスを取得します。
      POA rootPOA =
POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
      // 独自のサーバーエンジンポリシーを作成します。
      org.omg.CORBA.Any seAny = orb.create_any();
      org.omg.CORBA.StringSequenceHelper.insert(seAny, new String[]{"mySe"});
      org.omg.CORBA.Policy sePolicy =
      orb.create_policy(

com.inprise.vbroker.PortableServerExt.SERVER_ENGINE_POLICY_TYPE.value,
                  seAny);
         // 永続的 POA のポリシーを作成します。
           org.omg.CORBA.Policy[] policies = {

rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT),sePolicy
      };
      // 適切なポリシーで myPOA を作成します。
      POA myPOA = rootPOA.create_POA("bank_se_policy_poa",
            rootPOA.the_POAManager(),
            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。
      myPOA.activate_object_with_id("BankManager".getBytes(), managerServant);
      // リファレンスを取得します。
      org.omg.CORBA.Object ref = myPOA.servant_to_reference(managerServant);

      // IOR を書き出します。
      try {
         java.io.PrintWriter pw =
               new java.io.PrintWriter( new java.io.FileWriter("ior.dat") );
         pw.println(orb.object_to_string(ref));
         pw.close();
      } catch ( java.io.IOException e ) {
         System.out.println("Error writing the IOR to file ior.dat");
         return;
      }
      // POA マネージャをアクティブ化
      rootPOA.the_POAManager().activate();
      System.out.println(ref + " is ready.");
      // 着信要求を待機します。
      orb.run();
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

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，101 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.java

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，128 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，126 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。

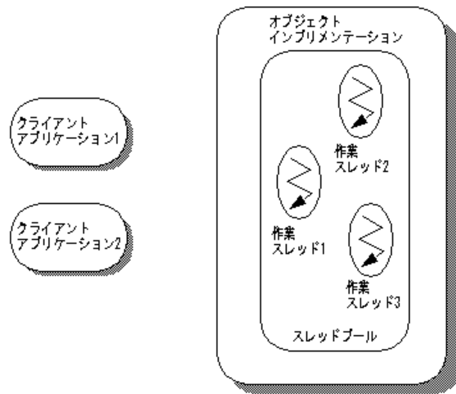
- 最大および最小のスレッド数
- 最大アイドル時間

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。

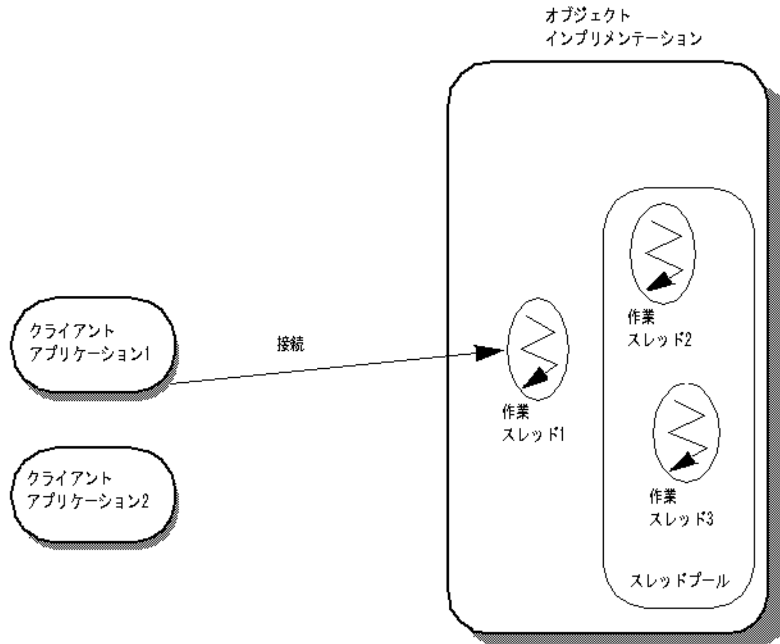
スレッドプールは，デフォルトのスレッドポリシーです。この環境を定義するために必要な設定はありません。スレッドプールのプロパティを設定する場合は，126 ページの「ディスパッチのポリシーとプロパティの設定」を参照してください。

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。

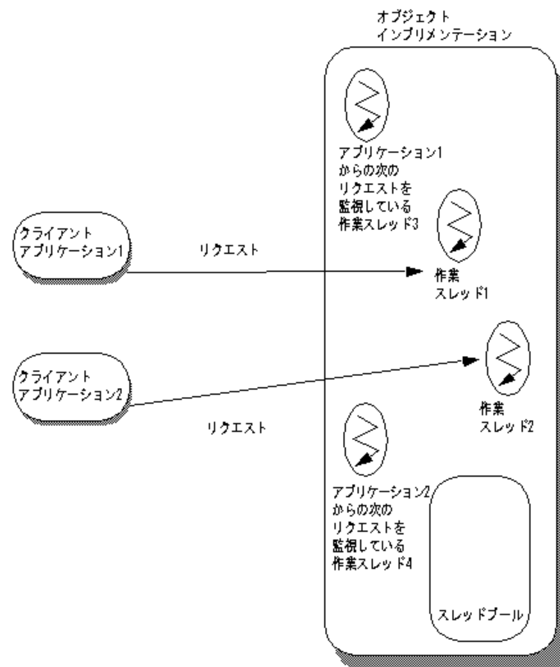
図 10.2　クライアントアプリケーション＃1 による要求の送信



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。

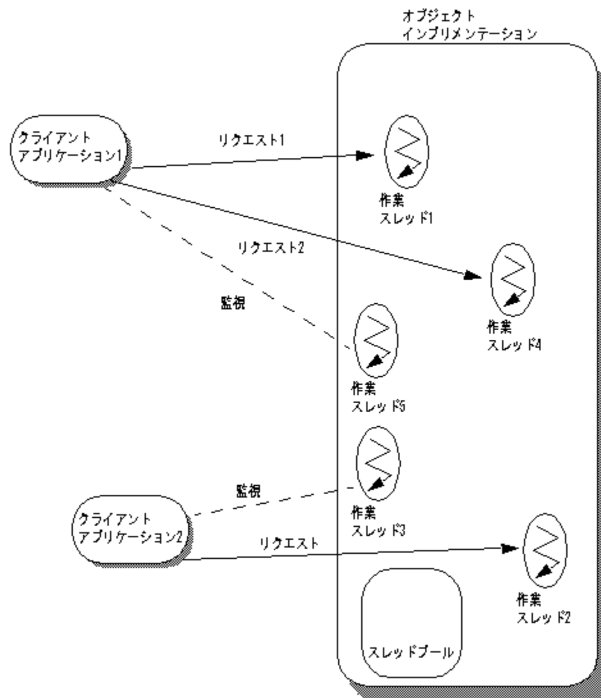
スレッドプールから除去された作業スレッドは，常時要求を監視しています。要求が着信すると，作業スレッドは要求を読み取り，適切なオブジェクトインプリメンテーションにその要求を送ります。作業スレッドが要求をオブジェクトインプリメンテーションに送る前に，その作業スレッドは次の要求を監視する別の作業スレッドを起動します。

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



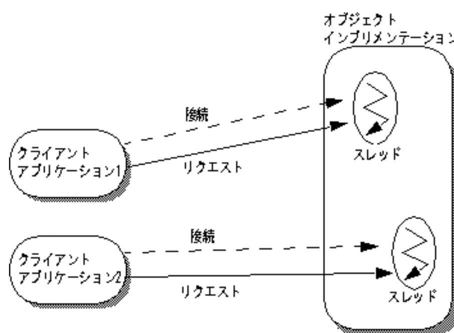
上の図では，クライアントアプリケーション＃1 から 2 番めの要求が着信し，作業スレッド＃4 が使用されています。このとき，新しい要求を監視する作業スレッド＃5 が作成されます。クライアントアプリケーション＃1 からさらに要求が着信すると，監視中のスレッドが要求を受信するたびに新しいスレッドが生成され，要求の処理に割り当てられます。作

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

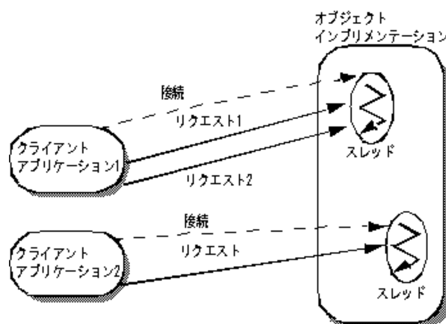
セッションごとのスレッド（TSession）ポリシーでは，クライアントとサーバーのプロセス間の接続によってスレッドが駆動されます。サーバーでセッションごとのスレッドポリシーを選択すると，新しいクライアントがサーバーに接続するたびに，新しいスレッドが割り当てられます。特定のクライアントから受信したすべての要求を処理するために，1 つのスレッドが割り当てられます。このため，セッションごとのスレッドポリシーは，接続ごとのスレッドポリシーとも呼ばれます。クライアントがサーバーから切断すると，スレッドは破棄されます。クライアント接続に割り当てることのできる最大のスレッド数を制限する場合は，vbroker.se.iiop_ts.scm.iiop_ts.manager.connectionMax 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



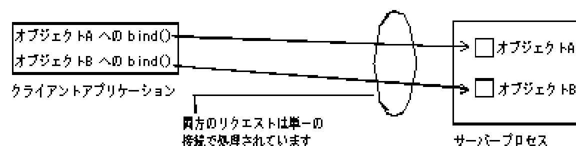
上の図では，クライアントアプリケーション #1 からオブジェクトインプリメンテーションに 2 番めの要求が着信しています。この要求 2 は，要求 1 を処理したスレッドと同じスレッドによって処理されます。このため，このスレッドは，要求 1 の処理を完了するまで，要求 2 をブロックします。セッションごとのスレッドポリシーでは，同じクライアントからの要求は同時に処理されません。要求 1 が完了すると，スレッドはクライアントアプリケーション #1 からの要求 2 を処理します。クライアントアプリケーション #1 から複数の要求を着信することができます。この複数の要求は着信順に処理され，クライアントアプリケーション #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。

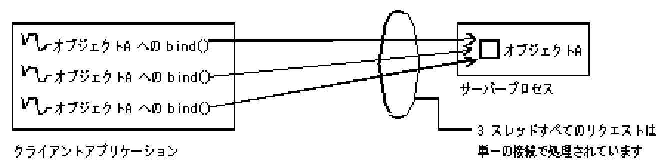
次の例では，サーバープロセス内の 2 つのオブジェクトに 1 つのクライアントアプリケーションがバインドされています。`bind()` ごとに対象となるサーバープロセス内のオブジェクトは異なりますが，`bind()` は，サーバープロセスへの接続は共有しています。

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。

  このプロパティは，特定の条件で VisiBroker for Java に適用されます。詳細については，130 ページの「VisiBroker for Java の高スケーラビリティ設定（Java NIO の使用）」を参照してください。VisiBroker for Java では，このプロパティのデフォルト値は 0 です。これは，すぐにサービスに利用できる新しい要求がない場合，サービスを提供されている GIOP 接続が「ホット」でなくなることを示します。テストによってアプリケーションのパフォーマンスに効果があることが示されない限り, coolingTime の値をデフォルト以外の値に変更しないことをお勧めします。

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。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

  このプロパティのほかの有効な値は，Socket_nio だけです。

- vbroker.se.xxx.scm.yyy.listener.type=IIOP

  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 0，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。VisiBroker for Java では，このプロパティは，サーバー接続マネージャが Socket_nio というマネージャタイプを持つ場合にだけ適用されます。詳細については，130 ページの「VisiBroker for Java の高スケーラビリティ設定（Java NIO の使用）」を参照してください。

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。

VisiBroker for Java の高スケーラビリティ設定（Java NIO の使用）

J2SE 1.4 の Java NIO パッケージを使用すると，接続ごとに専用のスレッドを使用する必要がなく，サーバーは複数の接続を効率的に処理できます。これにより，サーバーは，少ないスレッドで多くのクライアント接続を処理でき，より大きなスケーラビリティを得ることができます。VisiBroker for Java サーバーは，Java NIO 技術を利用するように設定できます。ThreadPool ポリシーを使用するサーバーは，マネージャタイプを Socket ではなく Socket_nio に設定することで，Java NIO を使用できます。たとえば，次のようにします。

```
vbroker.se.iiop_tp.scm.iiop_tp.manager.type=Socket_nio
```

この機能は，threadMax プロパティとともに使用する必要があります。threadMax プロパティは，要求のディスパッチ（呼び出しの処理）に使用するスレッドプール内のスレッドの数を制限するために使用します。マネージャタイプが Socket_nio である場合，スレッドプール内のスレッド数は，処理される接続数に比例して（指定された threadMax の数を超えて）増えることはありません。これは，必ずしも接続ごとにスレッドが必要でないためです。

接続ごとのスレッドのモデル（VisiBroker for Java スレッドプールのデフォルト）は，接続数が比較的少ない（接続数が数百のレベルに達しない）場合に，サーバーのパフォーマンスで NIO ベースのモデルより優れることが期待されます。適切なモデルを決定するには，アプリケーションに典型的な負荷状態を与えてテストを実行することをお勧めします。

J2SE 1.4 以上を使用するサーバーは，この機能を使用できます。現在のところ，VisiBroker for Java ベースのクライアントは，ORB で Java NIO を使用しても有利になりません。

NIO ベースのディスパッチが有効である場合，VisiBroker for Java には coolingTime プロパティが効果的です。詳細については，126 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。

ガベージコレクション

VisiBroker for Java ORB は，メモリ以外のさまざまなリソースにガベージコレクションを自動的に実行します。メモリのガベージコレクションは，Java 仮想マシンによって実行されます。ガベージコレクション期間を制御するために，さまざまなプロパティが提供されています。さらに，スレッドや接続などのリソースには，これらのリソースのガベージコレクションを制御するタイムアウトプロパティが定義されています。

ORB ガベージコレクションの動作

ORB ガベージコレクタスレッドは，通常の優先順位のスレッドです。タイムアウト（vbroker.orb.gcTimeout プロパティで指定）になると，スレッドが起動され，アイドル状態ですべての未使用リソースにガベージコレクションが実行されます。ガベージコレクションの対象になるクラスは，自分自身をガベージコレクタに登録します。そのようなクラスは，*コレクタブル*と呼ばれます。コレクタブルの典型的な例は，スレッドと接続です。ほかに，GateKeeper のキャッシュなど，さまざまなキャッシュに対するタイムアウトなどがあります。ほとんどのコレクタブルは，ガベージコレクションが実行されるときに，保持していたリソースを無効にするか，適切に解放します（接続を閉じる，スレッドの run メソッドを終了するなど）。これらのリソースは，後で Java ガベージコレクタによって再回収されます。

メモ　ORB ガベージコレクタは内部サービスであり，ユーザーに公開されていません。

ORB ガベージコレクションに関連するプロパティ

ガベージコレクション期間を制御するメインプロパティは vbroker.orb.gcTimeout です。タイムアウト値は秒単位であり，デフォルト値は 30 秒です。

スレッドと接続には，*アイドルタイムアウト*のプロパティが定義されています。たとえば，スレッドプールディスパッチャには，次のプロパティが定義されています。

```
vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.threadMaxIdle
```

値は秒単位です。デフォルト値の 300 秒を過ぎると，スレッドはスレッドプールから削除されます。同様に，デフォルトのサーバー接続マネージャ（iiop_tp）には，接続のアイドルタイムアウトプロパティが定義されています。

```
vbroker.se.iiop_tp.scm.iiop_tp.manager.connectionMaxIdle
```

値は秒単位です。デフォルト値の 0 は，接続のアイドル状態の長さに関係なく，接続が継続することを示します。ただし，接続が切断されると，ORB は接続へのすべてのリファレンスを削除し，リソースは後で Java ガベージコレクタによって回収されます。ORB ガベージコレクタは，connectionMaxIdle プロパティが 0 以外の値に設定された接続だけを回収します。

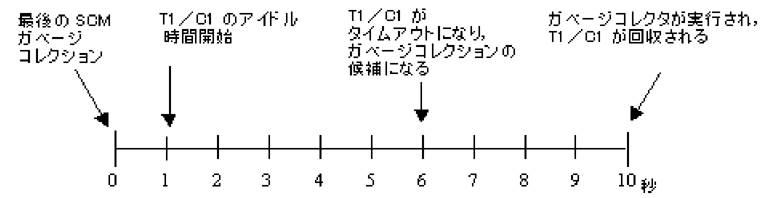
さまざまなタイムアウトのプロパティと vbroker.orb.gcTimeout プロパティは微妙な関係にあります。たとえば，次のプロパティが指定されているとします。

```
vbroker.orb.gcTimeout=10
vbroker.se.iiop_tp.scm.iiop_tp.dispatcher.threadMaxIdle=5
vbroker.se.iiop_tp.scm.iiop_tp.manager.connectionMaxIdle=5
```

ここでは，ガベージコレクションのタイムアウト期間が 10 秒に設定され，スレッドと接続のタイムアウトが 5 秒に設定されています。次の図は，これらのプロパティの相互関係を示します。ここでは，スレッド T1 と接続 C1 がアイドル状態になり，回収されています。

メモ　ここでは，ORB ガベージコレクタがちょうど 10 秒後に実行されるように示されていますが，実際は，JVM によるガベージコレクタ（GC）スレッドのスケジュールによって異なります。

図 10.9 ORB GC によるリソースの回収



T1 と C1 は，ガベージコレクションの候補である場合でも，ORB ガベージコレクタの実行時にだけ回収されます。それまで，T1 と C1 はタイムアウト状態で残ります。

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 Java 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，`idl2java` 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，`org.omg.PortableServer.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，`org.omg.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 `org.omg.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
```

サーバークラスの変更

次のサンプルコードは，Server クラスに対する変更を示しています。AccountManagerManagerPOATie のインスタンスを作成する手順が追加されています。

```
import org.omg.PortableServer.*;

public class Server {
  public static void main(String[] args) {
    try {
      // ORB を初期化します。
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
      // ルート POA へのリファレンスを取得します。
      POA rootPOA = POAHelper.narrow(
        orb.resolve_initial_references("RootPOA"));
      // 永続的 POA のポリシーを作成します。
      org.omg.CORBA.Policy[] policies = {
        rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)
      };
      // 適切なポリシーで myPOA を作成します。
      POA myPOA = rootPOA.create_POA("bank_agent_poa",
        rootPOA.the_POAManager(), policies);
      // AccountManagerImpl のインスタンスにデリゲートする tie を作成します。
      Bank.AccountManagerPOATie tie =
        new Bank.AccountManagerPOATie(new AccountManagerImpl(rootPOA));
      // サーバントの ID を決定します。
      byte[] managerId = "BankManager".getBytes();
      // 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, tie);
      // POA マネージャをアクティブ化
      rootPOA.the_POAManager().activate();
      System.out.println("Server is ready.");
      // 着信要求を待機します。
      orb.run();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

AccountManager の変更

AccountManager クラスには，次の変更が加えられています。Bank_agent のサンプルコードと対比してください。

- AccountManagerImpl は Bank.AccountManagerPOA を拡張しなくなります。
- 新しい Account を作成するとき，AccountPOATie も作成および初期化します。

```
import org.omg.PortableServer.*;
import java.util.*;

public class AccountManagerImpl implements
    Bank.AccountManagerOperations {
  public AccountManagerImpl(POA poa) {
    _accountPOA = poa;
  }
  public synchronized Bank.Account open(String name) {
    // account ディレクトリ内で口座を検索します。
    Bank.Account account = (Bank.Account) _accounts.get(name);
    // ディクショナリに口座が 1 つもない場合は，作成します。
    if (account == null) {
      // 0 ～ 1000 ドルの範囲で口座に残高を設定します。
      float balance = Math.abs(_random.nextInt()) % 100000 / 100f;
      // AccountManagerImpl のインスタンスにデリゲートする tie を作成します。
      Bank.AccountPOATie tie =
        new Bank.AccountPOATie(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。
        account =
          Bank.AccountHelper.narrow(_accountPOA.servant_to_reference(tie));
      }
      catch (Exception e) {
        e.printStackTrace();
      }
      // 新しい口座を出力します。
      System.out.println("Created " + name +
        "'s account: " + account);
      // 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 account;
  }
  private Dictionary _accounts = new Hashtable();
  private Random _random = new Random();
  private POA _accountPOA = null;
}
```

Account クラスの変更

Account クラスに加えられた変更（Bank のサンプルコードと対比）によって Bank.AccountPOA を拡張しなくなります。

```
// Server.java
public class AccountImpl implements Bank.AccountOperations {
  public AccountImpl(float balance) {
    _balance = balance;
  }
  public float balance() {
    return _balance;
  }
  private float _balance;
}
```

tie サンプルのビルド

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 は，システムリソースを集中的に使用するため，VisiBroker ORB のインスタンスは，1 プロセスにつき 1 つだけ作成することをお勧めします。

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「Java 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，`args` 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。

```
public class Client {
  public static void main (String[] args) {
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
    . . .
}
```

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>Helper` の `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。

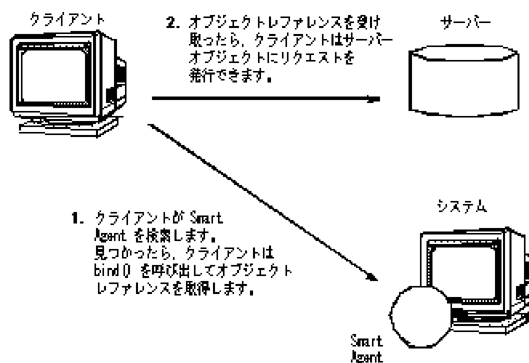
バインド処理中に実行される動作

サーバーのプロセスが起動すると，サーバーは，`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。

```
Bank.AccountManager manager =
  Bank.AccountManagerHelper.bind(orb,
  "/bank_agent_poa",
  "BankManager".getBytes());
```

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，139 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。
System.out.println("The balance in Account1: $" + account1.balance());
```

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 org.omg.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。

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。

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
_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

_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，true 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，false 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。

instanceof キーワードを使用して，実行時の型を判定することはできません。

_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 true 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，false 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
_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 true を返し，オブジェクトがバインドされていない場合は false 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，true 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，true 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。

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。

Java のキャスト機能を使ってナローイングを実行することはできません。

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。

ナローイングに失敗した場合は，IDL 例外 CORBA::BAD_PARAM が生成されます。オブジェクトリファレンスでは，要求対象の型をサポートしていないからです。

```
public abstract class AccountManagerHelper {
  . . .
  public static Bank.AccountManager narrow(org.omg.CORBA.Object object) {
  . . .
  }
. . .
}
```

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 account;
org.omg.CORBA.Object obj;
account = AccountHelper.bind();
obj = (org.omg.CORBA.Object) account;
. . .
```

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。
- 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。

org.omg.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。

com.borland.vbroker.CORBA.Object（Borland）

このインターフェースを使用するには，org.omg.CORBA.Object を com.borland.vbroker.CORBA.Object にキャストする必要があります。このインターフェースは org.omg.CORBA.Object から派生しているので，org.omg.CORBA.Object で定義されているメソッドのほかに，次のメソッドも使用できます。

- _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。

org.omg.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，org.omg.CORBA.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，org.omg.CORBA.NO_PERMISSION が生成されます。この要求にって指定された 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。

org.omg.CORBA.PolicyCurrent

PolicyCurrent インターフェースは，新しいメソッドを追加しないで PolicyManager から派生させます。このインターフェースにより，スレッドレベルでオーバーライドされたポリシーへのアクセスが可能になります。PolicyCurrent 識別子を指定して org.omg.CORBA.ORB.resolve_initial_references を呼び出すと，スレッドの PolicyCurrent へのリファレンスを取得できます。

com.borland.vbroker.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。
boolean deferMode = true;
Any policyValue= orb.create_any();
policyValue.insert_boolean(deferMode);

Policy policies =
  orb.create_policy(DEFER_BIND_POLICY_TYPE.value, policyValue);

// スレッドマネージャへのリファレンスを取得します。
PolicyManager orbManager =
  PolicyManagerHelper.narrow(
  orb.resolve_initial_references("ORBPolicyManager"));

// ORB レベルでポリシーを設定します。
orbManager.set_policy_overrides(new Policy[] {policies},
  SetOverrideType.SET_OVERRIDE);

// バインドメソッドを取得します。
byte[] managerId = "BankManager".getBytes();
Bank.AccountManager manager =
  Bank.AccountManagerHelper.bind(orb, "/qos_poa", managerId);
```

com.borland.vbroker.QoSExt.ExclusiveConnectionPolicy

ExclusiveConnectionPolicy は，指定されたサーバーオブジェクトへの排他的（非共有）接続を有効にする VisiBroker 固有のポリシーです。このポリシーには，ブール値 true または false を代入します。このポリシーが true の場合，サーバーオブジェクトへの接続は排他的になります。false の場合，既存の接続を再利用できる場合は再利用され，再利用できない場合にだけ新しい接続が確立されます。デフォルト値は，false です。

このポリシーは，VisiBroker 3.x の Object._clone() と同じ機能を提供します。

排他的接続と非排他的接続を確立する例は，CloneClient.java サンプルで示します。このサンプルは次の場所にあります。

```
<install_dir>¥examples¥vbe¥QoS_policies¥qos¥
```

com.borland.vbroker.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。

次のサンプルコードは，RelativeConnectionTimeoutPolicy の作成方法を示します。

```
Any connTimeoutPolicyValue =  orb.create_any();
  // 入力は，100 ナノ秒単位で指定します。
  // 20 秒の値を指定するには，入力として「20 * 10^7」ナノ秒を設定します。

  int connTimeout = 20;

  connTimeoutPolicyValue.insert_ulonglong(connTimeout * 10000000);
  org.omg.CORBA.Policy ctoPolicy =
    orb.create_policyRELATIVE_CONN_TIMEOUT_POLICY_TYPE.value,
    connTimeoutPolicyValue);
  PolicyManager orbManager = PolicyManagerHelper.narrow (
    orb.resolve_initial_references("ORBPolicyManager"));

orbManager.set_policy_overrides(new Policy[] ¥{ctoPolicy¥},
  SetOverrideType.SET_OVERRIDE);
```

org.omg.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，org.omg.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。

- org.omg.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。次のサンプルコードは，TRANSPARENT 型の RebindPolicy を作成し，VisiBroker ORB レベル，スレッドレベル，およびオブジェクトレベルでポリシーを設定する例を示しています。
- org.omg.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。
- org.omg.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。

- com.borland.vbroker.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 com.borland.vbroker.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。
- com.borland.vbroker.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。

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。

次のサンプルコードでは，型が TRANSPARENT の RebindPolicy を作成し，VisiBroker ORB レベル，スレッドレベル，およびオブジェクトレベルでそのポリシーを設定します。

```
Any policyValue= orb.create_any();
RebindModeHelper.insert(policyValue,
  org.omg.Messaging.TRANSPARENT.value);
Policy myRebindPolicy = orb.create_policy(REBIND_POLICY_TYPE.value,
  policyValue);
//ORB ポリシーマネージャへのリファレンスを取得します。
org.omg.CORBA.PolicyManager manager;
try {
```

```
    manager =
PolicyManagerHelper.narrow(orb.resolve_initial_references("ORBPolicyManager"));
}
catch(org.omg.CORBA.ORBPackage.InvalidName e) {}
// スレッドごとのマネージャへのリファレンスを取得します。
org.omg.CORBA.PolicyManager current;
try {
  current =
    PolicyManagerHelper.narrow(orb.resolve_initial_references
      ("PolicyCurrent"));
}
catch(org.omg.CORBA.ORBPackage.InvalidName e) {}
//ORB レベルでポリシーを設定します。
try{
  manager.set_policy_overrides(myRebindPolicy,
    SetOverrideType.SET_OVERRIDE);
}
catch (InvalidPolicies e){}
// スレッドレベルでポリシーを設定します。
try {
  current.set_policy_overrides(myRebindPolicy,
    SetOverrideType.SET_OVERRIDE);
}
catch (InvalidPolicies e){}
// オブジェクトレベルでポリシーを設定します。
org.omg.CORBA.Object oldObjectReference=bind(...);
org.omg.CORBA.Object newObjectReference=oldObjectReference._set_policy_override
         (myRebindPolicy, SetOverrideType.SET_OVERRIDE);
```

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。

org.omg.CORBA.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。

次のコードは，RelativeRequestTimeoutPolicy の作成方法を示しています。

```
// 要求タイムアウトを 100 ナノ秒単位で指定します。
// タイムアウトを 20 秒に設定するには，「20 * 10^7」を設定します。
int reqTimeout = 20;
Any policyValue =  orb.create_any();
policyValue.insert_ulonglong(reqTimeout * 10000000);
// ポリシーを作成します。
org.omg.CORBA.Policy reqPolicy = orb.create_policy(
              RELATIVE_REQ_TIMEOUT_POLICY_TYPE.value, policyValue);
PolicyManager orbManager = PolicyManagerHelper.narrow(
             orb.resolve_initial_references("ORBPolicyManager"));
orbManager.set_policy_overrides(new Policy[] {reqPolicy},
                                SetOverrideType.SET_OVERRIDE);
```

org.omg.CORBA.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。

次のサンプルコードは，RelativeRoundTripTimeoutPolicy の作成方法を示します。

```
// 往復タイムアウトを 100 ナノ秒単位で指定します。
   // タイムアウトを 50 秒に設定するには,「50 * 10^7」を設定します。
    int rttTimeout = 50;
    Any policyValue =  orb.create_any();
    policyValue.insert_ulonglong(rttTimeout * 10000000);
    //RelativeRoundTripTimeoutPolicy を作成し，それを ORB レベルで設定します。
    org.omg.CORBA.Policy rttPolicy = orb.create_policy(
                    RELATIVE_RT_TIMEOUT_POLICY_TYPE.value, policyValue);
    PolicyManager orbManager = PolicyManagerHelper.narrow(
                 orb.resolve_initial_references("ORBPolicyManager"));
    orbManager.set_policy_overrides(new Policy[] {rttPolicy},
                                  SetOverrideType.SET_OVERRIDE);
```

org.omg.CORBA.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
org.omg.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。
org.omg.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。
org.omg.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。
org.omg.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だからです。

コードセットサポート

VisiBroker は，char または wchar IDL データ型をマーシャリングする際に，アプリケーションが共通のコードセットについて取り決めるためのコードセットネゴシエーションを提供します。コードセットは，文字セットの各文字とそのビット表記または数値との 1 対 1 関係を確立する明確な規則の集合です。

コードセットのタイプ

コードセットにはいくつかのタイプがあります。一部の言語環境では，バイト単位とワイド文字が区別されます。バイト単位文字は，いくつかの 8 ビットバイト単位でエンコードされます。ASCII（英語などの西ヨーロッパ言語で使用）は，典型的な 1 バイトのエンコードの例です。各文字に 1 ～ 3 バイトを使用する一般的なマルチバイトエンコードは，日本のワークステーションで使用される eucJP（日本語拡張 UNIX コード）です。UTF-8 などのバイト単位コードセットでは，1 文字の表現に 1 ～ 6 バイトが使用されますが，CORBA 仕様では，char データに依然として 1 バイトのサイズ制限があり，1 バイトを超える表現には char[] を使用する必要があります。

ワイド文字は 16 または 32 ビット固定長であり，中国語や日本語で使用されます。中国語や日本語では，8 ビットの組み合わせでは数が不十分であり，固定幅のエンコードが必要です。典型的な例は，Unicode（Unicode コンソーシアムの定義による「ユニバーサル」文字）です。Unicode 文字の拡張エンコードスキームは，UTF-16（UCS 変換フォーマット，16 ビット表記）です。

ネイティブコードセット

ネイティブコードセットは，クライアントまたはサーバーが ORB と通信するために使用するコードセットです。char データと wchar データに個別のネイティブコードセットが存在します。

変換コードセット（CCS）

これは，ORB がネイティブコードセットとの間ですべてのエンコードの変換を行うことができるターゲットコードセットの集まりです。この CCS 内の各コードセットに対して，ORB は適切な変換プロシージャを維持しています。また，転送されたデータに対してネイティブコードセットのほかにそのコードセットを使用できることを宣言します。

転送コードセット（TCS）

クライアントの ORB とサーバーの ORB との間で文字データの転送に使用されるエンコーディングとして，一般に合意されるのが転送コードセットです。クライアントとそのサーバーの間でセッションごとに 2 つの転送コードセットが確立されます。1 つは char データ（TCS-C）に使用され，もう 1 つは wchar データ（TCS-W）に使用されます。

コードセットネゴシエーション

クライアント側の ORB は，IOR マルチコンポーネントプロファイル構造体からサーバーのネイティブコードセットと変換コードセットを決定すると同時に，クライアントのネイティブコードセットと変換コードセットを決定します。この情報から，クライアント側の ORB は，char および wchar の転送コードセット（TCS-C および TCS-W）を選択します。要求と応答の両方で，char TCS-C によって char データと string データのエンコードが決定され，wchar TCS-W によって wchar データと wstring データのエンコードが決定されます。

サポートされるコードセット

VisiBroker は，次のコードセットをサポートします。

- IDL `char` データ型の場合，ネイティブコードセットは ISO 8859-1（Latin-1），サポートされる変換コードは UTF-8 です。
- IDL `wchar` データ型の場合，ネイティブコードセットは UTF-16，変換コードセットはありません。

クライアントランタイムによるクライアント専用アプリケーションの配布

多くのアプリケーション配布シナリオで，フルサイズの ORB インプリメンテーションではなく，クライアントランタイムだけを必要とすることがよくあります。アプリケーションが純粋なクライアントであり，POA の作成やオブジェクトのアクティブ化などのサーバー側機能がない場合は，このような配布シナリオで VisiBroker のクライアントランタイムライブラリを使用できます。VisiBroker クライアントランタイムのメモリフットプリントは，完全な VisiBroker インプリメンテーションに比べて小さくなります。クライアントランタイムは Java アーカイブ（vbjclientorb.jar）ファイルとして提供され，VisiBroker のインストール場所の /lib ディレクトリにインストールされています。

メモ　クライアントランタイムは，ORB の機能を完全にはサポートしていません。

VisiBroker クライアントランタイムライブラリでは，次の機能がサポートされています。

- リモートのサーバーやサービスのオペレーションを呼び出すなどのクライアント側の機能が提供されます。クライアントランタイムを使用するアプリケーションは，インターフェースリポジトリ，ネーミングサービス，リクエストエージェント（ポーリングモードのみ）などのサービスを引き続き利用できます。また，ファイアウォールを通過するために GateKeeper も利用できます。オブジェクトアクティベーションデーモン（OAD）に登録されているサーバーのオペレーションも呼び出すことができます。OSAgent を使ってサーバーを検索することもできます。
- バインドインターセプタなどのクライアント側インターセプタと，リクエストインターセプタ（VisiBroker 4x とポータブルインターセプタの両方）を使用できます。
- VisiSecure のクライアント側の機能も使用できます。

VisiBroker クライアントランタイムライブラリでは，次の機能がサポートされていません。

- POA の作成やオブジェクトのアクティブ化などのサーバー側の機能は，すべて使用できません。`resolve_initial_references("RootPOA")` は使用できません。
- 通知，イベントサービス，およびリクエストエージェントのコールバックモードは使用できません。
- ロケーションサービスはサポートされていません。
- POALifeCycleInterceptor，リクエストインターセプタ（VisiBroker 4x とポータブルインターセプタの両方），IOR インターセプタなどのサーバー側のインターセプタは，すべて使用できません。ただし，クラスパスに追加のセキュリティ JAR ファイルが含まれている必要があります。次の 149 ページの「使い方」を参照してください。

使い方

vbjclientorb.jar を使用するには，<install_dir>/bin/vbj.config を変更して，vbjclientorb.jar の addpath エントリを設定します。それには，vbj.config ファイルの次の行を

```
addpath $var(defaultJarPath)/vbjorb.jar
```

次の行に置き換えます。

```
addpath $var(defaultJarPath)/vbjclientorb.jar
```

クライアントアプリケーションで VisiSecure を使用する場合は，vbsec.jar，sunjce_provider.jar，local_policy.jar，US_export_policy.jar もクラスパスに存在する必要が

あります。JDK 1.3.1 を使用する場合は，以上の JAR に加えて，jsse.jar，jcert.jar，jnet.jar，jaas.jar，jce1_2_1.jar の各 JAR ファイルもクラスパスに存在する必要があります。

メモ　クライアントランタイム（vbjclientorb.jar）によって特定の機能がサポートされていない場合は，実行時に ClassNotFound または NoClassDefFound 例外とともに次の標準エラーメッセージが出力されます。

```
"******Client runtime does not support full ORB functionality ******"
```

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，Java 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「Java 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。idl2java 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);
};
```

生成されるコードの概要

IDL コンパイラは，上の 152 ページの「IDL 仕様のサンプル」から複数のファイルを生成します。

- _exampleStub.java は，クライアント側の example オブジェクトのスタブコードです。
- example.java は，example インターフェースの宣言です。
- exampleHelper.java は，exampleHelper クラスを宣言します。このクラスは，ユーティリティ機能と example インターフェースに関するサポート機能を定義します。
- exampleHolder.java は，exampleHolder クラスを宣言します。このクラスは，out パラメータと inout パラメータを渡すためのホルダーを提供します。
- exampleOperations.java は example インターフェースのメソッドを定義し，クライアント側とサーバー側の両方で使用します。また，tie クラスとともに機能して tie メカニズムを提供します。
- examplePOA.java には，サーバー側の example オブジェクトのスケルトンコード（インプリメンテーションベースコード）が含まれています。
- examplePOATie.java には，tie メカニズムを使って example オブジェクトをサーバー側に実装するために使用されるクラスが含まれています。

_<interface_name>Stub.java

ユーザー定義型ごとに，idl2java コンパイラは 1 つのスタブクラスを作成します。これはクライアント側でインスタンス化され，<interface_name> インターフェースを実装するクラスです。

```
public class exampleStub extends com.inprise.vbroker.CORBA.portable.ObjectImpl
  implements example {
  final public static java.lang.Class _opsClass = exampleOperations.class;
  public java.lang.String[] ids () {
      . . .
  }
  public int op1 (char x, org.omg.CORBA.ShortHolder y) {
      . . .
}
```

<interface_name>.java

<interface_name>.java ファイルは，各 IDL インターフェース用に生成された Java インターフェースです。これは，IDL インターフェース定義を適切な Java インターフェースに直接マッピングしたものです。このインターフェースは，クライアントとサーバースケルトンの両方で実装されます。

```
public interface example extends com.inprise.vbroker.CORBA.Object,
  exampleOperations,
  org.omg.CORBA.portable.IDLEntity {
}
```

<interface_name>Helper.java

ユーザー定義型ごとに，idl2java は 1 つのヘルパークラスが作成されます。Helper クラスは, 生成される Java インターフェースのさまざまな静的メソッドを保持する抽象クラスです。

```
public final class exampleHelper {
  public static example narrow (final org.omg.CORBA.Object obj) {
    . . .
  }
  public static example unchecked_narrow (org.omg.CORBA.Object obj) {
    . . .
  }
  public static example bind (org.omg.CORBA.ORB orb) {
    . . .
  }
  public static example bind (org.omg.CORBA.ORB orb,
    java.lang.String name) {
    . . .
  }
  public static example bind (org.omg.CORBA.ORB orb, java.lang.String name,
    java.lang.String host,
    com.inprise.vbroker.CORBA.BindOptions _options) {
     . . .
  }
  public static example bind (org.omg.CORBA.ORB orb, java.lang.String
    fullPoaName,
    byte[] oid) {
    . . .
  }
  public static example bind (org.omg.CORBA.ORB orb,
    java.lang.String fullPoaName, byte[] oid,
    java.lang.String host,
    com.inprise.vbroker.CORBA.BindOptions _options) {
    . . .
  }
  public java.lang.Object read_Object (final org.omg.CORBA.portable.
    InputStream istream) {
    . . .
  }
  public void write_Object (
    final org.omg.CORBA.portable.OutputStream ostream,
    final java.lang.Object obj) {
    . . .
  }
  public java.lang.String get_id () {
    . . .
  }
  public org.omg.CORBA.TypeCode get_type () {
    . . .
  }
  public static example read (
    final org.omg.CORBA.portable.InputStream _input) {
```

```
      . . .
    }
    public static void write (
      final org.omg.CORBA.portable.OutputStream _output,
      final example value) {
      . . .
    }
    public static void insert (
      final org.omg.CORBA.Any any, final example value) {
      . . .
    }
    public static example extract (final org.omg.CORBA.Any any) {
      . . .
    }
    public static org.omg.CORBA.TypeCode type () {
      . . .
    }
    public static java.lang.String id () {
      . . .
    }
  }
```

<interface_name>Holder.java

ユーザー定義型ごとに，idl2java コンパイラは 1 つのホルダークラスを作成します。これは，out パラメータや inout パラメータとして渡されて，<interface_name> インターフェースをサポートするオブジェクトをラップするオブジェクトのクラスを提供します。

```
public final class exampleHolder
  implements org.omg.CORBA.portable.Streamable {
    public foo.example value;
    public exampleHolder () {
    }
    public exampleHolder (final foo.example _vis_value) {
       . . .
    }
    public void _read (final org.omg.CORBA.portable.InputStream input) {
       . . .
    }
    public void _write (final org.omg.CORBA.portable.OutputStream output) {
       . . .
    }
    public org.omg.CORBA.TypeCode _type () {
       . . .
    }
}
```

<interface_name>Operations.java

ユーザー定義型ごとに，idl2java コンパイラは IDL 宣言で定義されているすべてのメソッドを保持するオペレーションクラスを作成します。

```
public interface exampleOperations {
  public int op1(char x, org.omg.CORBA.ShortHolder y);
}
```

<interface_name>POA.java

<interface_name>POA.java ファイルは，インターフェースのサーバー側スケルトンです。これは in パラメータをアンマーシャリングしてオブジェクトインプリメンテーションへの呼び出しに渡します。また，戻り値とすべての out パラメータをマーシャリングし直します。

```
public abstract class examplePOA
  extends org.omg.PortableServer.Servant
  implements org.omg.CORBA.portable.InvokeHandler, exampleOperations {
  public example _this () {
     . . .
  }
  public example _this (org.omg.CORBA.ORB orb) {
     . . .
  }
  public java.lang.String[] _all_interfaces (
    final org.omg.PortableServer.POA poa,
     . . .
  }
  public org.omg.CORBA.portable.OutputStream _invoke (java.lang.String opName,
    org.omg.CORBA.portable.InputStream _input,
    org.omg.CORBA.portable.ResponseHandler handler) {
    . . .
  }
  public static org.omg.CORBA.portable.OutputStream _invoke (exampleOperations _self,
    int _method_id, org.omg.CORBA.portable.InputStream _input,
    org.omg.CORBA.portable.ResponseHandler _handler) {
     . . .
  }
}
```

<interface_name>POATie.java

<interface_name>POATie.java ファイルは，<interface_name> インターフェースのデリゲートインプリメンテーションです。tie クラスの各インスタンスは，すべてのオペレーションのデリゲート先となる <interface_name>Operations クラスを実装するインプリメンテーションクラスのインスタンスとともに初期化する必要があります。

```
public class examplePOATie extends examplePOA {
  public examplePOATie (final exampleOperations _delegate) {
     . . .
  }
  public examplePOATie (final exampleOperations _delegate,
    final org.omg.PortableServer.POA _poa) {
     . . .
  }
  public exampleOperations _delegate () {
     . . .
  }
  public void _delegate (final exampleOperations delegate) {
     . . .
  }
  public org.omg.PortableServer.POA _default_POA () {
     . . .
  }
  public int op1 (char x, org.omg.CORBA.ShortHolder y) {
     . . .
  }
}
```

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。

```
public interface TestOperations {
  public int count ();
  public void count (int count);
  public java.lang.String 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。

```
public interface parentOperations {
  public void operation1 ();
}
public interface childOperations extends parentOperations {
  public int operation2 (short s);
}
public interface parent
  extends com.inprise.vbroker.CORBA.Object, parentOperations,
    org.omg.CORBA.portable.IDLEntity {
}
public interface child extends childOperations, Baz.parent,
  org.omg.CORBA.portable.IDLEntity {
}
```

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，95 ページの「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，167 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。

例を次に示します。

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> vbj -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。

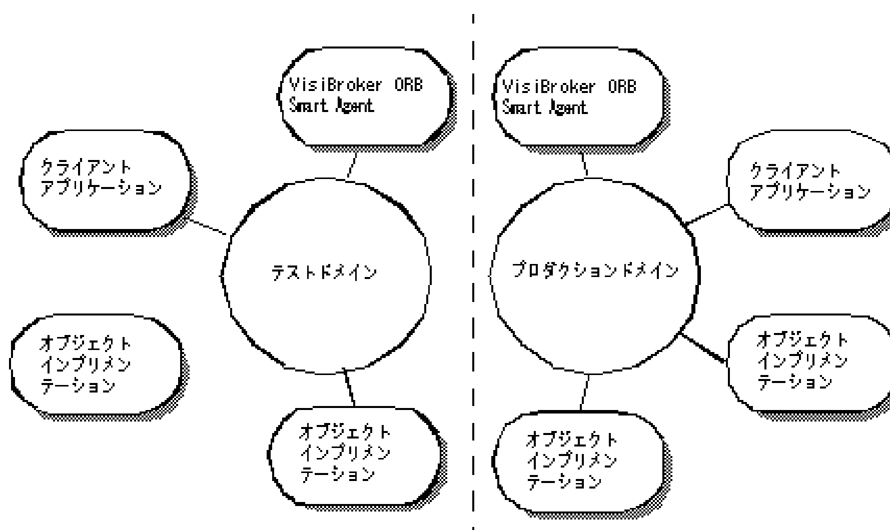
クライアントをポーリングする間隔は変更できません。

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。

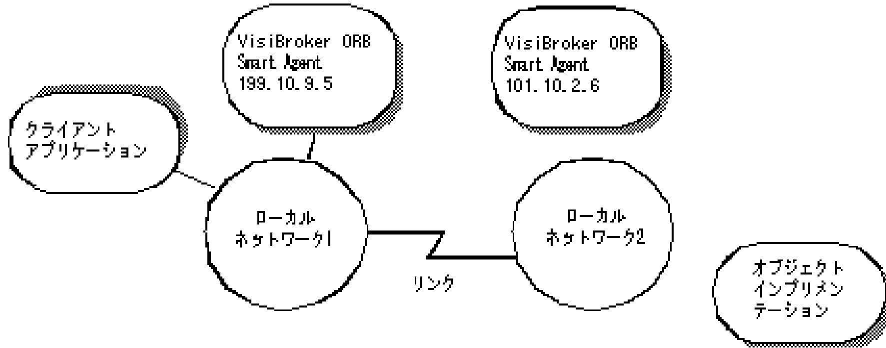
```
prompt> setenv OSAGENT_PORT 5678
prompt> osagent &
prompt> oad &
```

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。

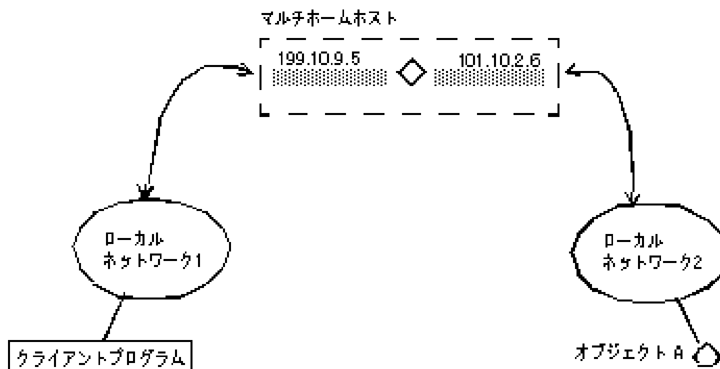
- エージェント 2 が起動すると，その存在を UDP ブロードキャストし，その他のスマートエージェントを検索するために要求メッセージを送信します。
- エージェント 1 は，エージェント 2 がネットワーク上で使用可能であることを認識し，要求メッセージに応答します。
- エージェント 2 は，ほかのエージェント（エージェント 1）がネットワーク上で使用可能であることを認識します。

  〔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。166 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166 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> vbj -Dvbroker.agent.addr=<ip_address> Server
```

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() メンバー関数が

呼び出されます。クライアントは，この関数を実行して，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，275 ページの第 20 章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。

オプション	説明
-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。

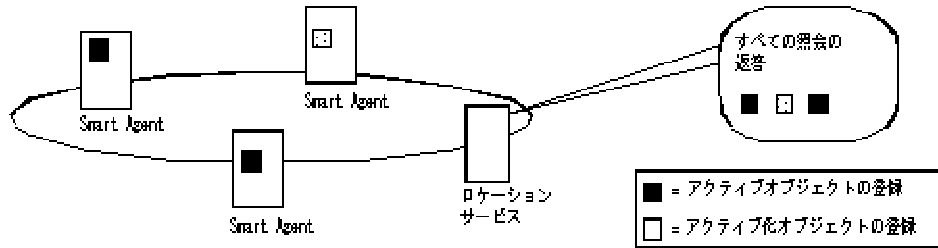
ロケーションサービスの概要

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，95 ページの「POA の使い方」，417 ページの「VisiBroker における BOA の使い方」，および 275 ページの「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ることにより，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。

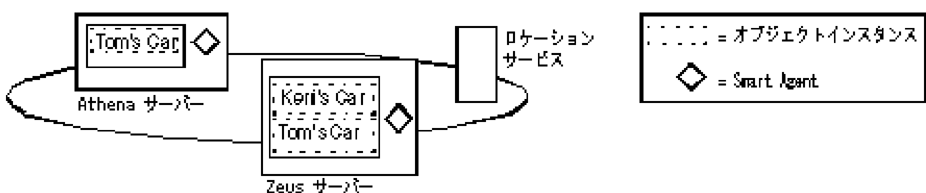
下の図は，次のサンプル IDL で指定されたインターフェースリポジトリ ID とインスタンス名の使用例を示します。

```
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

`all_agent_locations()` メソッドで `String[]` 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。それには，`all_repository_ids()` メソッドで `String[]` 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
`org.omg.CORBA.Object[] all_instances(String 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。
`Desc[] all_instance_descs(Stringrepository_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
`org.omg.CORBA.Object[] all_replica String repository_id, String 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。
`Desc[] all_replica_descs(String repository_id, String 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( com.inprise.vbroker.ObjLocation. TriggerDesc desc, com.inprise.vbroker. ObjLocation.TriggerHandler handler)`	トリガーハンドラを登録します。
`void unreg_trigger( com.inprise.vbroker.ObjLocation. TriggerDesc desc, com.inprise.vbroker.ObjLocation. TriggerHandler 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(com.inprise.vbroker. ObjLocation.TriggerDescdesc)`	`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(com.inprise.vbroker. ObjLocation.TriggerDescdesc)`	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

```
// AccountFinder.java
public class AccountFinder {
  public static void main(String[] args) {
  try {
    // ORB を初期化します。
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
    com.inprise.vbroker.ObjLocation.Agent the_agent = null;
    try {
      the_agent = com.inprise.vbroker.ObjLocation.AgentHelper.narrow(
        orb.resolve_initial_references("LocationService"));
    }
    catch (org.omg.CORBA.ORBPackage.InvalidName e) {
      System.out.println("Not able to resolve references " +
        "for LocationService");
      System.exit(1);
    }
    catch (Exception e) {
      System.out.println("Unable to locate LocationService!");
```

```
        System.out.println("Caught exception: " + e);
        System.exit(1);
      }
      org.omg.CORBA.Object[] accountRefs =
        the_agent.all_instances("IDL:Bank/AccountManager:1.0");
      System.out.println("Agent returned " + accountRefs.length +
        " object references");
      for (int i=0; i < accountRefs.length; i++) {
        System.out.println("Stringified IOR for account #" + (i+1) + ":");
        System.out.println(orb.object_to_string(accountRefs[i]));
        System.out.println();
      }
      } catch (Exception e) {
        System.out.println("Caught exception: " + e);
        System.exit(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

```
// Find.java
public class Find {
public static void main(String[] args) {
   try {
      // ORB を初期化します。
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
      com.inprise.vbroker.ObjLocation.Agent agent = null;
      try {
         agent = com.inprise.vbroker.ObjLocation.AgentHelper.narrow(
      orb.resolve_initial_references("LocationService"));
      } catch (org.omg.CORBA.ORBPackage.InvalidName e) {
         System.out.println("Not able to resolve references " + "for
               LocationService");
         System.exit(1);
      } catch (Exception e) {
         System.out.println("Not able to resolve references " + "for
               LocationService");
         System.out.println("Caught exception: " + e);
         System.exit(1);
      }
      boolean done=false;
      java.io.BufferedReader in =
            new java.io.BufferedReader(new
java.io.InputStreamReader(System.in));
      while (! done) {
         System.out.print("-> ");
         System.out.flush();
         String line = in.readLine();
         if(line.startsWith("agents")) {
            java.lang.String[] agentList = agent.all_agent_locations();
            System.out.println("Located " + agentList.length + " agents");
            for (int i=0; i < agentList.length; i++) {
                System.out.println("¥t" + "Agent #" + (i+1) + ":  " +
                        agentList[i]);
            }
```

```
        } else if(line.startsWith("rep")) {
           java.lang.String[] repIds = agent.all_repository_ids();
           System.out.println("Located " + repIds.length + " repository Ids");
           for (int i=0; i < repIds.length; i++) {
              System.out.println("¥t" + "Repository Id #" + (i+1) + ":  " +
                       repIds[i]);
           }
        } else if(line.startsWith("objects ")) {
           String names = line.substring("objects ".length(), line.length());
           PrintObjects(names,agent,orb);
        } else if(line.startsWith("quit")) {
           done = true;
        } else {
           System.out.println("Commands: agents¥n" +
              "          repository_ids¥n" +
              "          objects        <rep Id>¥n" +
              "          objects        <rep Id> <obj name>¥n" +
              "          quit¥n");
        }
     }
  } catch (com.inprise.vbroker.ObjLocation.Fail err) {
     System.out.println("Location call failed with reason " + err.reason);
  } catch (java.lang.Exception err) {
     System.out.println("Caught error " + err);
     err.printStackTrace();
  }
}
public static void PrintObjects(String names,
              com.inprise.vbroker.ObjLocation.Agent agent,
              org.omg.CORBA.ORB orb)
      throws com.inprise.vbroker.ObjLocation.Fail {
   int space_pos = names.indexOf(' ');
   String repository_id;
   String object_name;
   if (space_pos == -1) {
      repository_id = names;
      object_name = null;
   } else {
      repository_id = names.substring(0,names.indexOf(' '));
      object_name = names.substring(names.indexOf(' ')+1);
   }
   org.omg.CORBA.Object[] objects;
   com.inprise.vbroker.ObjLocation.Desc[] descriptors;
   if (object_name == null) {
      objects = agent.all_instances(repository_id);
      descriptors = agent.all_instances_descs(repository_id);
   } else {
      objects = agent.all_replica(repository_id,object_name);
      descriptors = agent.all_replica_descs(repository_id,object_name);
   }
   System.out.println("Returned " + objects.length + " objects");
   for (int i=0; i<objects.length; i++) {
      System.out.println("¥n¥nObject #" + (i+1) + ":");
      System.out.println("==================");
      System.out.println("¥tRep ID: " +
            ((com.inprise.vbroker.CORBA.Object)objects[i])._repository_id());
      System.out.println("¥tInstance:" +
            ((com.inprise.vbroker.CORBA.Object)objects[i])._object_name());
      System.out.println("¥tIOR: " + orb.object_to_string(objects[i]));
      System.out.println();
      System.out.println("Descriptor #" + (i+1));
      System.out.println("=====================================");
      System.out.println("Host:          " + descriptors[i].iiop_locator.host);
      System.out.println("Port:          " + descriptors[i].iiop_locator.port);
      System.out.println("Agent Host:    " + descriptors[i].agent_hostname);
      System.out.println("Repository Id: " + descriptors[i].repository_id);
```

```
            System.out.println("Instance:      " + descriptors[i].instance_name);
            System.out.println("Activable:     " + descriptors[i].activable);
        }
    }
}
```

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。

トリガーハンドラが登録されていない場合は，System.exit() を呼び出して，プログラムを終了します。

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.java
import java.io.*;
import org.omg.PortableServer.*;
class TriggerHandlerImpl extends
        com.inprise.vbroker.ObjLocation.TriggerHandlerPOA {
    public TriggerHandlerImpl(com.inprise.vbroker.ObjLocation.Agent agent,
        com.inprise.vbroker.ObjLocation.TriggerDesc initial_desc) {
            agent = agent;
            initial_desc = initial_desc;
    }
    public void impl_is_ready(com.inprise.vbroker.ObjLocation.Desc desc) {
        notification(desc, true);
    }
    public void impl_is_down(com.inprise.vbroker.ObjLocation.Desc desc) {
        notification(desc, false);
    }
    private void notification(com.inprise.vbroker.ObjLocation.Desc desc,
        boolean isReady) {
        if (isReady) {
            System.out.println("Implementation is ready:");
        } else {
            System.out.println("Implementation is down:");
        }
        System.out.println("¥tRepository Id = " + desc.repository_id + "¥n" +
              "¥tInstance Name = " + desc.instance_name + "¥n" +
              "¥tHost Name     = " + desc.iiop_locator.host + "¥n" +
              "¥tBOA Port      = " + desc.iiop_locator.port + "¥n" +
              "¥tActivable     = " + desc.activable + "¥n" + "¥n");
        System.out.println("Unregister this handler and exit (yes/no)?");
        try {
            BufferedReader in = new BufferedReader(
                  new InputStreamReader(System.in));
            String line = in.readLine();
            if(line.startsWith("y") || line.startsWith("Y")) {
                try {
                    agent.unreg_trigger(_initial_desc, _this());
                } catch (com.inprise.vbroker.ObjLocation.Fail e) {
                    System.out.println("Failed to unregister trigger with reason=["
+
                          e.reason + "]");
                }
```

```
            System.out.println("exiting...");
            System.exit(0);
         }
      } catch (java.io.IOException e) {
         System.out.println("Unexpected exception caught: " + e);
         System.exit(1);
      }
   }
   private com.inprise.vbroker.ObjLocation.Agent _agent;
   private com.inprise.vbroker.ObjLocation.TriggerDesc  _initial_desc;
}
public class AccountTrigger {
   public static void main(String args[]) {
      try {
         // ORB を初期化します。
         org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
         POA rootPoa =
               POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
         rootPoa.the_POAManager().activate();
         com.inprise.vbroker.ObjLocation.Agent the_agent =
                     com.inprise.vbroker.ObjLocation.AgentHelper.narrow(

orb.resolve_initial_references("LocationService"));
         // トリガー記述および適切な TriggerHandler を作成します。
         // OSAgent がインターフェースの実装を検出すると，
         // TriggerHandler が呼び出されます
            "Bank::AccountManager"
         com.inprise.vbroker.ObjLocation.TriggerDesc desc =
                      new com.inprise.vbroker.ObjLocation.TriggerDesc(
                                     "IDL:Bank/AccountManager:1.0", "", "");
         TriggerHandlerImpl trig = new TriggerHandlerImpl(the_agent, desc);
         rootPoa.activate_object(trig);
         the_agent.reg_trigger(desc, trig._this());
         orb.run();
   } catch (Exception e) {
         e.printStackTrace();
         System.exit(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。

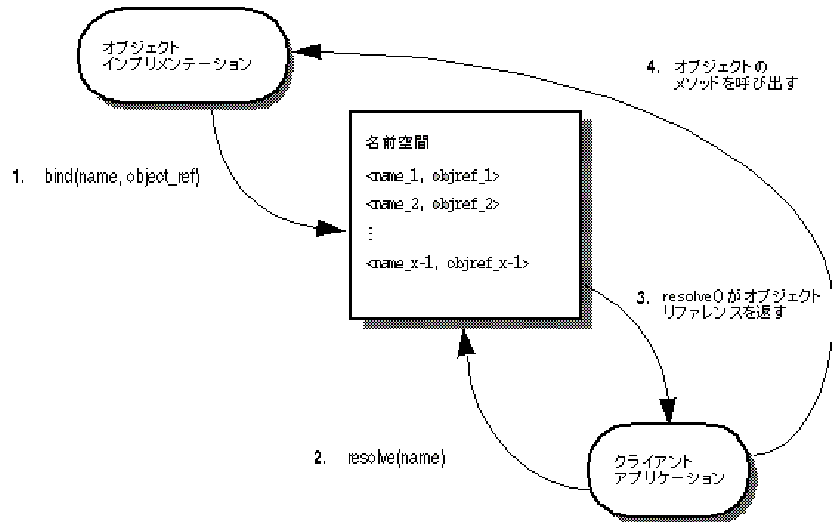
概要

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。

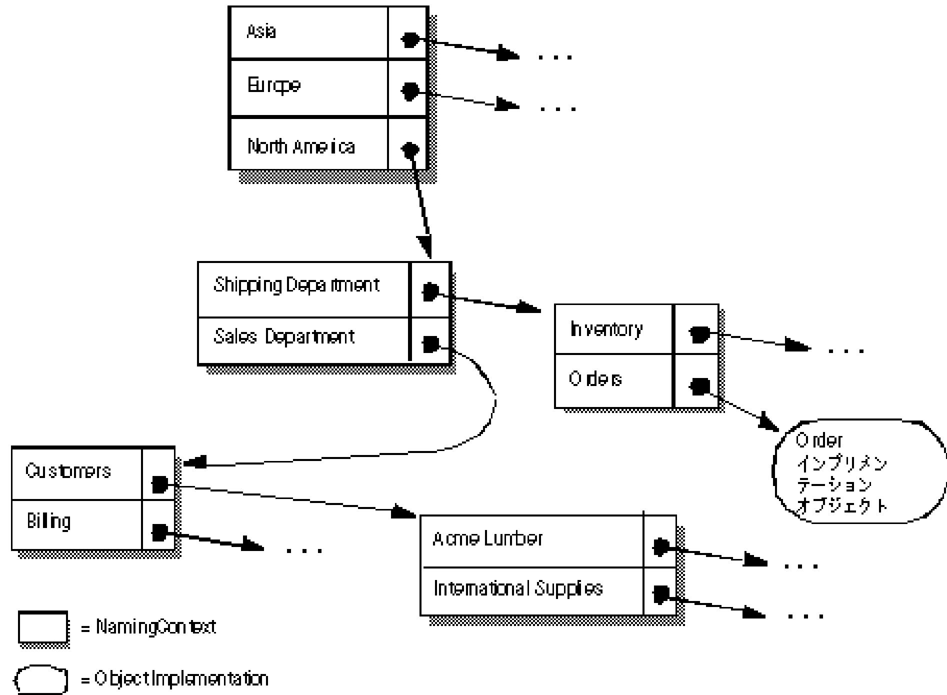
- スマートエージェントはフラットな名前空間を使用しますが，VisiNaming サービスでは階層構造を持つ名前空間を使用します。
- スマートエージェントを使用する場合，オブジェクトのインターフェース名は，クライアントとサーバーアプリケーションをコンパイルするときに定義します。したがって，インターフェース名を変更すると，アプリケーションも再コンパイルする必要があります。これに対して，VisiNaming サービスでは，実行時にオブジェクトインプリメンテーションでそのオブジェクトに論理名をバインドできます。
- スマートエージェントを使用する場合，オブジェクトは 1 つのインターフェース名しか実装できません。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。
NameComponent[] continentName = { new NameComponent("NorthAmerica", "") };
NamingContext continentContext =
         rootNamingContext.bind_new_context(continentName);
NameComponent[] departmentName = { new NameComponent("SalesDepartment", "") };
NamingContext departmentContext =
         continentContext.bind_new_context(departmentName);
NameComponent[] objectName = { new NameComponent("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，197 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，28 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>
```

vbj コマンドによる VisiNaming サービスの起動

VisiNaming サービスは vbj でも起動できます。

```
prompt>vbj com.inprise.vbroker.naming.ExtFactory <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。

```
. . .
   org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
   org.omg.CORBA.Object 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。

```
vbj -DSVCnameroot=ns1 <client_application>
```

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。

-DORBInitRef 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。

```
vbj -DORBInitRef=NameService=corbaloc::TestHost:20002/NameService
<client_application>
```

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。

```
vbj -DORBInitRef NameService=corbaname::TestHost:20003/ <client_application>
```

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。

-DORBDefaultInitRef

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。

-DORBDefaultInitRef 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。

```
vbj -DORBDefaultInitRef corbaloc::TestHost:20002 <client_application>
```

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。

次の例は，corbaloc を使用して複数の VisiNaming サービスを設定する方法です。

```
client -DORBDefaultInitRef
NameService=corbaloc::bart:20000,:Bart:20001,:Bart:20002/NameService
  -ORBpropStorage clt.props
```

-DORBDefaultInitRef 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。

URL が corbaname::TestHost:20002 の場合は，-DORBDefaultInitRef を指定すると，Java 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。

デフォルトネーミングコンテキストの取得

Java クライアントアプリケーションでは，ORB インターフェースの resolve_initial_references メソッドで VisiNaming サービスに接続できます。この機能を使用するには，クライアントの起動時に SVCnameroot または ORBInitRef パラメータを指定します。

たとえば，デフォルトネーミングコンテキストとしてネーミングコンテキスト Inventory を使用する ClientApplication という名前の Java アプリケーションを起動するには次のコマンドを入力します。

```
prompt> vbj -DSVCnameroot=NorthAmerica/ShippingDepartment/Inventory \
      ClientApplication
```

ここで，NorthAmerica はサーバー名であり，ShippingDepartment/Inventory はルートコンテキストの文字列化された名前です。

メモ　vbj コマンドを使用するには，すべての -D プロパティを Java クラス名の前に指定します。

ネーミングコンテキストファクトリの取得

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。

```
ExtendedNamingContextFactory myFactory =
ExtendedNamingContextFactoryHelper.narrow(
orb.resolve_initial_reference("VisiNamingContextFactory") );
```

osagent がネットワーク上で実行されている場合は，次のようにクライアントを開始する必要があります。

```
vbj -DSVCnameroot=<ns_name> Client
```

osagent がネットワーク上で実行されていない場合は，次のようにクライアントを開始する必要があります。

```
vbj -DORBInitRef=VisiNamingContextFactory=
     corbaloc::<host>:<port>/VisiNamingContextFactory Client
```

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，208 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 16.2　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.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
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，203 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。
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，208 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，202 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）

JDBC ドライバクラス名	説明
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）
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.com/jp/products/interbase/index.html 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，199 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 208 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 myFactory =
        ExtendedNamingContextFactoryHelper.bind(orb, "NamingService");
ClusterManager clusterMgr = myFactory.get_cluster_manager();
Cluster 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 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 203 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，197 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。

Java 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，システムプロパティ -DAnyServiceOrder=true を使用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，システムプロパティ -DSVCnameroot=<serverClusterName> 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できます。または，corbaloc メカニズムを使用できます。そのために，クラスタを構成するすべての VisiNaming サービスインスタンスに対してホストとポートのペアを指定します。これが resolve_initial_references で使用されます。

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，202 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;proxy
Host3:proxyPort3
```

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。

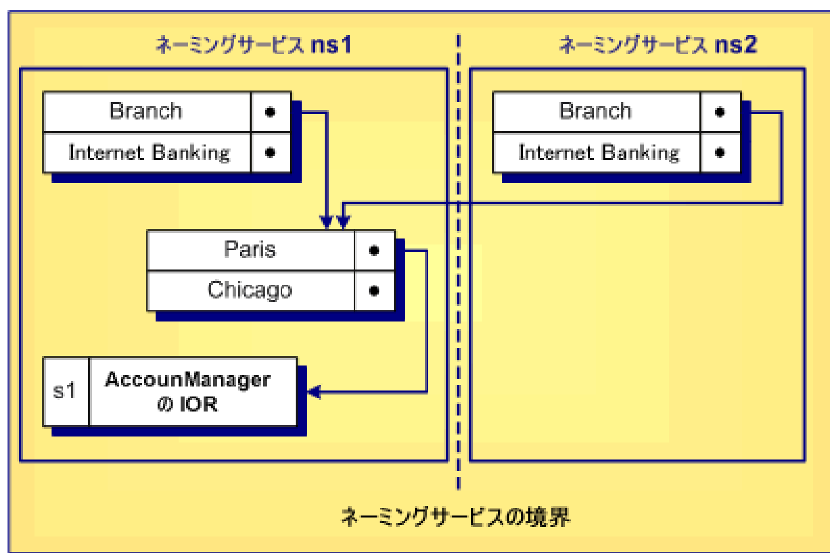
```
vbroker.se.iiop_tp.scm.ssl.listener.deferAccept=true
vbroker.se.iiop_tp.scm.ssl.listener.port=<port_number_for_ssl>
vbroker.se.iiop_tp.scm.ssl.listener.portRange=0
```

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
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，213 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
```

import ステートメント

VisiNaming サービスの VisiBroker ORB 拡張を使用するすべての Java クラスで，次の import ステートメントを使用する必要があります。

```
import com.inprise.vbroker.CosNamingExt.*;
. . .
```

VisiNaming サービスの OMG 準拠の機能にアクセスする場合は，次のパッケージが必要です。

```
import org.omg.CosNaming.*
Import org.omg.CosNaming.NamingContextPackage.*
Import org.omg.CosNaming.NamingContextExtPackage.*
```

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，187 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。

- 187 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 bind を発行しようとすると，CORBA.NO_IMPLEMENT 例外が生成されます。

名前のバインドの例

Bank Naming サンプルでは，AccountManager インターフェースで口座を開いたり，その口座の残高を照会します。次の Server クラスは，VisiNaming サービスを使用して名前をオブジェクトリファレンスにバインドする方法を示しています。サーバーが IOR をネーミングサーバーのルートコンテキストに公開すると，クライアントが IOR を取得します。

このサンプルでは，次のような操作を行う方法を習得できます。

1. VisiBroker ORB インスタンスの resolve_initial_references メソッドで，VisiNaming サービスのルートコンテキストへのリファレンスを取得します。このサンプルでは，デフォルト名の NameService で VisiNaming サービスを起動します。
2. NamingContextExtHelper クラスの narrow メソッドを使用して，ルートコンテキストのリファレンスをキャストします。
3. POA と AccountManagerImpl オブジェクトのサーバントを作成します。
4. 最後に，NamingContext インターフェースの bind メソッドを使用して，名前「BankManager」を AccountManagerImpl オブジェクトのオブジェクトリファレンスにバインドします。

POA の詳細は，第 9 章「POA の使い方」を参照してください。

```
Import org.omg.PortableServer.*;
Import org.omg.CosNaming.*;
public class Server {
   public static void main(String[] args) {
      try {
         // ORB を初期化します。
         org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);

         // ルート POA へのリファレンスを取得します。
         POA rootPOA =
            POAHelper.narrow(orb.resolve_initial_references("RootPOA"));

         // VisiNaming サービスのルートコンテキストへのリファレンスを取得します。
         org.omg.CORBA.Object rootObj =
            orb.resolve_initial_references("NameService");
```

```
         NamingContextExt root = NamingContextExtHelper.narrow(rootObj);

         // 永続的 POA のポリシーを作成します。
         org.omg.CORBA.Policy[] policies = {
            rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)
         };

         // 適切なポリシーで myPOA を作成します。
         POA myPOA = rootPOA.create_POA( "bank_agent_poa",
rootPOA.the_POAManager(),
            policies );

         // サーバントを作成します。
         AccountManagerImpl managerServant = new AccountManagerImpl();

         // サーバントの ID を決定します。
         byte[] managerId = "BankManager".getBytes();

         // その ID を使用して 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();

         // ルートコンテキストで BankManager を名前に関連付けます。
         // JDK 1.1.x のバグを回避するには，キャストが必要です。
         ((NamingContext)root).bind(root.to_name("BankManager"),
            myPOA.servant_to_reference(managerServant));
         System.out.println(myPOA.servant_to_reference(managerServant)
            + " is ready.");

         // 着信要求を待機します。
         orb.run();
      } catch (Exception e) {
         e.printStackTrace();
      }
   }
}
```

VisiNaming と JDataStore HA を組み合わせて使用する際の設定

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 （[Structure] 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 （[Structure] 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 （[Structure] ペインで）［Mirrors］を右クリックし，［Synchronize all mirrors］を選択します。
15 （［Structure］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。

```
vbj -DSVCnameroot=NameService ServerA
```

5 次のコマンドを使用して，ServerB を起動します。

```
vbj -DSVCnameroot=NameService ServerB
```

6 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=NameService Client NameService
```

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 「6 のクライアントを起動」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 「6 のクライアントを起動」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，「6 のクライアントを起動」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，「6 のクライアントを起動」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 「6 のクライアントを起動」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。

```
nameserv -VBJclasspath <JDS_Install>\lib\
jdsserver.jar -config autofailover.properties -VBJprop
vbroker.naming.url=jdbc:borland:dsremote://143.186.141.14/AutoFailover_Mirror5.jds
-VBJprop vbroker.se.iiop_tp.scm.iiop_tp.listener.port=12372 Master
```

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。

```
nameserv -VBJclasspath <JDS_Install>\lib\
jdsserver.jar -config autofailover.properties -VBJprop
vbroker.naming.url=jdbc:borland:dsremote://143.186.141.14/AutoFailover_Mirror6.jds
-VBJprop vbroker.se.iiop_tp.scm.iiop_tp.listener.port=12373 -VBJprop
vbroker.naming.jdsSvrPort=1116
 Slave
```

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

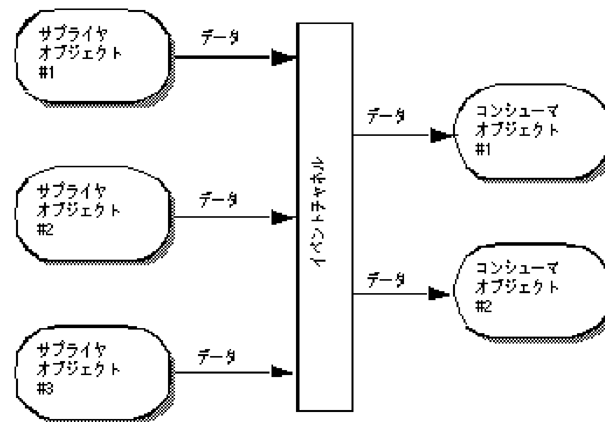
この節では，VisiBroker によるイベントサービスについて説明します。

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 ガイド』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

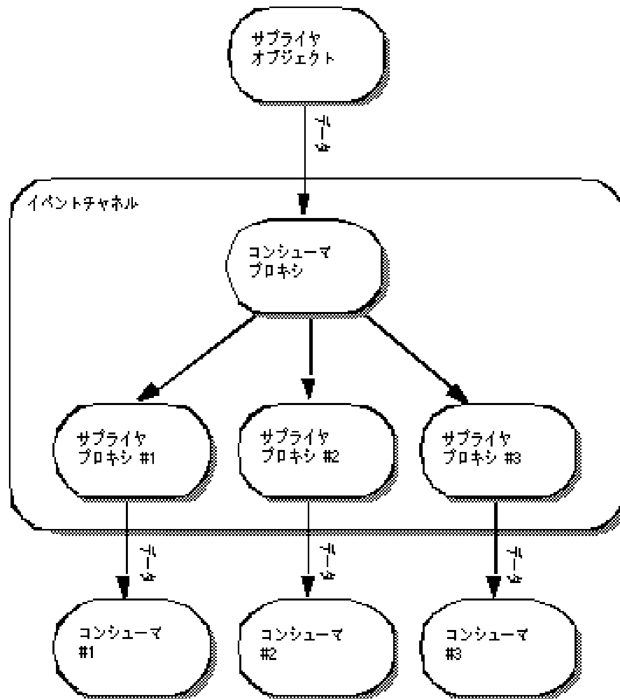
シューマオブジェクトによって処理されます。上の図に示されている 3 つのサプライヤがそれぞれ毎秒 1 つのメッセージを送信する場合，各コンシューマは毎秒 3 つのメッセージを受け取り，イベントチャネルは毎秒合計 6 つのメッセージを転送します。

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。詳細については，233 ページの「イベントサービスの起動」を参照してください。

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。

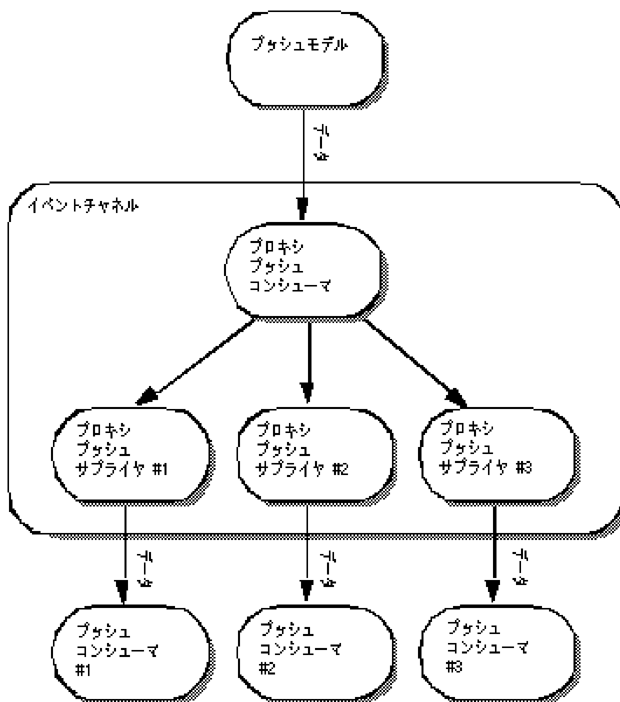
- VisiBroker によるイベントサービスは，イベントチャネルおよびコンシューマアプリケーションのいずれにも，データ配信の確認手段を提供しません。コンシューマ，サプライヤ，およびイベントチャネルの間の通信は，TCP/IP を使って実装され，これにより，チャネルおよびコンシューマの両方に信頼できるデータ配信が提供されます。ただし，これは，送信されたデータが実際にすべて受信者によって処理されたことを保証するものではありません。

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。詳細については，233 ページの「イベントサービスの起動」を参照してください。

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。

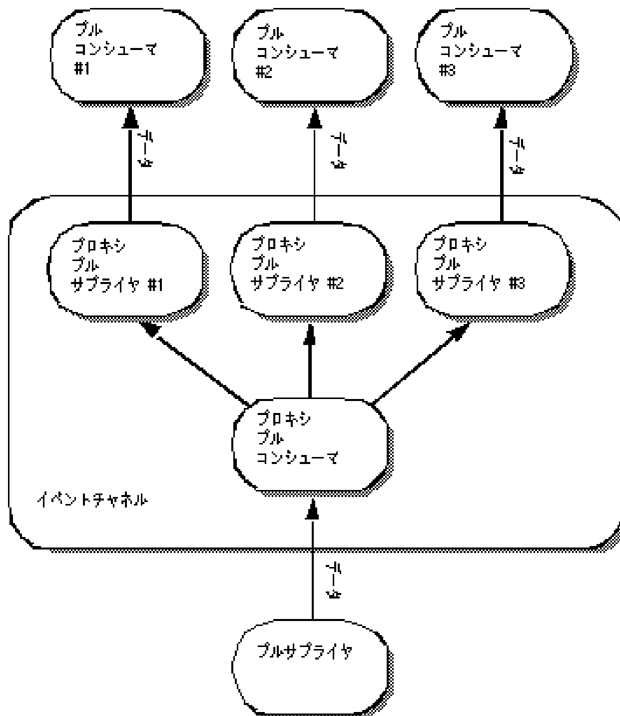
プルモデル

*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。

Java でイベントサービスを起動する方法の詳細については，234 ページの「インプロセスイベントチャネル」も参照してください。

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> &
```

ここで，<channelName> は，イベントチャネルのユーザー指定のオブジェクト名です。また，<iorFilename> は，そのオブジェクトの ior が書き込まれるファイルのユーザー指定のファイル名です。

EventChannel を作成する別の方法としては，次のように PushModelChannel を実行します。

```
prompt> vbj PushModelChannel <iorFilename>
```

PushModelChannel は，最初に EventChannel を作成し，その ior をユーザー指定のファイル <iorFilename> に公開します。これで，ほかのクライアント（PushModel など）が初期リファレンスを使用してその EventChannel にバインドできます。

それには，次のように指定します。

```
prompt> vbj -DORBInitRef=EventService=file:<fullpath + iorFilename>
PushModel
```

どちらの方法でイベントチャネルを作成した場合でも，<iorFilename> で指定した名前のファイルを指定のディレクトリに作成しておく必要があります。

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	EventChannelHelper.narrow(orb.resolve_initial_references("EventService"))	EventChannelHelper.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	`EventChannelHelper.narrow(orb.resolve_initial_references("EventService"))`	`EventChannelHelper.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。`PullSupply.java` ファイルと `PullConsume.java` ファイルは，サプライヤとコンシューマを実装します。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。

これらのサンプルを実行するには，サプライヤ／コンシューマのペアが必要です。プッシュまたはプルのコンシューマは，`Push` または `Pull` のサプライヤとペアにすることができます。サプライヤとコンシューマは任意の順序で起動できます。ただし，イベントチャネルは，同一のオブジェクトインスタンスである必要があります。

プッシュモデルサンプルを使用し始める前に，このサンプルを実行する必要があります。以降の節では，このサンプルの実行方法について説明します。

プッシュモデルサンプルの実行

PushModel サンプルを実行するには，次のように入力します。

```
prompt> vbj -DORBInitRef=EventService=file:<fullpath of iorFilename> PushModel
```

e を選択してイベントチャネルにバインドし，p を選択してイベントチャネルからプッシュコンシューマへのプロキシを取得し，m を選択して PushModel をインスタンス化し，c を選択してイベントチャネルに接続します。

EventChannel にプッシュされるメッセージの内容を示す文が次々に表示されます。画面に何が表示されていても，続けて操作を選択できます。s オプションを使用すると，イベント間の秒数を指定できます。最後に，d を選択して切断し，q を選択して終了します。

PushView を実行するには，次のように入力します。

```
prompt>vbj -DORBInitRef=EventService=file:
<fullpath of iorFilename> PushView
```

e を選択してイベントチャネルにバインドし，p を選択してイベントチャネルからプッシュコンシューマへのプロキシを取得し，v を選択して PushView をインスタンス化し，c を選択してイベントチャネルへ接続し，d を選択して切断し，q を選択して終了します。このサンプルを実行するには，PushView がデータを受信するために，Push または Pull のサプライヤが別のコンソールで実行されており，同じイベントチャネルにデータを継続して送信している必要があります。サプライヤとコンシューマは任意の順序で起動できます。

PullModel サンプルの実行

PullModel サンプルを実行するには，次のように入力します。

```
prompt> vbj -DORBInitRef=EventService=file:
<fullpath of iorFilename> PullModel
```

e を選択してイベントチャネルにバインドし，p を選択してイベントチャネルからプッシュコンシューマへのプロキシを取得し，m を選択して PullModel をインスタンス化し，c を選択してイベントチャネルへ接続し，d を選択して切断し，q を選択して終了します。

PullView サンプルの実行

PullView を実行するには，次のように入力します。

```
prompt>vbj -DORBInitRef=EventService=file:
<fullpath of iorFilename> PullView
```

e を選択してイベントチャネルにバインドし，p を選択してイベントチャネルからプッシュサプライヤへのプロキシを取得し，v を選択して PushView をインスタンス化し，c を選択してイベントチャネルに接続します。次に，a を選択して非同期にプルするか，または s を同期にプルします。終了するには，d を選択して切断し，q を選択して終了します。

このサンプルを実行するには，PullView がデータを受信するために，Push または Pull のサプライヤが別のコンソールで実行されており，同じイベントチャネルにデータを継続して送信している必要があります。サプライヤとコンシューマは任意の順序で起動できます。

PullSupply

PullSupply クラスは，PullSupplierPOA クラスから派生し，main，pull，および try_pull メソッドのインプリメンテーションを提供します。次に示す pull メソッドは，番号付きの「hello」メッセージを返します。try_pull メソッドは，常に hasEvent フラグを true に設定し，pull メソッドを呼び出してメッセージを提供します。PullSupply オブジェクトが EventChannel に接続されると，これらのメソッドは，サプライヤからデータを引き出すために，チャネルによって使用されます。

次に示す main メソッドは，通常の VisiBroker ORB と POA を作成し，指定された EventChannel に接続し，EventChannel から ProxyPullConsumer を取得し，PullSupply オブジェクトをインスタンス化して，POA 上で PullSupply オブジェクトを起動し，次にこのプルサプライヤをプロキシプルコンシューマに接続します。

PullSupply の実行

まず，PullSupply.java コンパイルし，イベントサービスを起動します。234 ページの「インプロセスイベントチャネル」を参照してください。次に，下のコマンドを使ってサプライヤを実行します。

```
vbj -DORBInitRef = <channel_name> = file:<fullpath of iOrFilename> PullSupply
```

pull メソッドと try_pull メソッドのインプリメンテーション

```
// PullSupply.java
import org.omg.CosEventComm.*;
import org.omg.CosEventChannelAdmin.*;
import org.omg.PortableServer.*;
public class PullSupply extends PullSupplierPOA {
   private POA _myPOA;
   private PullConsumer _pullConsumer;
   private int _counter;
   PullSupply(PullConsumer pullConsumer, POA myPOA) {
      _pullConsumer = pullConsumer;
      _myPOA = myPOA;
   }
   public void disconnect_pull_supplier() {
      System.out.println("Model::disconnect_pull_supplier()");
      try {
         _myPOA.deactivate_object("PullSupply".getBytes());
      } catch(Exception e) {
         e.printStackTrace();
      }
   }
   public org.omg.CORBA.Any pull() throws Disconnected {
      if(_pullConsumer == null) {
         throw new Disconnected();
      }
      try {
         Thread.currentThread().sleep(1000);
      } catch(Exception e) {
      }
      //org.omg.CORBA.Any message =
            new org.omg.CORBA.Any().from_string("Hello #" + ++_counter);
      org.omg.CORBA.Any message = _orb().create_any();
      message.insert_string("Hello #" + ++_counter);
      System.out.println("Supplier being pulled: " + message);
      return message;
   }
   public org.omg.CORBA.Any try_pull(org.omg.CORBA.BooleanHolder hasEvent)
throws
         org.omg.CORBA.SystemException, Disconnected {
      hasEvent.value = true;
      return pull();
   }
. . .
```

PullSupply の main メソッド

```
// PullSupply.java
import org.omg.CosEventComm.*;
import org.omg.CosEventChannelAdmin.*;
import org.omg.PortableServer.*;
public class PullSupply extends PullSupplierPOA {
   . . .
   public static void main(String[] args) {
      try {
         org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
         // ルート POA へのリファレンスを取得します。
         POA rootPOA =
      POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
         // 永続的 POA のポリシーを作成します。
         org.omg.CORBA.Policy[] policies = {
               rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)
         };
         // 適切なポリシーで myPOA を作成します。
         POA myPOA = rootPOA.create_POA("event_service_poa",
               rootPOA.the_POAManager(), policies);
         EventChannel channel = null;
         PullSupply model = null;   ProxyPullConsumer pullConsumer = null;
         channel =

EventChannelHelper.narrow(orb.resolve_initial_references("EventService"));
         System.out.println("Located event channel: " + channel);
         pullConsumer = channel.for_suppliers().obtain_pull_consumer();
         System.out.println("Obtained pull consumer: " + pullConsumer);
         model = new PullSupply(pullConsumer, myPOA);
         myPOA.activate_object_with_id("PullSupply".getBytes(), model);
         myPOA.the_POAManager().activate();
         System.out.println("Created model: " + model);
         System.out.println("Connecting ...");
         pullConsumer.connect_pull_supplier(model._this());
      } catch(Exception e) {
         e.printStackTrace();
      }
   }
}
```

PullConsume

PullConsume クラスは PullConsumerPOA class から派生し，PullSupply クラスからデータをプルするコマンドラインインターフェースを提供します。上に示したサンプルコードは，アプリケーションが使用可能な任意の EventChannel への接続，ProxyPullSupplier の取得，チャネルへの接続，およびコマンドプロンプトの表示を行う方法を示します。次の表に入力できるコマンドをまとめます。

表 17.3　PullConsume コマンド

コマンド	説明
a	try_pull メソッドを使用して，イベントチャネルから非同期にデータを引き出します。現在使用できるデータがない場合，このコマンドは「no data」メッセージを返します。
s	pull メソッドを使用して，イベントチャネルから同期的にデータを引き出します。現在使用できるデータがない場合，このコマンドはデータが利用できるまでブロックします。
q	チャネルから接続を解除し，ツールを終了します。

PullConsume の実行

まず，PullConsume.java コンパイルし，イベントサービスを起動します。234 ページの「インプロセスイベントチャネル」を参照してください。次に，下のコマンドを使ってコンシューマを実行します。

```
vbj -DORBInitRef = <channel_name> = file:<fullpath of iOr_filename> PullConsume
```

```
// PullConsume.java
import org.omg.CosEventComm.*;
import org.omg.CosEventChannelAdmin.*;
import org.omg.PortableServer.*;
import java.io.*;
public class PullConsume extends PullConsumerPOA {
   public void disconnect_pull_consumer() {
      System.out.println("View.disconnect_pull_consumer");
   }
   public static void main(String[] args) {
      try {
         org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
         // ルート POA へのリファレンスを取得します。
         POA rootPOA =
         POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
         // 永続的 POA のポリシーを作成します。
         org.omg.CORBA.Policy[] policies = {
               rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)
         };
         // 適切なポリシーで myPOA を作成します。
         POA myPOA = rootPOA.create_POA("event_service_poa",
               rootPOA.the_POAManager(), policies );
         EventChannel channel = null;
         PullConsume view = null;
         ProxyPullSupplier pullSupplier = null;
         BufferedReader in = new BufferedReader(new
InputStreamReader(System.in));
         channel =

EventChannelHelper.narrow(orb.resolve_initial_references("EventService"));
         System.out.println("Located event channel: " + channel);
         view = new PullConsume();
         myPOA.activate_object_with_id("PullConsume".getBytes(), view);
         myPOA.the_POAManager().activate();
         System.out.println("Created view: " + view);
         pullSupplier = channel.for_consumers().obtain_pull_supplier();
         System.out.println("Obtained pull supplier: " + pullSupplier);
         System.out.println("Connecting...");
         System.out.flush();
         pullSupplier.connect_pull_consumer(view._this());
         while(true) {
            System.out.print("-> ");
            System.out.flush();
            if (System.getProperty("VM_THREAD_BUG") != null) {
               while(!in.ready()) {
                  try {
                     Thread.currentThread().sleep(100);
                  } catch(InterruptedException e) {
                  }
               }
            }
            String line = in.readLine();
            if(line.startsWith("a")) {
               org.omg.CORBA.BooleanHolder hasEvent = new
org.omg.CORBA.BooleanHolder();
               org.omg.CORBA.Any result = pullSupplier.try_pull(hasEvent);
               System.out.println("try_pull: " +
                     (hasEvent.value ? result.toString() : "NO DATA"));
               continue;
```

```
            } else if(line.startsWith("s")) {
               org.omg.CORBA.Any result = pullSupplier.pull();
               System.out.println("pull: " + result);
               continue;
            } else if(line.startsWith("q")) {
               System.out.println("Disconnecting...");
               pullSupplier.disconnect_pull_supplier();
               System.out.println("Quitting...");
               break;
            }
            System.out.println("Commands: a  [a]synchronous pull¥n" +
                  "          s  [s]ynchronous pull¥n" +
                  "          q  [q]uit¥n");
         }
      } catch(Exception e) {
         e.printStackTrace();
      }
   }
}
```

イベントサービスの起動

VisiBroker for Java を使用している場合は，次のコマンドを使ってイベントサービスを起動できます。

```
vbj [-Dvbroker.events.debug] [-Dvbroker.events.interactive] [-
Dvbroker.events.max_queue_length=<number>] [-Dvbroker.events.debug.factory] ¥
[-Dvbroker.events.vm_thread_bug] com.inprise.vbroker.CosEvent.EventServer -ior
<ior filename> <channel name>
```

オプション	説明
-Dvbroker.events.debug	デバッグメッセージを標準出力に出力するためのオプションパラメータ。
-Dvbroker.events.interactive	イベントチャネルをコンソールから対話モードで実行するように指定します。
-Dvbroker.events.maxQueueLength	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。デフォルトの最大のキューの長さは，コンシューマごとに 100 メッセージです。
-Dvbroker.events.factory	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するように指定します。
channel_name	チャネルまたはチャネルファクトリの名前。

メモ Solaris などのいくつかの Java 仮想マシンのインプリメンテーションにはバグが見つかっており，このコマンドが停止する可能性があります。問題が発生した場合は，イベントサービス起動時に，-Dvbroker.events.vm_thread_bug パラメータを指定してください。

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。

```
vbj -Dvbroker.events.maxQueueLength=15 CosEvent.EventServer -ior myChannel.ior
        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。

インプロセスイベントチャネル

EventChannel を別のスタンドアローンサーバーとして実行するほかに，Event Service を使用すると，サーバーやクライアントアプリケーション内で EventChannel を作成することもできます。こうすると，サプライヤまたはコンシューマアプリケーションに EventChannel を提供するために別のプロセスを開始する必要がなくなります。

Java アプリケーションには，EventLibrary クラスがあり，EventChannel を作成するためのメソッドが提供されます。次にこのクラスが，必要なクラスをロードします。サプライヤ／コンシューマアプリケーションでインプロセスの EventChannel オブジェクトを作成するには，次の呼び出しを行います。

```
EventLibrary.create_Channel("MyChannel",whetherToDebug,maxQueueLength);
```

たとえば，デバッグをオフにし，キューの最大長を 100 にして，MyChannel という名前のチャネルを作成するには，次のように指定します。

```
EventLibrary.create_Channel("MyChannel",false,100);
```

この呼び出しが完了すると，クライアントアプリケーションは，ほかの CORBA オブジェクトにバインドする場合と同様に，この EventChannel にバインドできます。

たとえば，インプロセスでチャネルを作成するサプライヤアプリケーションがあり，コンシューマアプリケーションでその同じチャネルに接続するとします。その場合は，サプライヤアプリケーションからコンシューマアプリケーションにチャネルオブジェクトを渡す必要があります。オブジェクトを渡すためには，EventChannel オブジェクトを ior 文字列に変換し，その文字列をファイルに書き込みます。

```
try {
EventChannel channel = EventLibrary.create_Channel("MyChannel",false,100);
   PrintWriter pw = new PrintWriter(new FileWriter(ior_filename));
   pw.println(orb.object_to_string(channel));
   pw.close();
}
catch(IOException e) {
   System.out.println("Error writing the IOR to file " ior_filename);
}
```

ior_filename により，チャネルの ior 文字列を書き込むファイル名が指定されます。

PushModelChannel を実行するには，次のように指定します。

```
vbj PushModelChannel <ior_filenamegt;
```

PushModelChannel はプッシュサプライヤです。PushModelChannel で作成されたイベントチャネルには，プッシュコンシューマとプルコンシューマのどちらを接続することもできます。

```
vbj -DORBInitRef=EventService=file:<fullpath of ior_filename> PushView
```

ここで，<fullpath of ior_filename> は，PushModelChannel に渡された ior_filename のフルパスです。EventService は，<ior_filename> 内に保持されている ior にバインドされる名前（または識別子）です。PushView 内からは，次のようにイベントチャネルに接続できます。

```
EventChannel channel =
   EventChannelHelper.narrow(orb.resolve_initial_references("EventService"));
```

インプロセスイベントチャネルの使い方

アプリケーションでインプロセスイベントチャネル機能を使用する場合は，次の import ステートメントを追加する必要があります。

```
import com.inprise.vbroker.CosEvent.*;
```

Java EventLibrary クラス

EventLibrary クラスには，アプリケーションのプロセス内で EventChannel を作成するためのメソッドがいくつかあります。

Java サンプル

PushModelChannel.java ファイルは，インプロセスイベントチャネルを使用するプッシュサプライヤを実装します。このアプリケーションはコマンドプロンプトを表示し，そこで次のコマンドの 1 つを入力できます。

コマンド	説明
e	イベントチャネルを作成します。
s <number_of_seconds>	イベントチャネルの遅延を指定された秒数に設定します。0 以上の値を指定する必要があります。
p	プッシュコンシューマプロキシオブジェクトを取得します。
m	PushModelChannel を作成し，それを POA 上でアクティブ化します。
c	プッシュサプライヤを接続します。
d	プッシュコンシューマの接続を解除します。
q	アプリケーションを終了します。

下のサンプルコードは，ChannelLib.create_channel メソッドの使用方法を示す PushModelChannel.java の抜粋です。

```
public static void main(String[] args) {
      . . .
      channel = EventLibrary.create_channel("channel_server", false, 100);
      . . .
```

import ステートメント

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 import ステートメントを使用する必要があります。

```
import org.omg.CosEventComm.*;
import org.omg.CosEventChannelAdmin.*;
. . .
```

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。

   ```
   import com.inprise.vbroker.ServerManager.*;

   // サーバーマネージャ BigBadBoss 最上位コンテナへのリファレンスを返します。
   Container topContainer = ContainerHelper.bind(orb,"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。

   ```
   import com.inprise.vbroker.ServerManager.*;

   // "manager" には，AccountManager オブジェクトへのリファレンスが含まれています
   // AccountManager は com.inprise.vbroker.CORBA.Object
   // なので，ナローイングは不要です。ただし，返されたサーバー
   // マネージャリファレンスをコンテナに変換するには，
   // ナローイングが必要です
   Container topContainer = ContainerHelper.narrow(
   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，239 ページの「Container インターフェース」を参照してください。244 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，241 ページの「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 クラス

```
public interface Container extends
 com.inprise.vbroker.CORBA.Object
 com.inprise.vbroker.ServerManager.ContainerOperations
 org.omg.CORBA.portable.IDLEntity
```

このクラスをコードで使用する場合は，次のステートメントをコードに入れる必要があります。

```
import com.borland.vbroker.ServerManager.*;
import com.borland.vbroker.ServerManager.ContainerPackage.*;
```

Container のメソッド（Java）

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。

ここでは，Container インターフェースで実行できる Java 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

```
public String[] 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。

```
public com.inprise.vbroker.ServerManager.ContainerPackage.Property[]
      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。

```
public com.inprise.vbroker.ServerManager.ContainerPackage.Property
      get_property(String name) throws
com.inprise.vbroker.ServerManager.ContainerPackage.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。

```
public void set_property(String name, org.omg.CORBA.Any
      value) throws com.inprise.vbroker.ServerManager.
ContainerPackage.NameInvalid,
      com.inprise.vbroker.ServerManager. ContainerPackage.ValueInvalid,
      com.inprise.vbroker.ServerManager. ContainerPackage.ValueNotSettable
```

プロパティ **name** の値を要求された **value** に設定します。

```
public void persist_properties(boolean recurse) throws
      com.inprise.vbroker.ServerManager.StorageException;
```

コンテナは，関連付けられている 241 ページの「Storage インターフェースのクラスとメソッド」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。

```
public void restore_properties(boolean recurse) throws
      com.inprise.vbroker.ServerManager.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

```
public String[] 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。

```
public com.inprise.vbroker.ServerManager.ContainerPackage.Operation[]
      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。

```
public com.inprise.vbroker.ServerManager.ContainerPackage.Operation
      get_operation(String name) throws
com.inprise.vbroker.ServerManager.ContainerPackage.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。

```
public org.omg.CORBA.Any do_operation(
      com.inprise.vbroker.ServerManager.ContainerPackage.Operation op) throws
      com.inprise.vbroker.ServerManager.ContainerPackage.NameInvalid,
      com.inprise.vbroker.ServerManager.ContainerPackage.ValueInvalid,
      com.inprise.vbroker.ServerManager.ContainerPackage.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

```
public String[] 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。

```
public com.inprise.vbroker.ServerManager.ContainerPackage.NamedContainer[]
      get_all_containers();
```

すべての子コンテナを返します。

```
public com.inprise.vbroker.ServerManager.ContainerPackage.NamedContainer
      get_container(String name) throws
      com.inprise.vbroker.ServerManager.ContainerPackage.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。

```
public void add_container(
      com.inprise.vbroker.ServerManager.ContainerPackage.NamedContainer
container)
      throws
com.inprise.vbroker.ServerManager.ContainerPackage.NameAlreadyPresent,
      com.inprise.vbroker.ServerManager.ContainerPackage.ValueInvalid;
```

このコンテナの子コンテナとして，*container* を追加します。

```
public void set_container(String name,
com.inprise.vbroker.ServerManager.Container value) throws
      com.inprise.vbroker.ServerManager.ContainerPackage.NameInvalid,
      com.inprise.vbroker.ServerManager.ContainerPackage.ValueInvalid,
      com.inprise.vbroker.ServerManager.ContainerPackage.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 com.inprise.vbroker.ServerManager.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。

```
com.inprise.vbroker.ServerManager.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クラスとメソッド

Storage クラス

```
public interface Storage extends
 com.inprise.vbroker.CORBA.Object
 com.inprise.vbroker.ServerManager.StorageOperations,
 org.omg.CORBA.portable.IDLEntity
```

Storage インターフェースを使用する場合は，次のステートメントをコードに入れる必要があります。

```
import com.borland.vbroker.ServerManager.*;
import com.borland.vbroker.ServerManager.ContainerPackage.*;
```

Storage インターフェースのメソッド

```
public void open() throws
      com.inprise.vbroker.ServerManager.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。

```
public void close() throws
      com.inprise.vbroker.ServerManager.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。

```
public com.inprise.vbroker.ServerManager.ContainerPackage.Property[]
      read_properties() throws
      com.inprise.vbroker.ServerManager.StorageException;
```

ストレージからすべてのプロパティを読み取ります。

```
public com.inprise.vbroker.ServerManager.ContainerPackage.Property
read_property(String
        propertyName)
throwscom.inprise.vbroker.ServerManager.ContainerPackage.NameInvalid,
        com.inprise.vbroker.ServerManager.StorageException;
```

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。

```
public void
write_properties(com.inprise.vbroker.ServerManager.ContainerPackage.
      Property[] props) throws
com.inprise.vbroker.ServerManager.StorageException;
```

ストレージにプロパティシーケンスを保存します。

```
public void
write_property(com.inprise.vbroker.ServerManager.ContainerPackage.Property
      prop) throws com.inprise.vbroker.ServerManager.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（238 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）。

```
Container topContainer = ContainerHelper.bind(orb,"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。

```
public void displayContainer(NamedContainer cont, boolean top) {

    // すべてのコンテナを取得します。
    NamedContainer[] nc = cont.value.get_all_containers();

    // 現在のコンテナのすべてのプロパティを取得します。
    Property[] props=cont.value.get_all_properties();

    // 現在のコンテナのすべてのオペレーションを取得します。
    Operation[] ops=cont.value.get_all_operations();

    ....
    // ここで，すべてのプロパティとオペレーションを出力し，
    // すべてのコンテナを再帰的に処理します。

 }
```

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。

```
public void getSetProperties(NamedContainer topCont) throws Exception {

      // 最上位コンテナから ORB コンテナを取得します。
      Container orbCont=topCont.value.get_container("ORB").value;

      // iiop_tp SCM コンテナを取得します。
      // このコンテナは次のように包含されています
      // topCont->ORB->ServerEngines->iiop_tp->iiop_tp
      // （最初の iiop_tp は ServerEngine の名前）

      Container iiopCont=orbCont.get_container(
                        "ServerEngines").value.get_container(
                        "iiop_tp").value.get_container(
                        "iiop_tp").value;

      // bank_agent_poa コンテナを取得します。
      // このコンテナは次のように包含されています
      // topCont->ORB->RootPOA->Children->bank_agent_poa
      Container poaCont=orbCont.get_container(
                        "RootPOA").value.get_container(
                        "Children").value.get_container(
                        "bank_agent_poa").value;

      // ORB コンテナからプロセス ID プロパティを取得します。
```

```
        Property procIdProp=orbCont.get_property("vbroker.orb.procId");

        // iiop_tp コンテナからリスナーポートプロパティを取得します。
        Property portProp=iiopCont.get_property(
                        "vbroker.se.iiop_tp.scm.iiop_tp.listener.port");

        // bank_agent_poa コンテナから upTime プロパティを取得します。
        Property upTimeProp=poaCont.get_property("upTime");

        ....

        // ユーザーがリスナーポート値を変更します。
   org.omg.CORBA.Any portAny=orb.create_any();
      portAny.insert_long(newPort);
      iiopCont.set_property(
  "vbroker.se.iiop_tp.scm.iiop_tp.listener.port",portAny);

    ....

   // 更新されたプロパティをファイルに保存します。

  iiopCont.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。

```
public void invokeShutdown(NamedContainer topCont) throws Exception {

    Container orbCont=topCont.value.get_container("ORB").value;

    System.out.println("Executing ShutDown ...");

    // パラメータ（boolean wait_for_completion）を準備します。
    org.omg.CORBA.Any any=orb.create_any();
    any.insert_boolean(false);
    Parameter[] params=new Parameter[1];

    // 結果（void）を準備します。
    params[0]=new Parameter("waitForCompletion",any);
    org.omg.CORBA.TypeCode result=orb.get_primitive_tc(
                    org.omg.CORBA.TCKind.tk_void);

    // オペレーションを準備します。
    Operation op=new Operation("shutdown",params, result);

    // オペレーションを呼び出します。
    orbCont.do_operation(op);
}
```

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/

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 つの呼び出しフェーズ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フェーズで，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。

```
// Java でのリクエストエージェントリファレンス取得
org.omg.CORBA.Object ref
    = orb.resolve_initial_references("RequestAgent");
NativeMessaging.RequestAgentEx agent
    = NativeMessaging.RequestAgentExHelper.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。

```
// Java での Current オブジェクトリファレンス取得
org.omg.CORBA.Object ref
    = orb.resolve_initial_references("NativeMessagingCurrent");
NativeMessaging.Current current
    = NativeMessaging.CurrentHelper.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 つの独立したフェーズで実行できます。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
boolean stock_added = stock_manager.add_stock("ACME", 100.5);
omg.org.CORBA.StringHolder symbol_holder
    = new omg.org.CORBA.StringHolder("ACMA");
boolean closest_found
    = stock_manager.find_closest_symbol(symbol_holder);
```

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 253 ページの「ポーリング／プルモデル」と 254 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/PollingClient.java

// 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。
RequestProxy proxy = agent.create_request_proxy(
        stock_manager, "", null,
        new NameValuePair[0]);

// 2. プロキシの要求受信者を取得します。
StockManager stock_manager_rcv
    = StockManagerHelper.narrow(proxy.the_receiver());

// 3.  2 つの要求を受信者に送信し，
//    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[] pollers = new StockManager[2];
```

```
stock_manager_rcv.add_stock("ACME", 100.5);
pollers[0] = StockManagerHelper.narrow(current.the_poller());
StringHolder symbol_holder = new StringHolder("ACMA");
stock_manager_rcv.find_closest_symbol(symbol_holder);
pollers[1] = StockManagerHelper.narrow(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);

boolean stock_added;
do { stock_added = pollers[0].add_stock("", 0.0); }
while(current.reply_not_available());

boolean closest_found;
do { closest_found = pollers[1].find_closest_symbol(symbol_holder); }
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／プルフェーズ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 つのフェーズ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フェーズは，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／プルフェーズ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ングの API の構文およびセマンティクスの仕様については，257 ページの「高度な項目」と 263 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 呼び出しフェーズを実行します。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株式管理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/CallbackClient.java

// 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。
RequestProxy proxy = agent.create_request_proxy(
        stock_manager, "", reply_recipient,
        new NameValuePair[0]);

// 2. プロキシの要求受信者を取得します。
StockManager stock_manager_rcv
    = StockManagerHelper.narrow(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);
StringHolder symbol_holder = new StringHolder("ACMA");
Stock_manager_rcv.find_closest_symbol(symbol_holder);
```

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.java

void reply_available(
    omg.org.CORBA.Object reply_holder,
    String operation,
    byte[] cookie)
{
    StockManager poller
        = StockManagerHelper.narrow(reply_poller);

    // 反射コールバックを使って応答を取得します。
    if( operation.equals( "add_stock") ) {
        // add_stock() の戻り値を取得します。
        boolean stock_added = poller.add_stock("", 0.0);
        ...
    }
    else
    if( operation.equals("find_closest_symbol") ) {
            StringHolder symbol_holder = new StringHolder("");
            // find_closest_symbol() の戻り値を取得します。
            boolean closest_found
                = poller.find_closest_symbol(symbol_holder);
            ...
    }
}
```

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 つのフェーズ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，257 ページの「高度な項目」と 263 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/GroupPollingClient.java
StockManager pollers[] = new StockManager[2];
// タグ付き要求を 1 つ送信します。
current.request_tag("0".getBytes());
stock_manager_rcv.add_stock("ACME", 100.5);
pollers[0] = StockManagerHelper.narrow(current.the_poller());

// タグ付き要求をもう 1 つ送信します。
current.request_tag("1".getBytes());
StringHolder symbol_holder = new StringHolder("ACMA");
Stock_manager_rcv.find_closest_symbol(symbol_holder);
pollers[1] = StockManagerHelper.narrow(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。
byte[][] tags = null;
while(true) {
    // 可用性をポーリングします。
    try {
        tags = proxy.poll(max_timeout, true);
    }
```

```
        catch(PollingGroupIsEmpty e) {
            proxy.destroy(true);
            break;
        }

        // 応答を取得します。
        for(int i=0;i<tags.length;i++) {
            int id = Integer.parseInt(new String(tags[i]));

            switch(id) {
                case 0: // 最初に送信されたタグ付き要求
                    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
                    boolean closest_found;
                    closest_found
                        = pollers[1].find_closest_symbol(symbol_holder);
                    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("add stock".getBytes());
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("find symbol".getBytes());
StringHolder symbol_holder = new StringHolder("ACMA");
stock_manager_rcv.find_closest_symbol(symbol_holder);
```

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 reply_available(
    omg.org.CORBA.Object reply_poller,
    String operation,
    byte[] cookie)
{
    StockManager poller
        = StockManagerHelper.narrow(reply_poller);

    String id = new String(cookie);

    if( id.equals( "add stock") ) {
        boolean stock_added.add_stock("", 0.0);

        ...
    }
    else
    if( id.equals("find symbol") ) {
        StringHolder symbol_holder = new StringHolder("");
        boolean closest_found
            = poller.find_closest_symbol(symbol_holder);
        ...
    }
}
```

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，別の呼び出しフェーズ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，後で別のフェーズ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。
org.omg.CORBA.NameValuePair nv = new org.omg.CORBA.NameValuePair();
nv.id = new String("WaitReply");
nv.value = orb.create_any();
nv.value.insert_boolean(true);
org.omg.CORBA.NameValuePair[] props
    = new org.omg.CORBA.NameValuePair[]{nv};

RequestProxy 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。
StockManager stock_manager_rcv
    = StockManagerHelper.narrow(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。
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() == false ) {
    // 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 poller = StockManagerHelper.narrow(current.the_poller());
do { stock_added = poller.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
org.omg.CORBA.NameValuePair nv = new org.omg.CORBA.NameValuePair();
nv.id = new String("ReplyDropping");
nv.value = orb.create_any();
nv.value.insert_octet(0x01);
org.omg.CORBA.NameValuePair[] props
    = new org.omg.CORBA.NameValuePair[]{nv};

RequestProxy 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/PollingClientPortable.java

void yield_non_rna(org.omg.CORBA.NO_RESPONSE e) {
    if(e.minor != NativeMessaging.REPLY_NOT_AVAILABLE.value) {
        throw e;
    }
}

...

// 抑制モードをオフにします。
current.suppress_mode(false);

// 複数の要求を受信者に送信し，
// 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 pollers[2];
try{ stock_manager_rcv.add_stock("ACME", 100.5)); }
catch(org.omg.CORBA.NO_RESPONSE e) { yield_non_rna(e); }
pollers[0] = StockManagerHelper.narrow(current.the_poller());
StringHolder symbol_holder = new StringHolder("ACMA");
try{ stock_manager_rcv.find_closest_symbol(symbol_holder)); }
catch(org.omg.CORBA.NO_RESPONSE e) { yield_non_rna(e); }
pollers[1] = StockManagerHelper.narrow(current.the_poller());
```

```
// 2 つの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);

boolean stock_added;
do { try{ stock_added = pollers[0].add_stock("", 0.0)) }
    catch(org.omg.CORBA.NO_RESPONSE e) { yield_non_rna(e); } }
while(current.reply_not_available());

boolean closest_found;
do { try{ closest_found = pollers[1].find_closest_symbol(symbol_holder)) }
    catch(org.omg.CORBA.NO_RESPONSE e) { yield_non_rna(e); } }
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，251 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，259 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，261 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，261 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，263 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，261 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。263 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。267 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，257 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，251 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，262 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，258 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。265 ページの「poll()」と257 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段階と応答ポーリングフェーズは，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／プルフェーズ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
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。254 ページの「コールバックモデル」と258 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。

254 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。

```
// Java でのリクエストエージェントリファレンス取得
org.omg.CORBA.Object ref
    = orb.resolve_initial_references("RequestAgent");
NativeMessaging.RequestAgentEx agent
    = NativeMessaging.RequestAgentExHelper.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

Java 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。Java クライアントコンポーネントのネイティブメッセージングは，`vbroker.orb.enableNativeMessaging` を `true` に設定することによって明示的に初期化する必要があります（デフォルト値は `false`）。

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。272 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 Messaging 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，272 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。

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。286 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 VisiBroker for 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。詳細は，162 ページの「スマートエージェント（osagent）の起動」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用して制御できます。

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。

Java においてオブジェクトインプリメンテーションの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 java: <interface_name>Helper.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，指定できるものは一度に 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，277 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 の指定方法の詳細は，277 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，指定できるものは一度に 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，指定できるものは一度に 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，指定できるものは一度に 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 を指定した場合にのみ，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。開発およびテストフェーズでは，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 に登録できるものは，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，指定できるものは一度に 1 つです。リポジトリ ID の指定方法の詳細は，277 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，指定できるものは一度に 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，指定できるものは一度に 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，指定できるものは一度に 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 を指定した場合にのみ，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。

オプション	必須	説明
-version	いいえ	このツールのバージョンを表示します。
-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ものは，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 \
        -java factory_r -a ReentrantServer -p unshared
```

メモ 上の例では，指定された Java クラスが CLASSPATH にあるものとします。

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 \
        -java Server -a CreditUnion -p unshared -e DEBUG=1
```

メモ 上の例では，指定された Java クラスが CLASSPATH にあるものとします。

上の登録では，要求されたサーバーを生成するとき，次のコマンドを実行するように OAD に指示しています。

```
vbj -DDEBUG=1 Server CreditUnion
```

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 引数を使用して oadutil reg を実行します。

次の例は，UNIX シェルから Windows 上の OAD にリモート登録を行う方法です。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 \
 -java c:\\vbroker\\examples\\library\\libsrv -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 を使用してサーバーにアクセスするには，vbroker.orb.activationIOR プロパティを使用して 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 を使用し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 -java 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> vbj -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> vbj -Dvbroker.agent.enableLocator=false Client
```

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用し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
```

```
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用し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。

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。これは，283 ページの「オブジェクトの作成と登録の例」で説明されています。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。

オブジェクトの作成と登録の例

次のサンプルコードは，CreationImplDef クラスと OAD.reg_implementation() メソッドで OAD サーバーを登録する方法を示しています。このメカニズムは独立した管理プログラムの中で使用されることもあり，必ずしもオブジェクトインプリメンテーション自身で使用されるわけではありません。オブジェクトインプリメンテーションの中で使用する場合は，オブジェクトインプリメンテーションをアクティブ化する前に，これらの作業を行う必要があります。

ORB オブジェクトを作成し，OAD に登録する

```
// Register.java
import com.inprise.vbroker.Activation.*;
import com.inprise.vbroker.extension.*;
public class Register{
   public static void main(String[] args) {
      // ORB を初期化します。
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
      // OAD を検索します。
```

```
        try {
           OAD anOAD =
              OADHelper.bind(orb);
           // ImplDef を作成します。
           CreationImplDef _implDef = new
                    com.inprise.vbroker.extension.CreationImplDef();
           _implDef.repository_id = "IDL:Bank/AccountManager:1.0";
           _implDef.object_name = "BankManager";
           _implDef.path_name = "vbj";
           _implDef.id = new byte[0];
           _implDef.activation_policy =
                    com.inprise.vbroker.extension.Policy.SHARED_SERVER;
           _implDef.env = new String[0];
           String[] str = new String[1];
           str[0] = "Server";
           _implDef.args = str;
           try {
              anOAD.reg_implementation(_implDef);
           } catch (Exception e) {
              System.out.println("Caught " + e);
           }
        }
        catch (org.omg.CORBA.NO_IMPLEMENT e) {
        }
     }
  }
```

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 で登録解除できるものは，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 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，指定できるものは一度に 1 つです。リポジトリ ID の指定方法の詳細は，277 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，指定できるものは一度に 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，指定できるものは一度に 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 を指定した場合にのみ，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> を使用して登録した 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。

オペレーション	説明
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。

メモ　irep サーバーは，内部的なデータの一貫性を維持するために，ロールバックファイルを必要とします。たとえば，irep サーバーを初めて起動した場合など，ファイルがまだ存在しなければ作成されます。コマンドラインで指定された IRepName がロールバックファイルの名前の作成に使用されます。この名前に，使用するプラットフォームの有効なファイルシステム文字だけが含まれていることを確認してください。指定された名前に存在しないディレクトリ名が含まれている場合は，そのディレクトリが自動的に作成されます。

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「Java 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
```

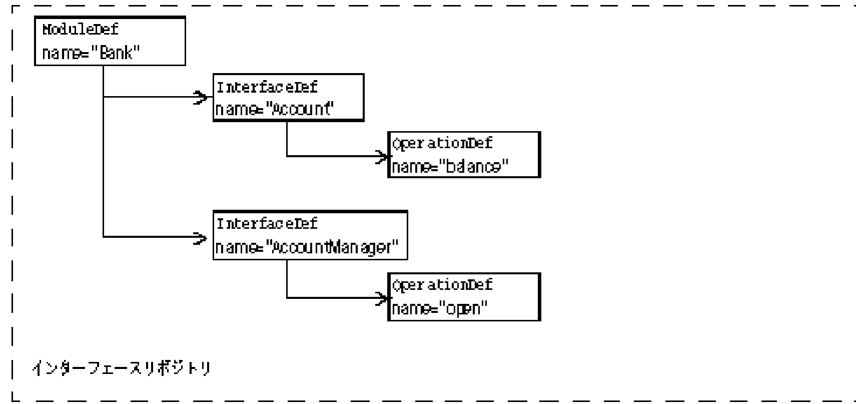
インターフェースリポジトリの構造体の概要

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。
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。

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（続き）

インターフェース	継承する側	主なクエリーメソッド
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。

```
package org.omg.CORBA;
public interface Repository extends Container {
    . . .
    org.omg.CORBA.Contained lookup_id(string id);
    org.omg.CORBA.PrimitiveDef get_primitive(org.omg.CORBA.PrimitiveKind kind);
    org.omg.CORBA.StringDef create_string(long bound);
    org.omg.CORBA.SequenceDef create_sequence(long bound,
            org.omg.CORBA.IDLType element_type);
    org.omg.CORBA.ArrayDef create_array(long length,
            org.omg.CORBA.IDLType 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，291 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，292 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

```
//Client.java
import org.omg.CORBA.InterfaceDef;
import org.omg.CORBA.InterfaceDefHelper;
import org.omg.CORBA.Request;
import java.util.Random;
public class Client {
   public static void main(String[] args) {
      try {
         // ORB を初期化します。
         org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
         // マネージャの ID を取得します。
         byte[] managerId = "BankManager".getBytes();
         // 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 manager =
               Bank.AccountManagerHelper.bind(orb, "/bank_ir_poa", managerId);
         // 口座名またはデフォルトとして args[0] を使用します。
         String name = args.length > 0 ? args[0] : "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 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。
         System.out.println("The balance in " + name + "'s account is $" +
               balance);
         // 新しい残高を計算して設定します。
         balance = args.length > 1 ?Float.parseFloat(args[1]) :
               Math.abs(new Random().nextInt()) % 100000 / 100f;
         account.balance(balance);
 // 残高の明細を取得できるかを調べて，取得できる場合はそれを出力します。
String desc = getDescription(account);
System.out.println("Balance description:¥n" + desc);
} catch (org.omg.CORBA.SystemException e) {
System.err.println("System exception caught:" + e);
} catch (Exception e) {
System.err.println("Unexpected exception caught:");
e.printStackTrace();
}
}
   static String getDescription (Bank.Account account) {
      // この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取得します。
      InterfaceDef accountDef =
               InterfaceDefHelper.narrow(account._get_interface_def());
      // この特定のインプリメンテーションが「describe」オペレーションを
      // サポートしているかをチェックします。
      if (accountDef.lookup("describe") != null) {
         // ここでは，静的なスケルトンのメソッドを使用できません
         // スケルトンの作成時に，Account インターフェースの IDL のバージョンに，
         // IDL になかったからです。かわりに DII を使用します。
         Request request = account._request("describe");
         request.result().value().insert_string("");
         request.invoke();
         return request.result().value().extract_string();
      } else {
```

```
            return "<no description>";
        }
    }
}
```

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，idl2java 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。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. 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。
2. Request オブジェクトを作成する。
3. 要求のパラメータおよび返される結果を初期化する。
4. 要求を呼び出し，結果を待つ。
5. 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。

idl2java コンパイラの使い方

idl2java コンパイラのフラグ（-dynamic_marshal）をオンにすると，DII を使用するスタブコードが生成されます。どの型の DII の使い方を理解する場合にも，次の手順を実行します。

1. IDL ファイルを作成する。
2. -dynamic_marshal を使って生成する。
3. スタブコードを確認する。

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 org.omg.CORBA.Object を戻します。

```
. . .
org.omg.CORBA.Object account;
try {
   // ORB を初期化します。
 org.omg.CORBA.ORB.init(args, null);
} catch(Exception e) {
   System.err.println ("Failure during ORB_init");
   e.printStackTrace();
}
. . .
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) {
   System.err.println ("Error binding to account" );
   excep.printStackTrace();
}
System.out.println ("Bound to account object");
. . .
```

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 インターフェース

次のサンプルコードは Request インターフェースです。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，要求のターゲットが暗黙的に設定されます。Request を作成するとき，オペレーションの名前を指定する必要があります。

```
package org.omg.CORBA;
public abstract class Request  {
   public abstract org.omg.CORBA.Object target();
   public abstract java.lang.String operation();
   public abstract org.omg.CORBA.NVList arguments();
   public abstract org.omg.CORBA.NamedValue result();
   public abstract org.omg.CORBA.Environment env();
   public abstract org.omg.CORBA.ExceptionList exceptions();
   public abstract org.omg.CORBA.ContextList contexts();
   public abstract void ctx(org.omg.CORBA.Context ctx);
   public abstract org.omg.CORBA.Context ctx();
```

```
    public abstract org.omg.CORBA.Any add_in_arg();
    public abstract org.omg.CORBA.Any add_named_in_arg(
    public abstract org.omg.CORBA.Any add_inout_arg();
    public abstract org.omg.CORBA.Any add_named_inout_arg(
    public abstract org.omg.CORBA.Any add_out_arg();
    public abstract org.omg.CORBA.Any add_named_out_arg(
    public abstract void set_return_type(
    public abstract org.omg.CORBA.Any return_value();
    public abstract void invoke();
    public abstract void send_oneway();
    public abstract void send_deferred();
    public abstract void get_response();
    public abstract boolean 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，org.omg.CORBA.Object インターフェースから提供されるメソッドを示します。

```
package org.omg.CORBA;
public interface Object {
. . .
    public org.omg.CORBA.Request _request(java.lang.String operation;
    public org.omg.CORBA.Request _create_request(
          org.omg.CORBA.Context ctx,
          java.lang.String operation,
          org.omg.CORBA.NVList arg_list,
          org.omg.CORBA.NamedValue result
    )
    public org.omg.CORBA.Request _create_request(
          org.omg.CORBA.Context ctx,
          java.lang.String operation,
          org.omg.CORBA.NVList arg_list,
          org.omg.CORBA.NamedValue result,
          org.omg.CORBA.ExceptionList exceptions,
          org.omg.CORBA.ContextList contexts
    )
. . .
}
```

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

305 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。

```
// Client.java
public class Client {
   public static void main(String[] args) {
      if (args.length ! = 2) {
         System.out.println("Usage: vbj Client <manager-name> <account-
name>¥n");
         return;
      }
      String managerName = args[0];
      String accountName = args[1];
      org.omg.CORBA.Object accountManager, account;
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
      accountManager = orb.bind("IDL:Bank/AccountManager:1.0",
            managerName, null, null);
      org.omg.CORBA.Request request = accountManager._request("open");
      request.add_in_arg().insert_string(accountName);
      request.set_return_type(orb.get_primitive_tc(
            org.omg.CORBA.TCKind.tk_objref)
      );
      request.invoke();
      account = request.result().value().extract_Object();
      org.omg.CORBA.Request request = account._request("balance");
      request.set_return_type(orb.get_primitive_tc(
            org.omg.CORBA.TCKind.tk_float)
      );
      request.invoke();
      float balance = request.result().value().extract_float();
      System.out.println("The balance in " + accountName + "'s account is $" +
balance);
   }
}
```

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。

```
package org.omg.CORBA;
public abstract class NVList {
   public int count();
   public void add(int flags);
   public void add_item(java.lang.String name, int flags);
   public void add_value(
         java.lang.String name,
         org.omg.CORBA.Any value,
         int flags
   );
   public org.omg.CORBA.NamedValue item(int index);
   public void remove(int 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。

このクラスにコンストラクタはありません。NamedValue オブジェクトへのリファレンスを取得するには，ORB.create_named_value メソッドを使用します。

```
package org.omg.CORBA;
public abstract class NamedValue {
   public java.lang.String name();
   public org.omg.CORBA.Any value();
   public int flags();
}
```

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，306 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リファレンスを定義する TypeCode へのリファレンスがあります。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。

```
package org.omg.CORBA;
public abstract class Any {
   public abstract TypeCode type();
   public abstract void type(TypeCode type);
   public abstract void read_value(InputStream input, TypeCode type);
   public abstract void write_value(OutputStream output);
   public abstract boolean equal(Any rhs);
```

```
    . . .
}
```

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，TCKind クラスで定義された値の 1 つで表されるパラメータリストプロパティがあります。

メモ このクラスにコンストラクタはありません。ORB.get_primitive_tc メソッドまたは ORB.create_*_tc メソッドの 1 つを使用して，TypeCode オブジェクトを作成します。

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：

```
public abstract class TypeCode extends java.lang.Object
      implements org.omg.CORBA.portable.IDLEntity {
   public abstract boolean equal(org.omg.CORBA.TypeCode tc);
   public boolean equivalent(org.omg.CORBA.TypeCode tc);
   public abstract org.omg.CORBA.TCKind kind();
   public TypeCode get_compact_typecode()
   public abstract java.lang.String id()
         throws org.omg.CORBA.TypeCodePackage.BadKind;
   public abstract java.lang.String name()
         throws org.omg.CORBA.TypeCodePackage.BadKind;
   public abstract int member_count()
         throws org.omg.CORBA.TypeCodePackage.BadKind;
   public abstract java.lang.String member_name(int index)
         throws org.omg.CORBA.TypeCodePackage.BadKind,
               org.omg.CORBA.TypeCodePackage.Bounds;
   public abstract org.omg.CORBA.TypeCode member_type(int index)
         throws org.omg.CORBA.TypeCodePackage.BadKind,
               org.omg.CORBA.TypeCodePackage.Bounds;
   public abstract org.omg.CORBA.Any member_label(int index)
         throws org.omg.CORBA.TypeCodePackage.BadKind,
               org.omg.CORBA.TypeCodePackage.Bounds;
   public abstract org.omg.CORBA.TypeCode discriminator_type()
         throws org.omg.CORBA.TypeCodePackage.BadKind;
   public abstract int default_index()
         throws org.omg.CORBA.TypeCodePackage.BadKind;
   public abstract int length()
         throws org.omg.CORBA.TypeCodePackage.BadKind;
   public abstract org.omg.CORBA.TypeCode content_type()
         throws org.omg.CORBA.TypeCodePackage.BadKind;
   public short fixed_digits()
         throws org.omg.CORBA.TypeCodePackage.BadKind;
   public short fixed_scale()
         throws org.omg.CORBA.TypeCodePackage.BadKind;
   public short member_visibility(int index)
         throws org.omg.CORBA.TypeCodePackage.BadKind,
               org.omg.CORBA.Bounds;
   public short type_modifier()
         throws org.omg.CORBA.TypeCodePackage.BadKind;
   public TypeCode concrete_base_type()
         throws org.omg.CORBA.TypeCodePackage.BadKind;
}
```

DII 要求の送信と結果の受信

Request クラスは，303 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 オブジェクトへのリファレンスを返します。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(orb.get_primitive_tc
          (org.omg.CORBA.TCKind.tk_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。
    float balance;
    org.omg.CORBA.Any balance_result = request.return_value();
    balance = balance_result.extract_float();
    // 残高を出力します。
    System.out.println("The balance in " + name + "'s account is $" +
           balance);
} catch(Exception e) {
    e.printStackTrace();
}
```

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。
    org.omg.CORBA.Request request = manager._request("open");
    // 要求の引数を作成します。
    org.omg.CORBA.Any customer = orb.create_any();
    customer.insert_string(name);
    org.omg.CORBA.NVList arguments = request.arguments();
    arguments.add_value("name" , customer, org.omg.CORBA.ARG_IN.value);
    // 結果の型を設定します。
    request.set_return_type(orb.get_primitive_tc
          (org.omg.CORBA.TCKind.tk_objref));
    // 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();
    Thread.currentThread().sleep(1000);
    while (!request.poll_response()) {
       System.out.println("  Waiting for response...");
       Thread.currentThread().sleep(1000);  // 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。
    org.omg.CORBA.Object account;
    org.omg.CORBA.Any open_result = request.return_value();
    account = open_result.extract_Object();
    . . .
} catch(Exception e) {
    e.printStackTrace();
}
```

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。

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

```
package org.omg.CORBA;
public abstract class ORB {
   public abstract org.omg.CORBA.Environment create_environment();
   public abstract void send_multiple_requests_oneway(org.omg.CORBA.Request[]
reqs);
   public abstract void send_multiple_requests_deferred(org.omg.CORBA.Request[]
reqs);
   public abstract boolean poll_next_response();
   public abstract org.omg.CORBA.Request 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。

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，`idl2java` 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，`idl2java` 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。

idl2java コンパイラの使い方

`idl2java` コンパイラのフラグ（`-dynamic_marshal`）をオンにすると，DSI を使用するスケルトンコードが生成されます。どの型の DSI の使い方を理解する場合にも，IDL ファイル

を作成し，-dynamic_marshal を使って生成し，コンパイルし，結果のスケルトンコードを調べます。

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，-dynamic_marshal 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，org.omg.PortableServer.DynamicImplementation インターフェー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。

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

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。

```
import java.util.*;
import org.omg.PortableServer.*;
public class AccountImpl extends DynamicImplementation {
   public AccountImpl(org.omg.CORBA.ORB orb, POA poa) {
      _orb = orb;
      _poa = poa;
   }
   public synchronized org.omg.CORBA.Object get(String name) {
      org.omg.CORBA.Object obj;
      // 口座が存在するかどうかをチェックします。
      Float balance = (Float)_registry.get(name);
      if (balance == null) {
         // 新しい口座を作成するまでの間を設けます。
         try {
            Thread.currentThread().sleep(3000);
         } catch (Exception e) {
            e.printStackTrace();
         }
         // 0 ～ 1000 ドルの範囲で口座に残高を設定します。
         balance = new Float(Math.abs(_random.nextInt()) % 100000 / 100f);
         // 新しい口座を出力します。
         System.out.println("Created " + name + "'s account: " +
               balance.floatValue());
         _registry.put(name, balance);
      }
      // オブジェクトリファレンスを返します。
      byte[] accountId = name.getBytes();
      try {
         obj = _poa.create_reference_with_id(accountId, "IDL:Bank/
Account:1.0");
      } catch (org.omg.PortableServer.POAPackage.WrongPolicy e) {
         throw new org.omg.CORBA.INTERNAL(e.toString());
      }
      return obj;
   }
   public String[] _all_interfaces(POA poa, byte[] objectId) { return null; }
   public void invoke(org.omg.CORBA.ServerRequest request) {
      Float balance;
      // オブジェクト ID から口座名を取得します。
      String name = new String(_object_id());
      // オペレーション名が正しいことを確認します。
      if (!request.operation().equals("balance")) {
         throw new org.omg.CORBA.BAD_OPERATION();
      }
      // 残高を求め，結果を格納します。
      org.omg.CORBA.NVList params = _orb.create_list(0);
      request.arguments(params);
      balance = (Float)_registry.get(name);
      if (balance == null) {
         throw new org.omg.CORBA.OBJECT_NOT_EXIST();
      }
      org.omg.CORBA.Any result = _orb.create_any();
      result.insert_float(balance.floatValue());
      request.set_result(result);
      System.out.println("Checked " + name + "'s balance: " +
            balance.floatValue());
   }
   private Random _random = new Random();
   static private Hashtable _registry = new Hashtable();
   private POA _poa;
   private org.omg.CORBA.ORB _orb;
}
```

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。

```
import org.omg.PortableServer.*;
public class AccountManagerImpl extends DynamicImplementation {
   public AccountManagerImpl(org.omg.CORBA.ORB orb, AccountImpl accounts) {
      _orb = orb;
      _accounts = accounts;
   }
   public synchronized org.omg.CORBA.Object open(String name) {
      return _accounts.get(name);
   }
   public String[] _all_interfaces(POA poa, byte[] objectId) { return null; }
   public void invoke(org.omg.CORBA.ServerRequest request) {
      // オペレーション名が正しいことを確認します。
      if (!request.operation().equals("open")) {
         throw new org.omg.CORBA.BAD_OPERATION();
      }

 // 入力パラメータを取得します。
String name = null;
try {
org.omg.CORBA.NVList params = _orb.create_list(1);
org.omg.CORBA.Any any = _orb.create_any();
any.insert_string(new String(""));
params.add_value("name", any, org.omg.CORBA.ARG_IN.value);
request.arguments(params);
name = params.item(0).value().extract_string();
} catch (Exception e) {
throw new org.omg.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。
org.omg.CORBA.Object account = open(name);
org.omg.CORBA.Any result = _orb.create_any();
result.insert_Object(account);
request.set_result(result);
}
private AccountImpl _accounts;
private org.omg.CORBA.ORB _orb;
}
```

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。

```
package org.omg.CORBA;
public abstract class ServerRequest {
   public java.lang.String operation();
   public void arguments(org.omg.CORBA.NVList args);
   public void set_result(org.omg.CORBA.Any result);
   public void set_exception(org.omg.CORBA.Any except);
   public abstract org.omg.CORBA.Context ctx();
   // 次のメソッドは使用されなくなります。
   public java.lang.String op_name(); // operation() を使用します。
   public void params(org.omg.CORBA.NVList params); // arguments() を使用します。
   public void result(org.omg.CORBA.Any result); // set_result() を使用します。
   public abstract void except(org.omg.CORBA.Any except); // set_exception() を
使用します。
}
```

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。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。

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，第 3 章「VisiBroker を使ったサンプルアプリケーションの開発」で紹介した元のサンプルとほとんど同じです。

```
import org.omg.PortableServer.*;
public class Server {
   public static void main(String[] args) {
      try {
         // ORB を初期化します。
         org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
         // ルート POA へのリファレンスを取得します。
         POA rootPOA =
                POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
         // POA マネージャを取得します。
```

```
            POAManager poaManager = rootPOA.the_POAManager();
            // 適切なポリシーで accountPOA を作成します。
            org.omg.CORBA.Policy[] accountPolicies = {
               rootPOA.create_servant_retention_policy(
                     ServantRetentionPolicyValue.NON_RETAIN),
               rootPOA.create_request_processing_policy(
                     RequestProcessingPolicyValue.USE_DEFAULT_SERVANT)
            };
            POA 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。
            AccountImpl accountServant = new AccountImpl(orb, accountPOA);
            accountPOA.set_servant(accountServant);
            // 適切なポリシーで managerPOA を作成します。
            org.omg.CORBA.Policy[] managerPolicies = {
               rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT),
               rootPOA.create_request_processing_policy(
                     RequestProcessingPolicyValue.USE_DEFAULT_SERVANT)
            };
            POA 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 = new AccountManagerImpl(orb,
                           accountServant);
            managerPOA.set_servant(managerServant);
            // POA マネージャをアクティブ化します。
            poaManager.activate();
            System.out.println("AccountManager is ready");
            // 着信要求を待機します。
            orb.run();
         } catch(Exception e) {
            e.printStackTrace();
         }
      }
   }
```

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。

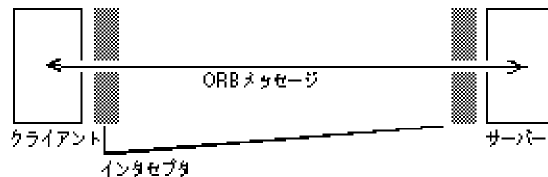
ポータブルインターセプタの詳細については，OMG Final Adopted Specification（最終採用仕様）ptc/2001-04-03 の「Portable Interceptors」を参照してください。

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（C++)」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：

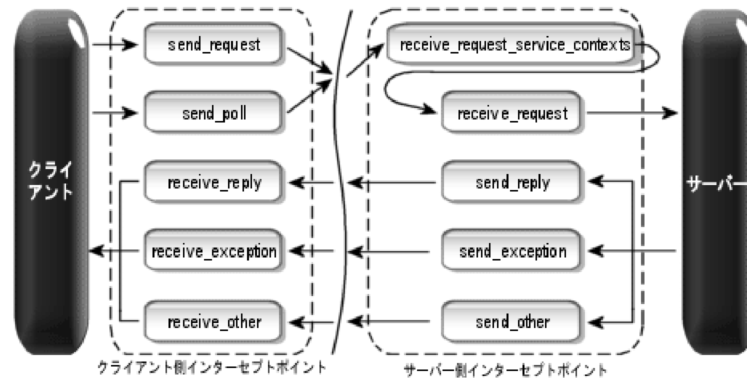
```
public interface Interceptor
   extends org.omg.CORBA.portable.IDLEntity, org.omg.CORBA.LocalInterface
{
   public java.lang.String name ( );
   public void destroy ( );
}
```

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（C++)」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（C++)」を参照してください。

```
package org.omg.PortableInterceptor;
public interface ClientRequestInterceptor
   extends Interceptor, org.omg.CORBA.portable.IDLEntity,
```

```
org.omg.CORBA.LocalInterface
{
   public void send_request(ClientRequestInfo ri) throws ForwardRequest;
   public void send_poll(ClientRequestInfo ri) throws ForwardRequest;
   public void receive_reply(ClientRequestInfo ri);
   public void receive_exception(ClientRequestInfo ri) throws ForwardRequest;
   public void receive_other(ClientRequestInfo ri) throws ForwardRequest;
}
```

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（C++）」を参照してください。

ServerRequestInterceptor インターフェース：

```
package org.omg.PortableInterceptor;
public interface ServerRequestInterceptor
   extends Interceptor, org.omg.CORBA.portable.IDLEntity,
org.omg.CORBA.LocalInterface
{
   public void receive_request_service_contexts(ServerRequestInfo ri)
      throws ForwardRequest;
   public void receive_request(ServerRequestInfo ri) throws ForwardRequest;
   public void send_reply(ServerRequestInfo ri);
   public void send_exception(ServerRequestInfo ri) throws ForwardRequest;
   public void send_other(ServerRequestInfo ri) throws ForwardRequest;
}
```

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（C++）」を参照してください。

```
package org.omg.PortableInterceptor;
public interface IORInterceptor
   extends Interceptor, org.omg.CORBA.portable.IDLEntity,
org.omg.CORBA.LocalInterface
{
   public void establish_components(IORInfo info);
   public void components_established(IORInfo info);
   public void adapter_manager_state_changed(int id, short state);
```

```
   public void adapter_state_changed(
      ObjectReferenceTemplate[] templates, short state);
}
```

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 インターフェース：

```
package org.omg.PortableInterceptor;
public interface Current
   extends org.omg.CORBA.CurrentOperations, org.omg.CORBA.portable.IDLEntity
{
   public org.omg.CORBA.Any get_slot(int id) throws InvalidSlot;

   public void set_slot(int id, org.omg.CORBA.Any data) throws InvalidSlot;
}
```

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，326 ページの「CodecFactory」を参照してください。

Codec インターフェース：

```
package org.omg.IOP;
public interface Codec
   extends org.omg.CORBA.portable.IDLEntity, org.omg.CORBA.LocalInterface
{
   public byte[] encode(org.omg.CORBA.Any data) throws InvalidTypeForEncoding;
   public org.omg.CORBA.Any decode(byte[] data) throws FormatMismatch;
   public byte[] encode_value(org.omg.CORBA.Any data) throws
InvalidTypeForEncoding;
   public org.omg.CORBA.Any decode_value(byte[] data, org.omg.CORBA.TypeCode
tc)
      throws FormatMismatch, TypeMismatch;
}
```

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 インターフェース：

```
package org.omg.IOP;
public interface CodecFactory
   extends org.omg.CORBA.portable.IDLEntity, org.omg.CORBA.LocalInterface
```

```
{
   public Codec create_codec(Encoding enc) throws UnknownEncoding;
}
```

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。

例を次に示します。`PortableInterceptor` の作成

```
import org.omg.PortableInterceptor.*;

public class SampleClientRequestInterceptor extends org.omg.CORBA.LocalObject
   implements ClientRequestInterceptor
{
   public java.lang.String name() {
      return "SampleClientRequestInterceptor";
   }

   public void send_request(ClientRequestInfo ri)
      throws ForwardRequest {
      ....... // 実際のインターセプタコード
   }

   public void send_poll(ClientRequestInfo ri)
      throws ForwardRequest {
      ....... // 実際のインターセプタコード
   }

   public void receive_reply(ClientRequestInfo ri) {
      ....... // 実際のインターセプタコード
   }

   public void receive_exception(ClientRequestInfo ri)
      throws ForwardRequest {
      ....... // 実際のインターセプタコード
   }

   public void receive_other(ClientRequestInfo ri)
      throws ForwardRequest {
      ....... // 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 インターフェース：

```
package org.omg.PortableInterceptor;

public interface ORBInitializer
   extends org.omg.CORBA.portable.IDLEntity, org.omg.CORBA.LocalInterface
{
   public void pre_init(ORBInitInfo info);
   public void post_init(ORBInitInfo info);
}
```

ORBInitInfo インターフェース：

```
package org.omg.PortableInterceptor;
public interface ORBInitInfo
   extends org.omg.CORBA.portable.IDLEntity, org.omg.CORBA.LocalInterface
{
   public java.lang.String[] arguments();
   public java.lang.String orb_id();
   public CodecFactory codec_factory();
   public void register_initial_reference(java.lang.String id,
org.omg.CORBA.Object obj)
      throws InvalidName;
   public void resolve_initial_references(java.lang.String id) throws
InvalidName;
   public void add_client_request_interceptor(ClientRequestInterceptor
interceptor)
      throws DuplicateName;
   public void add_server_request_interceptor(ServerRequestInterceptor
interceptor)
      throws DuplicateName;
   public void add_ior_interceptor(IORInterceptor interceptor) throws
DuplicateName;
   public int allocate_slot_id();
   public void register_policy_factory(int type, PolicyFactory policy_factory);
}
```

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。

register_orb_initializer( ) はグローバルメソッドなので，ORB に関するアプレットセキュリティが侵害されます。その結果，register_orb_initializer( ) を呼び出すかわりに，

Java ORB のプロパティを使用して，`ORBInitializers` は，VisiBroker Edition ORB に登録されます。

新しいプロパティ名は，次のような形式になります。

```
org.omg.PortableInterceptor.ORBInitializerClass.<Service>
```

`<Service>` は，`org.omg.PortableInterceptor.ORBInitializer` を実装するクラスの文字列名です。

`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
```

例を次に示します。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。

```
package com.abc.Monitoring;

import org.omg.PortableInterceptor.Interceptor;
import org.omg.PortableInterceptor.ORBInitializer;
import org.omg.PortableInterceptor.ORBInitInfo;

public class MonitoringService extends org.omg.CORBA.LocalObject
   implements org.omg.PortableInterceptor.ORBInitializer
{
   void pre_init(ORBInitInfo 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 info)
   {
      // この init ポイントは必要ではありません。
   }
}
```

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 インターフェース：

```
package com.inprise.vbroker.PortableInterceptor;

public interface IORInfoExt extends
   org.omg.CORBA.LocalInterface,
   org.omg.PortableInterceptor.IORInfo,
   com.inprise.vbroker.PortableInterceptor.IORInfoExtOperations,
   org.omg.CORBA.portable.IDLEntity
{
   public void add_server_request_interceptor(ServerRequestInterceptor
interceptor)
      throws DuplicateName;
   public java.lang.String[] full_poa_name();
}
```

システム例外の挿入と抽出

SystemExceptions を Any に簡単に挿入または Any から抽出するために，VisiBroker for Java にはユーティリティヘルパークラスが提供されています。com.inprise.vbroker.PortableInterceptor.SystemExceptionHelper クラスは SystemExceptions を Any に挿入または，Any から抽出するメソッドを提供します。次のパッケージをインポートする必要があります。

```
import com.inprise.vbroker.PortableInterceptor.*;
```

この 2 つメソッドは，次のシグニチャを持ちます。

```
public static void insert (final org.omg.CORBA.Any any, final
org.omg.CORBA.SystemException se);
public static org.omg.CORBA.SystemException extract (final org.omg.CORBA.Any
any);
```

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。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。
- received_exception() と received_exception_id():アプリケーションによってユーザー例外が生成された場合は，常に CORBA::UNKNOWN 例外とそれぞれのリポジトリ ID を返します。

ServerRequestInfo の制限

- exceptions() は値を返しません。動的呼び出しと静的スタブベースの呼び出しの両方で CORBA::NO_RESOURCES 例外が発生します。
- contexts() は，呼び出し時に利用できるコンテキストのリストを返します。
- sending_exception() は，動的な呼び出しが行われた場合でのみ（ユーザー例外を Any に挿入できる場合，またはその TypeCode 情報が利用できる場合），正しいユーザー例外を返します。
- arguments()，result()，contexts()，operation_contexts()：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。

ポータブルインターセプタのサンプル

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（C++）」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パッケージ

```
import org.omg.PortableInterceptor.*;
import org.omg.PortableInterceptor.ORBInitInfoPackage.*;
```

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

```
public class SampleServerLoader extends org.omg.CORBA.LocalObject implements
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 30 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(ORBInitInfo info) {
            try {
                info.add_client_request_interceptor(new
SampleClientInterceptor());
                ...
```

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，341 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。

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.java

import org.omg.PortableInterceptor.*;
import org.omg.PortableInterceptor.ORBInitInfoPackage.*;

public class SampleClientLoader extends org.omg.CORBA.LocalObject
implements ORBInitializer
{
   public void pre_init(ORBInitInfo info) {
      try {
         System.out.println("=====>SampleClientLoader: Installing ...");
         info.add_client_request_interceptor(new SampleClientInterceptor());
         System.out.println("=====>SampleClientLoader: Interceptors loaded.");
      }
      catch(DuplicateName dn) {
         System.out.println("=====>SampleClientLoader: " + dn.name + " already
   installed.");
      }
      catch(Exception e) {
         e.printStackTrace();
         throw new org.omg.CORBA.INITIALIZE(e.toString());
      }
   }

   public void post_init(ORBInitInfo info) {
      // ここでは何もしません。
   }
}
```

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(ORBInitInfo info) {
          try {
             info.add_server_request_interceptor(new
SampleServerInterceptor());
             ...
```

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

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.java

import org.omg.PortableInterceptor.*;
import org.omg.PortableInterceptor.ORBInitInfoPackage.*;
```

```
 public class SampleServerLoader extends org.omg.CORBA.LocalObject

implements ORBInitializer

 {

 public void pre_init(ORBInitInfo info) {
      try {
         info.add_server_request_interceptor(new
SampleServerInterceptor());
         System.out.println("=====>SampleServerLoader: Interceptors loaded");
   }
      catch(DuplicateName dn) {
         System.out.println("Interceptor: " + dn.name + " already installed.");
      }
      catch(Exception e) {
         e.printStackTrace();
         throw new org.omg.CORBA.INITIALIZE(e.toString());
      }
   }
 public void post_init(ORBInitInfo info) {

      // ここでは何もしません。
   }
}
```

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 String name() {
         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

```
public void send_request(ClientRequestInfo ri) throws ForwardRequest {
            ...
```

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。

```
public void send_poll(ClientRequestInfo 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。

```
public void receive_reply(ClientRequestInfo 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。

```
public void receive_exception(ClientRequestInfo ri) throws ForwardRequest {
           ...
```

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。

```
public void receive_other(ClientRequestInfo ri) throws ForwardRequest {
           ...
```

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.java

import org.omg.PortableInterceptor.*;
import org.omg.Dynamic.*;

public class SampleClientInterceptor extends org.omg.CORBA.LocalObject
implements ClientRequestInterceptor {

   public SampleClientInterceptor() {
```

```
      this("SampleClientInterceptor");
   }

public SampleClientInterceptor(String name) {
   _name = name;
}
private String _name = null;
/**
* InterceptorOperations のインプリメンテーション
*/
public String name() {
   return _name;
}

public void destroy() {
   System.out.println("=====>SampleServerLoader: Interceptors unloaded");
}

/**
* ClientRequestInterceptor のインプリメンテーション
*/

/**
* これは，VisiBroker 4.x ClientRequestInterceptor の次のメソッドに似ています。
*
* public void preinvoke_premarshal(org.omg.CORBA.Object target,
String operation,
*  ServiceContextListHolder service_contexts_holder, Closure
   closure);
*/

public void send_request(ClientRequestInfo ri) throws ForwardRequest {
   System.out.println("=====> SampleClientInterceptor id " +
     ri.request_id() +
           " send_request => " + ri.operation() +
           ": target = " + ri.target());
}

/**
* VisiBroker 4.x ClientRequestInterceptor に
* 対応するインターフェースはありません。
*/
public void send_poll(ClientRequestInfo ri) {
   System.out.println("=====> SampleClientInterceptor id " +
     ri.request_id() +
           " send_poll => " + ri.operation() +
           ": target = " + ri.target());
}

/**
* これは，VisiBroker 4.x ClientRequestInterceptor の次のメソッドに似ています。
*
* public void postinvoke(org.omg.CORBA.Object target,
*   ServiceContext[] service_contexts, InputStream payload,
*   org.omg.CORBA.Environment env, Closure closure);
*
* env は例外値を保持していません。
*/
public void receive_reply(ClientRequestInfo ri) {
   System.out.println("=====> SampleClientInterceptor id " +
```

```
      ri.request_id() +
            " receive_reply => " + ri.operation());
}

/**
* これは，VisiBroker 4.x ClientRequestInterceptor の次のメソッドに似ています。
*
* public void postinvoke(org.omg.CORBA.Object target,
*   ServiceContext[] service_contexts, InputStream payload,
*   org.omg.CORBA.Environment env, Closure closure);
*
* env は例外値を保持しています。
*/
public void receive_exception(ClientRequestInfo ri) throws ForwardRequest {
   System.out.println("=====> SampleClientInterceptor id " +
     ri.request_id() +
            " receive_exception => " + ri.operation() +
            ": exception = " + ri.received_exception());
}

/**
* これは，VisiBroker 4.x ClientRequestInterceptor の次のメソッドに似ています。
*
* public void postinvoke(org.omg.CORBA.Object target,
*   ServiceContext[] service_contexts, InputStream payload,
*   org.omg.CORBA.Environment env, Closure closure);
*
* env は例外値を保持しています。
*/
public void receive_other(ClientRequestInfo ri) throws ForwardRequest {
   System.out.println("=====> SampleClientInterceptor id " +
     ri.request_id() +
            " receive_reply => " + ri.operation() +
            ": exception = " + ri.received_exception() +
            ", reply status = "+ getReplyStatus(ri));
}

protected String getReplyStatus(RequestInfo ri) {
   switch(ri.reply_status()) {
      case SUCCESSFUL.value:
         return "SUCCESSFUL";
      case SYSTEM_EXCEPTION.value:
         return "SYSTEM_EXCEPTION";
      case USER_EXCEPTION.value:
         return "USER_EXCEPTION";
      case LOCATION_FORWARD.value:
         return "LOCATION_FORWARD";
      case TRANSPORT_RETRY.value:
         return "TRANSPORT_RETRY";
      default:
         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

```
public void receive_request_service_contexts(ServerRequestInfo ri) throws
ForwardRequest {
          ...
```

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

```
public void receive_request(ServerRequestInfo ri) throws ForwardRequest {
          ...
```

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

```
public void send_reply(ServerRequestInfo 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

```
public void send_exception(ServerRequestInfo ri) throws ForwardRequest {
          ...
```

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

```
public void send_other(ServerRequestInfo ri) throws ForwardRequest {
          ...
```

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.java

import org.omg.PortableInterceptor.*;
import org.omg.Dynamic.*;
import java.io.PrintStream;
```

```
public class SampleServerInterceptor extends org.omg.CORBA.LocalObject
implements ServerRequestInterceptor {

   private String _name = null;

   public SampleServerInterceptor() {
      this("SampleServerInterceptor");
   }

   public SampleServerInterceptor(String name) {
      _name = name;
   }
/**
* InterceptorOperations のインプリメンテーション
*/
public String name() {
      return _name;
}

public void destroy() {
   System.out.println("=====>SampleServerLoader: Interceptors unloaded");
}

/**
* ServerRequestInterceptor のインプリメンテーション
*/

/**
* これは，VisiBroker 4.x ServerRequestInterceptor の次のメソッドに似ています。
*
* public void preinvoke(org.omg.CORBA.Object target, String operation,
*   ServiceContext[] service_contexts, InputStream payload, Closure closure);
*/


public void receive_request_service_contexts(ServerRequestInfo ri)
      throws ForwardRequest {
   System.out.println("=====> SampleServerInterceptor id " + ri.request_id() +
         " receive_request_service_contexts => " + ri.operation());
}

/**
* VisiBroker 4.x SeverRequestInterceptor に
* 対応するインターフェースはありません。
*/
public void receive_request(ServerRequestInfo ri)
      throws ForwardRequest {
   System.out.println("=====> SampleServerInterceptor id " + ri.request_id() +
         " receive_request =>" + ri.operation() +
         ": object id = " + ri.object_id() +
         ", adapter_id = " + ri.adapter_id());
}

/**
* VisiBroker 4.x SeverRequestInterceptor に
* 対応するインターフェースはありません。
*/
public void send_reply(ServerRequestInfo ri) {
   System.out.println("=====> SampleServerInterceptor id " + ri.request_id() +
         " send_reply =>" + ri.operation());
```

```
}

/**
* これは，VisiBroker 4.x ServerRequestInterceptor の次のメソッドに似ています。
*
* public void postinvoke_premarshal(org.omg.CORBA.Object target,
*   ServiceContextListHolder service_contexts_holder,
*   org.omg.CORBA.Environment env, Closure closure);
*
* env は例外値を保持しています。
*/
public void send_exception(ServerRequestInfo ri)
      throws ForwardRequest {
   System.out.println("=====> SampleServerInterceptor id " + ri.request_id() +
           " send_exception =>" + ri.operation() +
           ": exception = " + ri.sending_exception() +
           ", reply status = " + getReplyStatus(ri));
}

/**
* これは，VisiBroker 4.x ServerRequestInterceptor の次のメソッドに似ています。
*
* public void postinvoke_premarshal(org.omg.CORBA.Object target,
*   ServiceContextListHolder service_contexts_holder,
*   org.omg.CORBA.Environment env, Closure closure);
*
* env は例外値を保持しています。
*/
public void send_other(ServerRequestInfo ri) throws ForwardRequest {
      System.out.print("=====> SampleServerInterceptor id " + ri.request_id() +
           " send_other =>" + ri.operation() +
           ": exception = " + ri.sending_exception() +
           ", reply status = " + getReplyStatus(ri));
}

protected String getReplyStatus(RequestInfo ri) {
   switch(ri.reply_status()) {
      case SUCCESSFUL.value:
         return "SUCCESSFUL";
      case SYSTEM_EXCEPTION.value:
         return "SYSTEM_EXCEPTION";
      case USER_EXCEPTION.value:
         return "USER_EXCEPTION";
      case LOCATION_FORWARD.value:
         return "LOCATION_FORWARD";
      case TRANSPORT_RETRY.value:
         return "TRANSPORT_RETRY";
      default:
         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。

OMG 仕様に厳密にしたがって，register_orb_initializer のマッピングを実装しました。これは，Java ORB のプロパティを使って登録されます。このサンプル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は，次のプロパティを含むプロパティファイル client.properties と server.properties を実際に読み取ります。

```
org.omg.PortableInterceptor.ORBInitializerClass.<Service>
```

<Service> は，org.omg.PortableInterceptor.ORBInitializer を実装するクラスの文字列名です。この場合，2 つのクラスは SampleClientLoader と SampleServerLoader になります。

ファイルからプロパティを読み取らないようにアプリケーションを記述する場合でも，コマンドラインオプションを使用できます。それには，次のコードを使ってアプリケーションを実行します。

```
vbj -Dorg.omg.PortableInterceptor.ORBInitializerClass.SampleClientLoader=
SampleClientLoader Client
vbj -Dorg.omg.PortableInterceptor.ORBInitializerClass.SampleServerLoader=
SampleServerLoader Server
```

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.java

import org.omg.PortableServer.*;

import java.util.Properties;
import java.io.FileInputStream;

public class Client {

   private static Properties property = null;

   public static void main(String[] args) {
      try {
         property = new Properties();
         property.load(new FileInputStream("client.properties"));

      // ORB を初期化します。
      org.omg.CORBA.ORB orb=org.omg.CORBA.ORB.init(args, property);
      // マネージャの ID を取得します。
      byte[] AccountManagerId="BankManager".getBytes();
      // 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 manager =
         Bank.AccountManagerHelper.bind(orb, "/bank_client_server_poa",
         AccountManagerId);
      // 口座名またはデフォルトとして args[0] を使用します。
      String name = null;
      name = args.length > 0 ? args[0] : "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 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。
      System.out.println("The balance in " + name + "'s account is $" +
         balance);
   }
```

```
      catch(Exception e) {
         e.printStackTrace();
      }
   }
}
```

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.java

import org.omg.PortableServer.*;
import java.util.Properties;
import java.io.FileInputStream;

public class Server {

  private static Properties property = null;

  public static void main(String[] args) {
    try {
      property = new Properties();
      property.load(new FileInputStream("server.properties"));

      // ORB を初期化します。
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, property);
      // ルート POA へのリファレンスを取得します。
      POA rootPOA =
POAHelper.narrow(orb.resolve_initial_references("RootPOA"));

      // 永続的 POA のポリシーを作成します。
      org.omg.CORBA.Policy[] policies = {
        rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT)
      };

      // 適切なポリシーで myPOA を作成します。
      POA myPOA = rootPOA.create_POA("bank_client_server_poa",
                    rootPOA.the_POAManager(), policies );

      // Account サーバントを作成します。
      AccountManagerImpl managerServant = new AccountManagerImpl();
      byte[] managerId = "BankManager".getBytes();
      myPOA.activate_object_with_id(managerId, managerServant);
      rootPOA.the_POAManager().activate();

      //Announce サーバントが準備できました
      System.out.println(myPOA.servant_to_reference(managerServant) + " is
              ready.");
      // 着信要求を待機します。
      orb.run();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

コンパイルの手順

Java 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> vbmake
```

または, 環境変数 <install_dir>¥bin が環境変数 PATH にすでに追加されている場合は, バッチファイルのアイコンをダブルクリックします。

UNIX

```
<install_dir>/examples/vbe/pi/client_server> make -f Makefile.java
```

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

インストールされているポータブルインターセプタを使って Java 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

2 つのコンソールウィンドウを開きます。

```
<install_dir>¥examples¥vbe¥pi¥client_server> start vbj Server (running under a
new command prompt window)
<install_dir>¥examples¥vbe¥pi¥client_server> vbj Client John (using a given
name)
```

または

```
<install_dir>¥examples¥vbe¥pi¥client_server> vbj Client(using the 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> vbj Server(in the first window)
<install_dir>/examples/vbe/pi/client_server> vbj Client John (in the second
window, using a given name)
```

または

```
<install_dir>/examples/vbe/pi/client_server> vbj Client (in the second window,
using the default 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。

```
package com.inprise.vbroker.InterceptorExt;
public interface BindInterceptor {
   public IORValue bind(IORValue ior,
         org.omg.CORBA.Object target,
         boolean rebind,
         Closure closure);
   public IORValue bind_failed(IORValue ior,
         org.omg.CORBA.Object target,
         Closure closure);
   public void bind_succeeded(IORValue ior,
         org.omg.CORBA.Object target,
         int Index,
         InterceptorManagerControl control,
         Closure closure);
   public void exception_occurred(IORValue ior,
         org.omg.CORBA.Object target,
         org.omg.CORBA.Environment env,
         Closure closure);
}
```

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。

```
package com.inprise.vbroker.InterceptorExt;
public interface ClientRequestInterceptor {
   public void preinvoke_premarshal(org.omg.CORBA.Object target,
         String operation,
         ServiceContextListHolder service_contexts_holder,
         Closure closure);
   public void preinvoke_postmarshal(org.omg.CORBA.Object target,
         OutputStream payload,
         Closure closure);
   public void postinvoke(org.omg.CORBA.Object target,
         ServiceContext[] service_contexts,
         InputStream payload,
         org.omg.CORBA.Environment env,
         Closure closure);
   public void exception_occurred(org.omg.CORBA.Object target,
         org.omg.CORBA.Environment env,
         Closure closure);
}
```

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。

```
package com.inprise.vbroker.InterceptorExt;
public interface POALifeCycleInterceptor {
   public void create(org.omg.PortableServer.POA poa,
         org.omg.CORBA.PolicyListHolder policies_holder,
         IORValueHolder iorTemplate,
         InterceptorManagerControl control) ;
   public void destroy(org.omg.PortableServer.POA poa);
}
```

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。

```
package com.inprise.vbroker.InterceptorExt;
public interface ActiveObjectLifeCycleInterceptor {
   public void create(byte[] oid,
         org.omg.PortableServer.Servant servant,
         org.omg.PortableServer.POA adapter);
   public void destroy (byte[] oid,
         org.omg.PortableServer.Servant servant,
         org.omg.PortableServer.POA adapter);
}
```

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。

```
package com.inprise.vbroker.InterceptorExt;
public interface ServerRequestInterceptor {
   public void preinvoke(org.omg.CORBA.Object target,
         String operation,
         ServiceContext[] service_contexts,
         InputStream payload,
         Closure closure);
   public void postinvoke_premarshal(org.omg.CORBA.Object target,
         ServiceContextListHolder service_contexts_holder,
         org.omg.CORBA.Environment env,
         Closure closure);
public void postinvoke_postmarshal(org.omg.CORBA.Object target,
         OutputStream payload,
         Closure closure);
public void exception_occurred(org.omg.CORBA.Object target,
         org.omg.CORBA.Environment env,
         Closure closure);
}
```

メモ　org.omg.CORBA.SystemException または任意のサブクラス（org.omg.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。

```
package com.inprise.vbroker.InterceptorExt;
public interface IORCreationInterceptor {
   public void create(org.omg.PortableServer.POA poa,
         IORValueHolder ior);
}
```

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。

```
public interface ServiceResolverInterceptor {
   public org.omg.CORBA.Object resolve (java.lang.String name):
}
public interface ServiceResolverInterceptorManager extends
   com.inprise.vbroker.interceptor.InterceptorManager {
      public void add (java.lang.String name,
      com.inprise.vbroker.interceptor.ServiceResolverInterceptor ¥interceptor)
;
   pubic void remove (java.lang.String name):
}
```

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。

デフォルトのインターセプタクラス

VisiBroker には，実装と拡張が可能なデフォルトのインターセプタ Java クラスが用意されています。これらのデフォルトインターセプタクラスは, インターセプタインターフェースと同じメソッドを提供します。ただし，これらのデフォルトインターセプタクラスを拡張する場合は，どのメソッドを実装（オーバーライド）するかを選択できます。これらのクラスを使用する場合は，提供されるデフォルトの動作をそのまま使用することできますし，変更することも可能です。

- DefaultBindInterceptor クラス
- DefaultClientInterceptor クラス
- DefaultServerInterceptor クラス

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，対応するインターセプタマネージャ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

表 25.1　InterceptorManagerControl オブジェクトの文字列値（続き）

値	対応するインターセプタインターフェース
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。ファクトリクラスでは，ServiceLoader インターフェースを実装します。

```
package com.inprise.vbroker.interceptor;
public interface ServiceLoader {
   // ORB.init() が呼び出されると，ORB によって呼び出されます。
   public abstract void init(org.omg.CORBA.ORB orb);
   // ORB.init() が完了した後で，ユーザーに制御が戻る前に
   // ほかのサービスの初期化にだけ利用される一定のリソースを無効にする
   // ために使用されます
   public abstract void init_complete(org.omg.CORBA.ORB orb);
   // ORB がシャットダウンされるときに呼び出されます。
   public abstract void shutdown(org.omg.CORBA.ORB orb);
}
```

メモ 350 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，vbroker.orb.dynamicLibs プロパティを設定する必要があります。このプロパティは，プロパティファイルで設定するか（第 6 章「VisiBroker のプロパティ」を参照），-D オプションを使って VisiBroker ORB に渡します。

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

353 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
- 暗号化

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>vbj -Dvbroker.orb.dynamicLibs=SampleServerLoader Server
prompt>vbj -Dvbroker.orb.dynamicLibs=SampleClientLoader Client John
```

ServiceLoader インターフェースを実装する 2 つのクラスを VisiBroker ORB サービスとして指定します。

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，352 ページの「ServiceResolverInterceptor のサンプル」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，349 ページの「VisiBroker ORB によるインターセプタの登録」を参照してください。

ServiceResolverInterceptor のサンプル

次のコードは，ServiceLoader インターフェースのインプリメンテーションサンプルです。

```
import com.inprise.vbroker.properties.*;
import com.inprise.vbroker.interceptor.*;
import com.inprise.vbroker.InterceptorExt.*;

public final class UtilityServiceLoader implements ServiceLoader,
   ServiceResolverInterceptor {
   private com.inprise.vbroker.orb.ORB _orb = null;
   private String[] _serviceNames = { "TimeService", "WeatherService"};

   public void init(org.omg.CORBA.ORB orb) {
      // resolve() で必要とされる場合に備えて
      _orb = (com.inprise.vbroker.orb.ORB) orb;

      PropertyManager pm = _orb.getPropertyManager();
      // 必要ならば，PropertyManager を使ってプロパティ設定を
      // 照会します（このサンプルでは使用しません）

      /**** 最初のリファレンスのインストール *****/
      InterceptorManagerControl control = _orb.interceptorManager();
      ServiceResolverInterceptorManager manager =

(ServiceResolverInterceptorManager)control.get_manager("ServiceResolver");
      for (int i = 0; i < _serviceNames.length; i++) {
         manager.add(_serviceNames[i], this);
      }
      /**** インストール終了 ***/

      if (_orb.debug)
         _orb.println("UtilityServices package has been initialized");
   }

   public void init_complete(org.omg.CORBA.ORB orb) {
      // 必要ならばインストール後の処理をここで実行します。
   }

   public void shutdown(org.omg.CORBA.ORB orb) {
      _orb = null;
      _serviceNames = null;
   }

   public org.omg.CORBA.Object resolve(java.lang.String service) {
      org.omg.CORBA.Object srv = null;
      byte[] serviceId = service.getBytes();
   try {
      if (service == "TimeService") {
         srv = UtilityServices.TimeServiceHelper.bind(_orb, "/
time_service_poa", serviceId);
      }
      else if (service == "WeatherService") {
         srv = UtilityServices.WeatherServiceHelper.bind(_orb,"/
weather_service_poa",
            serviceId);
      }
```

```
    } catch (org.omg.CORBA.SystemException e) {
       if (_orb.debug)
          _orb.println("UtilityServices package resolve error: " + e);
       srv = null;
    }

    return srv;
    }
}
```

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。

```
import java.util.*;
import com.inprise.vbroker.orb.*;
import com.inprise.vbroker.interceptor.*;
import com.inprise.vbroker.PortableServerExt.*;
public class SampleServerLoader implements ServiceLoader {
   public void init(org.omg.CORBA.ORB orb) {
      try {
         InterceptorManagerControl control =
               InterceptorManagerControlHelper.narrow(

orb.resolve_initial_references("VisiBrokerInterceptorControl"));
         // POA インターセプタをインストールします。
         POALifeCycleInterceptorManager poa_manager =
               (POALifeCycleInterceptorManager)
control.get_manager("POALifeCycle");
         poa_manager.add(new SamplePOALifeCycleInterceptor());
      } catch(Exception e) {
         e.printStackTrace();
         throw new org.omg.CORBA.INITIALIZE(e.toString());
      }
      System.out.println("============>SampleServerLoader:Interceptors
loaded");
   }
   public void init_complete(org.omg.CORBA.ORB orb) {
   }
   public void shutdown(org.omg.CORBA.ORB orb) {
   }
}
```

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。

```
import com.inprise.vbroker.interceptor.*;
import com.inprise.vbroker.PortableServerExt.*;
import com.inprise.vbroker.IOP.*;
```

```
public class SamplePOALifeCycleInterceptor implements POALifeCycleInterceptor {
   public void create(org.omg.PortableServer.POA poa,
         org.omg.CORBA.PolicyListHolder policies_holder,
         IORValueHolder iorTemplate,
         InterceptorManagerControl control) {
      if(poa.the_name().equals("bank_agent_poa")) {
         // 要求レベルのインターセプタを追加します。
         SampleServerInterceptor interceptor =
               new SampleServerInterceptor("MyServerInterceptor");
         // IORCreation インターセプタマネージャを取得します。
         ServerRequestInterceptorManager manager =

(ServerRequestInterceptorManager)control.get_manager("ServerRequest");
         // インターセプタを追加します。
         manager.add(interceptor);
         System.out.println("============>In POA " + poa.the_name() +
               ", 1 ServerRequest interceptor installed");
      } else
         System.out.println("============>In POA " + poa.the_name() + ".
Nothing to do.");
   }
   public void destroy(org.omg.PortableServer.POA poa) {
      // トレース用
      System.out.println("============> SamplePOALifeCycleInterceptor
destroy");
   }
}
```

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。

```
import com.inprise.vbroker.interceptor.*;
import com.inprise.vbroker.IOP.*;
import com.inprise.vbroker.CORBA.portable.*;
public class SampleServerInterceptor implements ServerRequestInterceptor {
   private String _id;
   public SampleServerInterceptor(String id) {
      _id = id;
   }
   public void preinvoke(org.omg.CORBA.Object target,
         String operation,
         ServiceContext[] service_contexts,
         InputStream payload,
         Closure closure) {
      // closure オブジェクトにこの ServerRequestInterceptor の _id を付加します。
      closure.object = new String(_id);
      System.out.println("============> SampleServerInterceptor id " +
            closure.object + " preinvoke => " + operation);
   }
   public void postinvoke_premarshal(org.omg.CORBA.Object target,
         ServiceContextListHolder service_contexts_holder,
         org.omg.CORBA.Environment env,
         Closure closure) {
      System.out.println("============> SampleServerInterceptor id " +
            closure.object + " postinvoke_premarshal");
   }
   public void postinvoke_postmarshal(org.omg.CORBA.Object target,
         OutputStream payload,
         Closure closure) {
```

```
        System.out.println("============> SampleServerInterceptor id " +
              closure.object + " postinvoke_postmarshal");
     }
     public void exception_occurred(org.omg.CORBA.Object target,
           org.omg.CORBA.Environment env,
           Closure closure) {
        System.out.println("============> SampleServerInterceptor id " +
              closure.object + " exception_occurred");
     }
  }
```

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。

```
  import com.inprise.vbroker.interceptor.*;
  import com.inprise.vbroker.IOP.*;
  import com.inprise.vbroker.CORBA.portable.*;
  public class SampleClientInterceptor implements ClientRequestInterceptor {
     private String _id;
     public SampleClientInterceptor(String id) {
        _id = id;
     }
     public void preinvoke_premarshal(org.omg.CORBA.Object target,
           String operation,
           ServiceContextListHolder service_contexts_holder,
           Closure closure) {
        // closure オブジェクトにこの ServerRequestInterceptor の _id を付加します。
        closure.object = new String(_id);
        System.out.println("============> SampleClientInterceptor id " +
              closure.object +
              " preinvoke_premarshal => " + operation);
     }
     public void preinvoke_postmarshal(org.omg.CORBA.Object target,
           OutputStream payload,
           Closure closure) {
        System.out.println("============> SampleClientInterceptor id " +
              closure.object + " preinvoke_postmarshal");
     }
     public void postinvoke(org.omg.CORBA.Object target,
           ServiceContext[] service_contexts,
           InputStream payload,
           org.omg.CORBA.Environment env,
           Closure closure) {
        System.out.println("============> SampleClientInterceptor id " +
              closure.object + " postinvoke");
     }
     public void exception_occurred(org.omg.CORBA.Object target,
           org.omg.CORBA.Environment env,
           Closure closure) {
        System.out.println("============> SampleClientInterceptor id " +
              closure.object + " exception_occurred");
     }
  }
```

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。

```
import java.util.*;
import com.inprise.vbroker.orb.*;
import com.inprise.vbroker.interceptor.*;
import com.inprise.vbroker.PortableServerExt.*;
public class SampleClientLoader implements ServiceLoader {
   public void init(org.omg.CORBA.ORB orb) {
      try {
         InterceptorManagerControl control =
               InterceptorManagerControlHelper.narrow(

orb.resolve_initial_references("VisiBrokerInterceptorControl"));
         BindInterceptorManager bind_manager =
               (BindInterceptorManager) control.get_manager("Bind");
         bind_manager.add(new SampleBindInterceptor());
      } catch(Exception e) {
         e.printStackTrace();
         throw new org.omg.CORBA.INITIALIZE(e.toString());
      }
      System.out.println("Bind Interceptors loaded");
   }
   public void init_complete(org.omg.CORBA.ORB orb) {
   }
   public void shutdown(org.omg.CORBA.ORB orb) {
   }
}
```

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。

```
import com.inprise.vbroker.interceptor.*;
import com.inprise.vbroker.IOP.*;
public class SampleBindInterceptor implements BindInterceptor {
   public IORValue bind(IORValue ior, org.omg.CORBA.Object target,
         boolean rebind, Closure closure) {
      // トレース用
      System.out.println("============> SampleBindInterceptor bind");
      return null;
   }
   public IORValue bind_failed(IORValue ior, org.omg.CORBA.Object target,
         Closure closure) {
      // トレース用
      System.out.println("============> SampleBindInterceptor bind_failed");
      return null;
   }
   public void bind_succeeded(IORValue ior, org.omg.CORBA.Object target,
         int Index, InterceptorManagerControl control,
         Closure closure) {
      // トレース用
```

```
        System.out.println("============> SampleBindInterceptor bind_succeeded");
        // クライアントリクエストインターセプタを作成します。
        SampleClientInterceptor interceptor =
        new SampleClientInterceptor("MyClientInterceptor");
        // マネージャを取得します。
        ClientRequestInterceptorManager manager =

(ClientRequestInterceptorManager)control.get_manager("ClientRequest");
        // CRQ をリストに追加します。
        manager.add(interceptor);
    }
    public void exception_occurred(IORValue ior, org.omg.CORBA.Object target,
            org.omg.CORBA.Environment env,
            Closure closure) {
        // トレース用
        System.out.println("============> SampleBindInterceptor
exception_occured");
    }
}
```

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。

```
int my response_expected =
    ((ExtendedClosure)closure).reqInfo.response_expected;
int 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，367 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，362 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

idl2java 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 `idl2java` コンパイラを実行する必要があります。これにより，次の要素が生成されます。

- 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
- オブジェクトラッパーを追加または削除するために Helper クラスに追加されるメソッド

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。

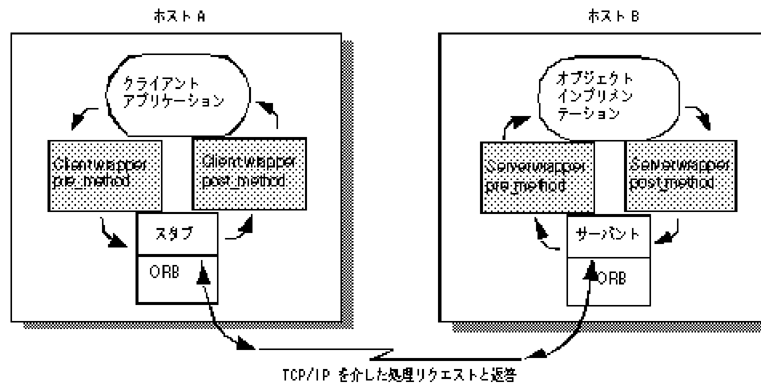
同じクライアントアプリケーションやサーバーアプリケーション内で，型付きと型なしのオブジェクトラッパーを併用することもできます。

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。

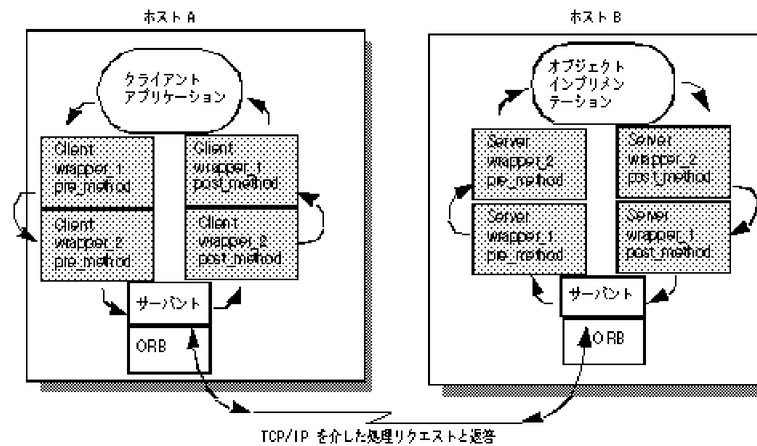
下図では，クライアントスタブメソッドの前に型なしオブジェクトラッパーの `pre_method` を呼び出してから，その後で `post_method` が呼び出される流れを示しています。また，オブジェクトインプリメンテーションについても，サーバー側の呼び出し経路を示します。

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，372 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 idl2java 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，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，UntypedObjectWrapper クラスから派生します。
5 適切な型の ChainUntypedObjectWrapperFactory にアクセスするように，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を変更します。
6 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。
7 ChainUntypedObjectWrapperFactory add メソッドで，ファクトリをチェインに追加します。

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 TimingUnTypedObjectWrapperFactory のインプリメンテーションでは，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。

```
package UtilityObjectWrappers;
import com.inprise.vbroker.interceptor.*;
public class TimingUntypedObjectWrapperFactory implements
      UntypedObjectWrapperFactory {
   public UntypedObjectWrapper create(org.omg.CORBA.Object target,
         com.inprise.vbroker.interceptor.Location loc) {
      return new TimingUntypedObjectWrapper();
   }
}
```

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，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 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 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示された pre_method は，現在の時刻を取得して private 変数に保存し，メッセージを出力します。post_method も現在の時刻を取得し，pre_method が呼び出されてからの経過時間を求めて出力します。

```
package UtilityObjectWrappers;
import com.inprise.vbroker.interceptor.*;
Public class TimingUntypedObjectWrapper implements UntypedObjectWrapper {
   private long time;
   public void pre_method(String operation,
          org.omg.CORBA.Object target,
          Closure closure) {
      System.out.println("Timing: " +
            ((com.inprise.vbroker.CORBA.Object) target)._object_name() + "->"
            + operation + "()");
      time = System.currentTimeMillis();
   }
   public void post_method(String operation,
         org.omg.CORBA.Object target,
         org.omg.CORBA.Environment env,
         Closure closure) {
      long diff = System.currentTimeMillis() - time;
      System.out.println("Timing: Time for call ¥t" +
((com.inprise.vbroker.CORBA.Object)
            target)._object_name() + "->" + operation + "() = " + diff + "
ms.");
   }
}
```

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
operation	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。
environment	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

次のコードは，サンプルファイル UntypedClient.java 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。

```
// UntypedClient.java
import com.inprise.vbroker.interceptor.*;
Public class UntypedClient {
   public static void main(String[] args) throws Exception {
      // ORB を初期化します。
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
      doMain (orb, args);
   }
   public static void domain(org.omg.CORBA.ORB orb, String[] args) throws
            Exception {
      ChainUntypedObjectWrapperFactory Cfactory =
            ChainUntypedObjectWrapperFactoryHelper.narrow(
```

```
orb.resolve_initial_references("ChainUntypedObjectWrapperFactory")
          );
 Cfactory.add(new UtilityObjectWrappers.TimingUntypedObjectWrapperFactory(),
                    Location.CLIENT);
      Cfactory.add(new

UtilityObjectWrappers.TracingUntypedObjectWrapperFactory(),
            Location.CLIENT);
      // アカウント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.Java 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.java
import com.inprise.vbroker.interceptor.*;
import org.omg.PortableServer.*;
Import com.inprise.vbroker.PortableServerExt.BindSupportPolicyValue;
import com.inprise.vbroker.PortableServerExt.BindSupportPolicyValueHelper;
import com.inprise.vbroker.PortableServerExt.BIND_SUPPORT_POLICY_TYPE;
public class UntypedServer {
   public static void main(String[] args) throws Exception {
      // ORB を初期化します。
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
      ChainUntypedObjectWrapperFactory Sfactory =
            ChainUntypedObjectWrapperFactoryHelper.narrow

(orb.resolve_initial_references("ChainUntypedObjectWrapperFactory"));
      Sfactory.add(new
               UtilityObjectWrappers.TracingUntypedObjectWrapperFactory(),
            Location.SERVER);
      // ルート POA へのリファレンスを取得します。
      POA rootPOA =
POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
      // POA が各サーバントを osagent に登録するように BindSupport
      // ポリシーを作成します。
      org.omg.CORBA.Any any = orb.create_any();
      BindSupportPolicyValueHelper.insert(any,
                  BindSupportPolicyValue.BY_INSTANCE);
      org.omg.CORBA.Policy bsPolicy =
            orb.create_policy(BIND_SUPPORT_POLICY_TYPE.value, any);
      // testPOA のポリシーを作成します。
      org.omg.CORBA.Policy[] policies = {
            rootPOA.create_lifespan_policy
                  (LifespanPolicyValue.PERSISTENT), bsPolicy
      };
      // 適切なポリシーで myPOA を作成します。
      POA myPOA = rootPOA.create_POA( "bank_agent_poa",
            rootPOA.the_POAManager(),
            policies );

      // AccountManager オブジェクトを作成します。
      AccountManagerImpl managerServant = new AccountManagerImpl();
      // サーバントの ID を決定します。
      byte[] managerId = "BankManager".getBytes();
      // 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
      rootPOA.the_POAManager().activate();
      System.out.println("AccountManager: BankManager is ready.");
      for( int i = 0; i < args.length; i++ ) {
         if( args[i].equalsIgnoreCase("-runCoLocated") ) {
            if( args[i+1].equalsIgnoreCase("Client") ){
```

```
                Client.doMain(orb, new String[0]);
            } else if( args[i+1].equalsIgnoreCase("TypedClient") ){
                TypedClient.doMain(orb, new String[0]);
            }
            if( args[i+1].equalsIgnoreCase("UntypedClient") ){
                UntypedClient.doMain(orb, new String[0]);
            }
            System.exit(1);
        }
    }
    // 着信要求を待機します。
    orb.run();
  }
}
```

型なしオブジェクトラッパーの削除

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，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。

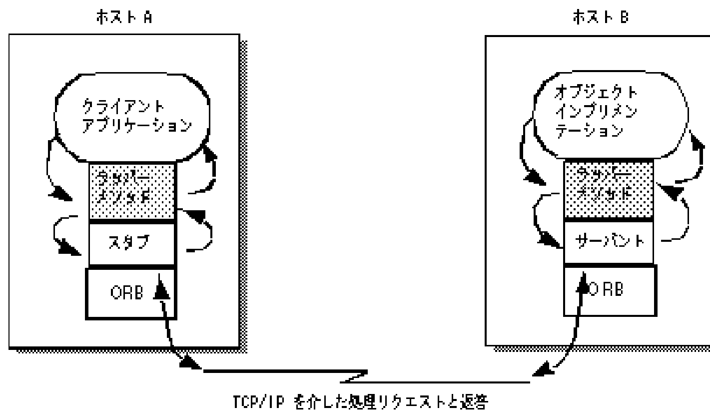
型付きオブジェクトラッパー

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，372 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



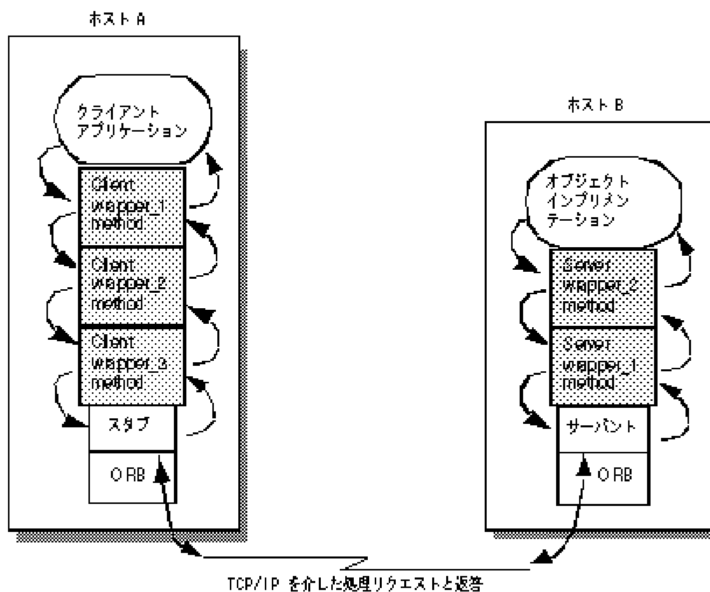
複数の型付きオブジェクトラッパーの使い方

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 super.<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。

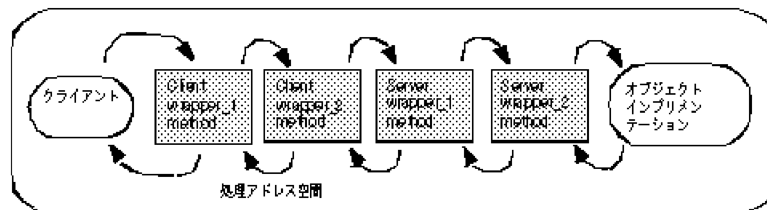
型付きオブジェクトラッパーメソッドにはチェイン内の前のメソッドに制御を戻す機能があり，この機能を使ってクライアントスタブやオブジェクトインプリメンテーションをまったく呼び出さないラッパーメソッドを作成することもできます。たとえば，頻繁に要求される処理の結果をキャッシュするようなオブジェクトラッパーメソッドを作成できます。この例では，バインドされたオブジェクトのメソッドを初めて呼び出すと，オブジェクトインプリメンテーションにオペレーションリクエストが送信されます。オブジェクトラッパーメソッドを介して制御の流れが戻る間に，その結果が格納されます。これ以降に同じメソッドが呼び出されると，オブジェクト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，372 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. idl2java 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

idl2java 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，Java 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. _initialized 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，_initialized 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。

```
package BankWrappers;
public class CachingAccountObjectWrapper extends Bank.AccountObjectWrapper {
 private boolean _initialized = false;
   private float _balance;
   public float balance() {
      System.out.println("+ CachingAccountObjectWrapper: Before calling
balance:
              ");
      try {
 if( !_initialized ) {
            _balance = super.balance();
            _initialized = true;
         } else {
            System.out.println("+ CachingAccountObjectWrapper: Returning Cached
                  value");
         }
         return _balance;
      } finally {
         System.out.println("+ CachingAccountObjectWrapper: After  calling
                  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，idl2java コンパイラがクラスに生成した addClientObjectWrapperClass メソッド（Java）の呼び出しによって，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。

```
// TypedClient.java
import com.inprise.vbroker.interceptor.*;
Public class TypedClient {
   public static void main(String[] args) throws Exception {
      // ORB を初期化します。
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
      domain (orb, args);
   }
   public static void domain(org.omg.CORBA.ORB orb, String[] args) {
      // Account オブジェクトの型付きオブジェクトラッパーを追加します。
 Bank.AccountHelper.addClientObjectWrapperClass(orb,
```

```
            BankWrappers.CachingAccountObjectWrapper.class);
        // アカウントマネージャを検索します。
        Bank.AccountManager manager =
            Bank.AccountManagerHelper.bind(orb, "BankManager");
        . . .
    }
}
```

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，Helper クラスが提供する addServerObjectWrapperClass 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る要求へのサービス提供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.java ファイルの一部を示します。

```
// TypedServer.java
import org.omg.PortableServer.*;
import com.inprise.vbroker.PortableServerExt.BindSupportPolicyValue;
import com.inprise.vbroker.PortableServerExt.BindSupportPolicyValueHelper;
import com.inprise.vbroker.PortableServerExt.BIND_SUPPORT_POLICY_TYPE;
public class TypedServer {
    public static void main(String[] args) throws Exception {
        // ORB を初期化します。
        org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
        // AccountManager オブジェクトの 2 つの型付きオブジェクトラッパーを追加します。
        Bank.AccountManagerHelper.addServerObjectWrapperClass(orb,
        BankWrappers.SecureAccountManagerObjectWrapper.class);
        Bank.AccountManagerHelper.addServerObjectWrapperClass(orb,
            BankWrappers.CachingAccountManagerObjectWrapper.class);
        // ルート POA へのリファレンスを取得します。
        POA rootPOA =
POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
        // POA が各サーバントを osagent に登録するように BindSupport
        // ポリシーを作成します。
        org.omg.CORBA.Any any = orb.create_any();
        BindSupportPolicyValueHelper.insert(any,
                        BindSupportPolicyValue.BY_INSTANCE);
        org.omg.CORBA.Policy bsPolicy =
            orb.create_policy(BIND_SUPPORT_POLICY_TYPE.value, any);
        // testPOA のポリシーを作成します。
        org.omg.CORBA.Policy[] policies = {
            rootPOA.create_lifespan_policy(LifespanPolicyValue.PERSISTENT),
                bsPolicy
        };
        // 適切なポリシーで myPOA を作成します。
        POA myPOA = rootPOA.create_POA( "lilo", rootPOA.the_POAManager(),
policies
);
        // AccountManager オブジェクトを作成します。
        AccountManagerImpl managerServant = new AccountManagerImpl();
        // サーバントの ID を決定します。
        byte[] managerId = "BankManager".getBytes();
        // 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
        rootPOA.the_POAManager()Activate();
```

```
        System.out.println("AccountManager: BankManager is ready.");

        For( int i = 0; i < args.length; i++ ) {
           if ( args[i].equalsIgnoreCase("-runCoLocated") ) {
              if( args[i+1].equalsIgnoreCase("Client") ){
                 Client.doMain(orb, new String[0]);
              } else if( args[i+1].equalsIgnoreCase("TypedClient") ){
                 TypedClient.doMain(orb, new String[0]);
              }
              if( args[i+1].equalsIgnoreCase("UntypedClient") ){
                 UntypedClient.doMain(orb, new String[0]);
              }
              System.exit(1);
           }
        }
        // 着信要求を待機します。
        orb.run();
     }
  }
```

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

Helper クラスは，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を削除するメソッドも提供します。

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.java と Server.java 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

型なしラッパーを有効にするには，コマンドラインで次の引数を指定します。

1 -Dvbroker.orb.dynamicLibs=BankWrappers.Init

2 次の表に示されている 1 つ以上のプロパティを使用します。

表 26.3 BankWrappers を有効または無効にするためのコマンドラインプロパティ

BankWrappers のプロパティ	説明
-DCachingAccount[=<client|server>]	クライアントまたはサーバーの 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インストールします。サブプロパティの値を指定しなければ，クライアントのラッパーとサーバーのラッパーの両方がインストールされます。
-DCachingAccountManager[=<client|server>]	クライアントまたはサーバーの 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インストールします。サブプロパティに値を指定しなければ，クライアントのラッパーとサーバーのラッパーの両方がインストールされます。
-DSecureAccountManager[=<client|server>]	クライアントまたはサーバーの 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インストールします。サブプロパティの値を指定しなければ，クライアントのラッパーとサーバーのラッパーの両方がインストールされます。

型付きラッパーのイニシャライザ

型付きラッパーは BankWrappers パッケージで定義され，次のコードに示すサービスイニシャライザ BankWrappers/Init.java を組み込みます。コマンドラインに -Dvbroker.orb.dynamicLibs=BankWrappers.Init を指定しておくと，vbj でクライアントやサーバーを起動するときにこのイニシャライザが呼び出されます。コマンドラインプロパティの指定に基づいて，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できます。

```
package BankWrappers;
import java.util.*;
import com.inprise.vbroker.orb.ORB;
import com.inprise.vbroker.properties.PropertyManager;
import com.inprise.vbroker.interceptor.*;
public class Init implements ServiceLoader {
   com.inprise.vbroker.orb.ORB _orb;
   public void init(final org.omg.CORBA.ORB orb) {
      _orb = (ORB) orb;
      PropertyManager pm = _orb.getPropertyManager();
      // CachingAccountObjectWrapper をインストールします。
 String val = pm.getString("CachingAccount", this.toString());
      Class c = CachingAccountObjectWrapper.class;
      if( !val.equals(this.toString())) {

 if( val.equalsIgnoreCase("client") ) {
            Bank.AccountHelper.addClientObjectWrapperClass(orb, c);
 } else if( val.equalsIgnoreCase("server") ) {
            Bank.AccountHelper.addServerObjectWrapperClass(orb, c);
 } else {
            Bank.AccountHelper.addClientObjectWrapperClass(orb, c);
            Bank.AccountHelper.addServerObjectWrapperClass(orb, c);
         }
      }
      // CachingAccountManagerObjectWrapper をインストールします。
 val = pm.getString("CachingAccountManager", this.toString());
      c = CachingAccountManagerObjectWrapper.class;
      if( !val.equals(this.toString())) {
 if( val.equalsIgnoreCase("client") ){
            Bank.AccountManagerHelper.addClientObjectWrapperClass(orb, c);
 } else if( val.equalsIgnoreCase("server") ) {
            Bank.AccountManagerHelper.addServerObjectWrapperClass(orb, c);
 } else {
            Bank.AccountManagerHelper.addClientObjectWrapperClass(orb, c);
```

```
               Bank.AccountManagerHelper.addServerObjectWrapperClass(orb, c);
            }
         }
         // CachingAccountManagerObjectWrapper をインストールします。
         val = pm.getString("SecureAccountManager",
               this.toString());
         c = SecureAccountManagerObjectWrapper.class;
         if( !val.equals(this.toString())) {
            if( val.equalsIgnoreCase("client") ){
               Bank.AccountManagerHelper.addClientObjectWrapperClass(orb, c);
            } else if( val.equalsIgnoreCase("server") ) {
               Bank.AccountManagerHelper.addServerObjectWrapperClass(orb, c);
            } else {
               Bank.AccountManagerHelper.addClientObjectWrapperClass(orb, c);
               Bank.AccountManagerHelper.addServerObjectWrapperClass(orb, c);
            }
         }
      }
      public void init_complete(org.omg.CORBA.ORB orb) {}
      public void shutdown(org.omg.CORBA.ORB orb) {}
   }
```

型なしラッパーのコマンドライン引数

型なしラッパーを有効にするには，コマンドラインで次の引数を指定します。

1 -Dvbroker.orb.dynamicLibs=UtilityObjectWrappers.Init

2 次の表に示されている 1 つ以上のプロパティを使用します。

表 26.4　UtilityObjectWrappers を有効／無効にするコマンドラインプロパティ

UtilityObjectWrappers のプロパティ	説明
-DTiming[=<client|server>]	クライアントまたはサーバーの処理時間を測定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。サブプロパティに値を指定しなければ，クライアントのラッパーとサーバーのラッパーの両方がインストールされます。
-DTracing[=<client|server>]	クライアントまたはサーバーの情報を追跡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。サブプロパティに値を指定しなければ，クライアントのラッパーとサーバーのラッパーの両方がインストールされます。

型なしラッパーのイニシャライザ

型なしラッパーは，UtilityObjectWrappers パッケージで定義され，次に示すサービスイニシャライザ UtilityObjectWrappers/Init.java を組み込みます。コマンドラインに -Dvbroker.orb.dynamicLibs=UtilityObjectWrappers.Init を指定しておくと，vbj でクライアントやサーバーを起動するときにこのイニシャライザが呼び出されます。374 ページの表 26.4「UtilityObjectWrappers を有効／無効にするコマンドラインプロパティ」には，さまざまな型なしオブジェクトラッパーのインストールに使用できるコマンドライン引数がまとめられています。

```
package UtilityObjectWrappers;
import java.util.*;
Import com.inprise.vbroker.orb.ORB;
import com.inprise.vbroker.properties.PropertyManager;
import com.inprise.vbroker.interceptor.*;
Public class Init implements ServiceLoader {
   com.inprise.vbroker.orb.ORB _orb;
   public void init(final org.omg.CORBA.ORB orb) {
      _orb = (ORB) orb;
      PropertyManager PM= _orb.getPropertyManager();
```

```
      try {
   ChainUntypedObjectWrapperFactory factory =
               ChainUntypedObjectWrapperFactoryHelper.narrow(

  orb.resolve_initial_references("ChainUntypedObjectWrapperFactory"));
         // 時間測定の ObjectWrapper をインストールします。
   String val = pm.getString("Timing", this.toString());
         if( !val.equals(this.toString())) {
            UntypedObjectWrapperFactory f= new
                              TimingUntypedObjectWrapperFactory();
     if( val.equalsIgnoreCase("client") ){
               factory.add(f, Location.CLIENT);
     } else if( val.equalsIgnoreCase("server") ) {
               factory.add(f, Location.SERVER);
     } else {
               factory.add(f, Location.BOTH);
            }
         }

         // トレースの ObjectWrapper をインストールします。
   val = pm.getString("Tracing", this.toString());
         if( !val.equals(this.toString())) {
            UntypedObjectWrapperFactory f= new
  TracingUntypedObjectWrapperFactory();
     if( val.equalsIgnoreCase("client") ){
               factory.add(f, Location.CLIENT);
     } else if( val.equalsIgnoreCase("server") ) {
               factory.add(f, Location.SERVER);
     } else {
               factory.add(f, Location.BOTH);
            }
         }
      } catch( org.omg.CORBA.ORBPackage.InvalidName e ) {
         return;
      }
   }
   public void init_complete(org.omg.CORBA.ORB orb) {}
   public void shutdown(org.omg.CORBA.ORB orb) {}
}
```

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> vbj 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> vbj 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> vbj 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> vbj -Dvbroker.orb.dynamicLibs=UtilityObjectWrappers.Init
         -DTiming=client¥
         -DTracing=client Client John
```

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> vbj -Dvbroker.orb.dynamicLibs=UtilityObjectWrappers.Init
         -DTiming=server¥
         -DTracing=server Server
```

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> vbj -Dvbroker.orb.dynamicLibs=BankWrappers.Init -
DCachingAccount=client¥
         -DCachingAccountManager=client¥
         -DSecureAccountManager=client
         Client John
```

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> vbj -Dvbroker.orb.dynamicLibs=BankWrappers.Init
         -DCachingAccount=server ¥
         -DCachingAccountManager=server ¥
         -DSecureAccountManager=server ¥
         Server
```

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> vbj -DOvbroker.orb.dynamicLibs=BankWrappers.Init,
               UtilityObjectWrappers.Init ¥
         -DCachingAccount=client ¥
         -DCachingAccountManager=client¥
         -DSecureAccountManager=client ¥
         -DTiming=client ¥
         -DTracing=client ¥
         Client John
```

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> vbj -Dvbroker.orb.dynamicLibs=BankWrappers.Init,
            UtilityObjectWrappers.Init ¥
         -DCachingAccount=server ¥
         -DCachingAccountManager=server¥
         -DSecureAccountManager=server ¥
         -DTiming=server ¥
         -DTracing=server ¥
         Server
```

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> vbj -Dvbroker.orb.dynamicLibs=BankWrappers.Init,
               UtilityObjectWrappers.Init ¥
         -DCachingAccount -DSecureAccountManager ¥
         -DTiming=client -DTracing=server ¥
         Server -runCoLocated Client
```

-runCoLocated コマンドラインオプションを指定すると，同じプロセスでクライアントとサーバーを実行できます。

プロパティ	説明
-runCoLocated Client	Server.java と Client.java を同じプロセス内で実行します。
-runCoLocated TypedClient	Server.java と TypedClient.java を同じプロセス内で実行します。
-runCoLocated UntypedClient	Server.java と UntypedClient.java を同じプロセス内で実行します。

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。

```
com.inprise.vbroker.interceptor.InterceptorManagerControl control =
  com.inprise.vbroker.interceptor.InterceptorManagerControlHelper.narrow(
    orb.resolve_initial_references("VisiBrokerInterceptorControl"));
EventQueueManager manager =
  (EventQueueManager)control.get_manager("EventQueue");
EventListener theListener = ...
manager.register_listeners(theListener);
```

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。

```
import com.inprise.vbroker.EventQueue.*;
import com.inprise.vbroker.interceptor.*;
import com.inprise.vbroker.PortableServerExt.*;
public class SampleServerLoader implements ServiceLoader {
  public void init(org.omg.CORBA.ORB orb) {
    try {
      InterceptorManagerControl control =
        InterceptorManagerControlHelper.narrow(
          orb.resolve_initial_references("VisiBrokerInterceptorControl"));
      EventQueueManager queue_manager =
        (EventQueueManager) control.get_manager("EventQueue");
      queue_manager.register_listener((EventListener)new
        ConnEventListenerImpl(),EventType.CONN_EVENT_TYPE);
    }
    catch(Exception e) {
      e.printStackTrace();
      throw new org.omg.CORBA.INITIALIZE(e.toString());
    }
    System.out.println("============>SampleServerLoader: ConnEventListener
            registered");
  }
  public void init_complete(org.omg.CORBA.ORB orb) {
```

```
  }
  public void shutdown(org.omg.CORBA.ORB orb) {
  }
}
```

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，380 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。

```
import com.inprise.vbroker.EventQueue.*;
import org.omg.CORBA.LocalObject;

public class ConnEventListenerImpl extends LocalObject implements
ConnEventListener {
  public void conn_established(ConnInfo info) {
    System.out.println("Received conn_established: address = " +
                       info.ipaddress + " port = " + info.port +
                       " connID = " + info.connID);
    System.out.println("Processing the event ...");
    try {
      Thread.sleep(30000);
    } catch (Exception e) { e.printStackTrace(); }
  }
  public void conn_closed(ConnInfo info) {
    System.out.println("Received conn_closed: address = " +
                       info.ipaddress+ " port = " + info.port +
                       " connID = " + info.connID);
  }
}
```

第 28 章

IIOP を介した RMI の使い方

この章では，RMI over IIOP を使用するための Java ツールについて説明します。また，RMI-IIOP を使用する Java アプレットの実行に必要な権限のセットアップについても簡単に説明します。

RMI over IIOP の概要

RMI（リモートメソッド呼び出し）は，オブジェクトを分散環境で作成および使用するための Java メカニズムです。この点で，RMI は Java 言語固有で，CORBA に準拠しない VisiBroker ORB であると言えます。OMG は Java 言語と IDL のマッピング仕様を発表しています。この仕様により，RMI で記述された Java クラスと CORBA オブジェクトを，IIOP エンコーディングで相互運用できます。

RMI-IIOP を使用する Java アプレットの設定

RMI-IIOP を使用するアプレットを実行できます。ただし，Reflect と Runtime で権限を設定する必要があります。これらの権限は，JRE　インストールディレクトリにある java.policy ファイルで設定します。java.policy ファイルでこれらの権限を設定する例を次に示します。

```
grant codeBase "http://xxx.xxx.xxx.xxx:8088/-" {
permission java.lang.reflect.ReflectPermission "suppressAccessChecks";

permission java.lang.RuntimePermission "accessDeclaredMembers";
};
```

java2iiop と java2idl

VisiBroker には，VisiBroker ORB を使用するほかのオブジェクトと同時に既存の Java クラスを使用できるコンパイラが 2 つ用意されています。

- java2iiop コンパイラでは，RMI 準拠のクラスで IIOP を使用するため，適切なスケルトン，スタブ，およびヘルパークラスを生成します。
- java2idl コンパイラは，Java クラスから IDL を生成します。この IDL は，Java 以外の言語でも実装できます。

java2iiop の使い方

java2iiop コンパイラを使用すると，IDL のかわりに Java でインターフェースとデータ型を定義し，そのインターフェースとデータ型を CORBA で使用できます。このコンパイラは，Java ソースコード（java ファイル）や IDL のかわりに Java bytecode（class ファイル）を読み取ります。そして，CORBA に必要なマーシャリングと通信のすべてを実行する IIOP 準拠のスタブとスケルトンを生成します。

サポートされるインターフェース

java2iiop コンパイラを実行すると，IDL でインターフェースを記述した場合と同じファイルが生成されます。数値型（short，int，long，float，double）などのすべてのプリミティブデータ型，文字列，CORBA オブジェクト，CORBA インターフェースオブジェクト，Any オブジェクト，typecode オブジェクトは，java2iiop コンパイラによって認識され，対応する IDL 型にマッピングされます。

java2iiop は，任意の Java クラスまたはインターフェースに対して使用できます。たとえば，Java インターフェースが次のどちらかの規則にしたがっているとします。

- java.rmi.Remote を拡張し，そのすべてのメソッドが java.rmi.RemoteException を生成する。
- org.omg.CORBA.Object を拡張する。

java2iiop は，このインターフェースを IDL の CORBA インターフェースに変換します。

次のサンプルコードは，Java RMI インターフェースです。このサンプルコードは次のサイトから入手できます。

```
<install_dir>/vbe/examples/rmi-iiop/
```

```
public interface Account extends java.rmi.Remote {
   String name() throws java.rmi.RemoteException;
   float getBalance() throws java.rmi.RemoteException;
   void setBalance(float bal) throws java.rmi.RemoteException;
}
```

java2iiop の実行

java2iiop コンパイラを使用する前に，Java クラスをコンパイルする必要があります。バイトコードを生成した後で，java2iiop を実行して，クライアントスタブ，サーバースケルトン，および関連する補助ファイルを生成します。

たとえば，次の場所にある Account.class ファイルに対して java2iiop を実行します。

```
<install_dir>/vbe/examples/rmi-iiop/Bank/
```

これで，次のファイルが生成されます。

- _Account_Stub
- AccountHelper
- AccountHolder
- AccountPOA
- Account_Tie
- AccountOperations

Java クラスから IDL への逆マッピング

idl2java コンパイラを使って IDL インターフェースを Java クラスにマッピングする場合，生成されたクラスを示すサフィックス（Helper，Holder，POA など）がインターフェー

ス名に付きます。次のような状況を正しく処理するため，idl2java ツールは，識別子の前にアンダースコアを付けてインターフェース名を変更します。

たとえば，IDL で Foo と FooHolder インターフェースの両方を定義した場合，idl2java は，Foo.java，FooHolder.java，_FooHolder.java，および _FooHolderHolder.java ファイルを生成します。

一方，java2iiop コンパイラで RMI Java クラスから IIOP 準拠の Java クラスを生成する場合は，名前を変更してクラスを生成することができません。

したがって，予約されたサフィックスを使用するインターフェースを宣言すると，同じ名前のインターフェースとして同じパッケージにまとめることができなくなります。たとえば，java2iiop コンパイラの使用時には，Foo と FooHolder クラスを同じパッケージにまとめることはできません。

開発の完了

インターフェースから関連するファイルを生成したら，これらの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。次の手順にしたがってください。

1. インターフェースクラスのインプリメンテーションを作成します。
2. サーバークラスをコンパイルします。
3. クライアントコードを記述し，コンパイルします。
4. サーバープログラムを起動します。
5. クライアントプログラムを実行します。

メモ　非準拠クラスのマーシャリングを試みると，org.omg.CORBA.MARSHAL: Cannot marshal non-conforming value of class <class name> という例外が生成されます。たとえば，次の 2 つのクラスを作成したとします。

```
// 準拠クラス
public class Value implements java.io.Serializable {
   java.lang.Object any;
   ...
}
// 非準拠クラス
public class Something {
...
}
```

ここで，次の代入を試みます。

```
Value val = new Value();
val.any = new Something();
```

val をマーシャリングしようとすると，org.omg.CORBA.MARSHAL 例外が発生します。

RMI-IIOP Bank サンプル

このコードサンプルは，次のディレクトリにあります。

```
<install_dir>/vbe/examples/rmi-iiop/
```

Account インターフェースは，java.rmi.Remote インターフェースを拡張し，AccountImpl クラスによって実装されます。

まず Client クラス（386 ページの「Client クラス：」を参照）によって，適切な残高の指定 Account オブジェクトが作成されます。具体的には，口座ごとに AccountData オブジェクトを作成し，それらのオブジェクトを AccountManager に渡して口座を作成します。次に，作成された口座の残高が正しいかどうかを確認します。次に，AccountManager にすべての口座のリストを照会し，各口座に 10 ドルずつ振り込みます。最後に，各口座の新しい残高が正しいかどうかを確認します。

```
public class AccountImpl extends Bank.AccountPOA {
   public AccountImpl(Bank.AccountData data) {
      _name = data.getName();
      _balance = data.getBalance();
   }
   public String name() throws java.rmi.RemoteException {
      return _name;
   }
   public float getBalance() throws java.rmi.RemoteException {
      return _balance;
   }
   public void setBalance(float balance) throws java.rmi.RemoteException {
      _balance = balance;
   }
   private float _balance;
   private String _name;
}
```

Client クラス：

```
public class Client {
   public static void main(String[] args) {
      try {
         // ORB を初期化します。
         org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
         // マネージャの ID を取得します。
         byte[] managerId = "RMIBankManager".getBytes();
         // 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 manager =
               Bank.AccountManagerHelper.bind(orb, "/rmi_bank_poa", managerId);
         // 任意の数の引数を組で使用して，作成する口座の名前と残高を
指定します。
         if (args.length == 0 || args.length % 2 != 0) {
            args = new String[2];
            args[0] = "Jack B. Quick";
            args[1] = "123.23";
         }
         int i = 0;
         while (i < args.length) {
            String name = args[i++];
            float balance;
            try {
               balance = new Float(args[i++]).floatValue();
            } catch (NumberFormatException n) {
               balance = 0;
            }
            Bank.AccountData data = new Bank.AccountData(name, balance);
            Bank.Account account = manager.create(data);
            System.out.println("Created account for " + name

                  + " with opening balance of $" + balance);
         }
         java.util.Hashtable accounts = manager.getAccounts();
         for (java.util.Enumeration e = accounts.elements();
e.hasMoreElements();) {
            Bank.Account account =

Bank.AccountHelper.narrow((org.omg.CORBA.Object)e.nextElement());
            String name = account.name();
            float balance = account.getBalance();
            System.out.println("Current balance in " + name + "'s account is $"
+ balance);
            System.out.println("Crediting $10 to " + name + "'s account.");
            account.setBalance(balance + (float)10.0);
            balance = account.getBalance();
            System.out.println("New balance in " + name + "'s account is $" +
```

```
balance);
         }
      } catch (java.rmi.RemoteException e) {
         System.err.println(e);
      }
   }
}
```

サポートされるデータ型

RMI-IIOP は，すべての Java プリミティブデータ型のほか，Java クラスのサブセットもサポートします。

プリミティブデータ型のマッピング

java2iiop によって生成されたクライアントスタブは，オペレーションリクエストを表現する Java プリミティブデータ型をマーシャリングし，それらをオブジェクトサーバーに転送できるようにします。Java プリミティブデータ型は，マーシャリングされるとき，IIOP 準拠の型に変換されます。次の表は，Java プリミティブデータ型と IDL/IIOP 型の対応を示しています。

表 28.1　IDL/IIOP への Java 型のマッピング

Java 型	IDL/IIOP 型
void	void
boolean	boolean
byte	octet
char	char
short	short
int	long
long	long long
float	float
double	double
java.lang.String	CORBA::WStringValue
java.lang.Object	any
java.io.Serializable	any
java.io.Externalizable	any

複合データ型のマッピング

ここでは，java2iiop コンパイラを使って複合データ型を処理する方法を示します。

インターフェース

Java インターフェースは，IDL では CORBA インターフェースで表され，org.omg.CORBA.Object インターフェースを継承します。これらのインターフェースを実装するオブジェクトを指定する場合は，そのリファレンスで指定します。

配列

クラス内で定義されるその他の複合データ型として，配列があります。配列を使用するインターフェースまたはクラス定義がある場合，それらの配列は，ボックス化された型のボックス化された CORBA シーケンスにマッピングされます。

第 29 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。第 4 章「Java 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 29.1 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。

```
// 動的 Any ファクトリを解決します。
DynAnyFactory factory =
    DynAnyFactoryHelper.narrow(
        orb.resolve_initial_references("DynAnyFactory"));
byte[] oid = "PrinterManager".getBytes();
// PrinterManager オブジェクトを作成します。
PrinterManagerImpl manager =
    new PrinterManagerImpl((com.borland.vbroker.CORBA.ORB) orb,
        factory, serverPoa, oid);
// 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(oid, manager);
System.out.println(manager + " is ready.");
```

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。392 ページの「DynAny サンプルクライアントアプリケーション」と 393 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.java
import org.omg.DynamicAny.*;
public class Client {
  public static void main(String[] args) {
    try {
      // ORB を初期化します。
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
      DynAnyFactory factory =
          DynAnyFactoryHelper.narrow(
              orb.resolve_initial_references("DynAnyFactory"));
      // PrinterManager を検索します。
      Printer.PrinterManager manager =
          Printer.PrinterManagerHelper.bind(orb, "PrinterManager");
      // 動的構造体を作成します。
      DynStruct info =
          DynStructHelper.narrow(factory.create_dyn_any_from_type_code
              (Printer.StructTypeHelper.type()));
      // NameValuePair のシーケンス（配列）を作成します。
      NameValuePair[] NVPair = new NameValuePair[3];
      // 動的構造体のデータを ANY として作成および初期化します。
      org.omg.CORBA.Any str_any = orb.create_any();
      str_any.insert_string("String");
      org.omg.CORBA.Any e_any = orb.create_any();
      Printer.EnumTypeHelper.insert(e_any, Printer.EnumType.second);
      org.omg.CORBA.Any fl_any = orb.create_any();
      fl_any.insert_float((float)864.50);
      NVPair[0] = new NameValuePair("str", str_any);
      NVPair[1] = new NameValuePair("e", e_any);
      NVPair[2] = new NameValuePair("fl", fl_any);
      // 動的構造体を初期化します。
      info.set_members(NVPair);
      manager.printAny(info.to_any());
      manager.shutdown();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

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。

```
// Server.java
import java.util.*;
import org.omg.DynamicAny.*;
import org.omg.PortableServer.*;
import com.inprise.vbroker.PortableServerExt.*;
public class Server {
  public static void main(String[] args) {
    try {
      // ORB を初期化します。
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
      // ルート POA を解決します。
      POA rootPoa =
          POAHelper.narrow(orb.resolve_initial_references(
              "RootPOA"));
      rootPoa.the_POAManager().activate();
      // POA が各サーバントを osagent に登録するように
      // BindSupport ポリシーを作成します。
      org.omg.CORBA.Any any = orb.create_any();
      BindSupportPolicyValueHelper.insert(any,
           BindSupportPolicyValue.BY_INSTANCE);
      org.omg.CORBA.Policy bsPolicy =
           orb.create_policy(BIND_SUPPORT_POLICY_TYPE.value, any);
      // testPOA のポリシーを作成します。
      org.omg.CORBA.Policy[] policies = {
           rootPoa.create_lifespan_policy(LifespanPolicyValue.PERSISTENT),
           bsPolicy
      };
      // 適切なポリシーで managerPOA を作成します。
      POA serverPoa =
           rootPoa.create_POA(
               "serverPoa",
               rootPoa.the_POAManager(),
               policies );
      // 動的 Any ファクトリを解決します。
      DynAnyFactory factory =
               DynAnyFactoryHelper.narrow(
                   orb.resolve_initial_references("DynAnyFactory"));
      byte[] oid = "PrinterManager".getBytes();
      // PrinterManager オブジェクトを作成します。
      PrinterManagerImpl manager =
          new PrinterManagerImpl((
              com.borland.vbroker.CORBA.ORB) orb,
              factory,
              serverPoa,
              oid);
      // 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(oid, manager);
      System.out.println(manager + " is ready.");
      // 着信要求を待機します。
      orb.run();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

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。

```
// PrinterManagerImpl.java
import java.util.*;
import org.omg.DynamicAny.*;
import org.omg.PortableServer.*;
public class PrinterManagerImpl extends Printer.PrinterManagerPOA {
  private com.borland.vbroker.CORBA.ORB _orb;
  private DynAnyFactory _factory;
  private POA _poa;
  private byte[] _oid;
  public PrinterManagerImpl(com.borland.vbroker.CORBA.ORB orb,
      DynAnyFactory factory, POA poa, byte[] oid) {
    _orb = orb;
    _factory = factory;
    _poa = poa;
    _oid = oid;
  }
  public synchronized void printAny(org.omg.CORBA.Any info) {
    // Any の内部の型について確実な情報がない場合に，
    // info の内容を表示します。
    try {
      // DynAny オブジェクトを作成します。
      DynAny dynAny = _factory.create_dyn_any(info);
      display(dynAny);
    }
    catch (Exception e) {
      e.printStackTrace();
    }

  }
  public void shutdown() {
    try {
      _poa.deactivate_object(_oid);
      System.out.println("Server shutting down");
      _orb.shutdown(false);
    }
    catch (Exception e) {
      System.out.println(e);
    }
  }
  private void display(DynAny value) throws Exception {
    switch(value.type().kind().value()) {
      case org.omg.CORBA.TCKind._tk_null:
      case org.omg.CORBA.TCKind._tk_void: {
       break;
      }
      case org.omg.CORBA.TCKind._tk_short: {
        System.out.println(value.get_short());
        break;
      }
      case org.omg.CORBA.TCKind._tk_ushort: {
        System.out.println(value.get_ushort());
        break;
      }
      case org.omg.CORBA.TCKind._tk_long: {
        System.out.println(value.get_long());
        break;
      }
```

```
case org.omg.CORBA.TCKind._tk_ulong: {
  System.out.println(value.get_ulong());
  break;
}
case org.omg.CORBA.TCKind._tk_float: {
  System.out.println(value.get_float());
  break;
}
case org.omg.CORBA.TCKind._tk_double: {
  System.out.println(value.get_double());
  break;
}
case org.omg.CORBA.TCKind._tk_boolean: {
  System.out.println(value.get_boolean());
  break;
}
case org.omg.CORBA.TCKind._tk_char: {
  System.out.println(value.get_char());
  break;
}
case org.omg.CORBA.TCKind._tk_octet: {
  System.out.println(value.get_octet());
  break;
}
case org.omg.CORBA.TCKind._tk_string: {
  System.out.println(value.get_string());
  break;
}
case org.omg.CORBA.TCKind._tk_any: {
  DynAny dynAny = _factory.create_dyn_any(value.get_any());
  display(dynAny);
  break;
}
case org.omg.CORBA.TCKind._tk_TypeCode: {
  System.out.println(value.get_typecode());
  break;
}
case org.omg.CORBA.TCKind._tk_objref: {
  System.out.println(value.get_reference());
  break;
}
case org.omg.CORBA.TCKind._tk_enum: {
  DynEnum dynEnum = DynEnumHelper.narrow(value);
  System.out.println(dynEnum.get_as_string());
  break;
}
case org.omg.CORBA.TCKind._tk_union: {
  DynUnion dynUnion = DynUnionHelper.narrow(value);
  display(dynUnion.get_discriminator());
  display(dynUnion.member());
  break;
}
case org.omg.CORBA.TCKind._tk_struct:
case org.omg.CORBA.TCKind._tk_array:
case org.omg.CORBA.TCKind._tk_sequence: {
  value.rewind();
  boolean next = true;
  while(next) {
    DynAny d = value.current_component();
    display(d);
    next = value.next();
  }
  break;
}
case org.omg.CORBA.TCKind._tk_longlong: {
  System.out.println(value.get_longlong());
```

```
        break;
      }
      case org.omg.CORBA.TCKind._tk_ulonglong: {
        System.out.println(value.get_ulonglong());
        break;
      }
      case org.omg.CORBA.TCKind._tk_wstring: {
        System.out.println(value.get_wstring());
        break;
      }
      case org.omg.CORBA.TCKind._tk_wchar: {
        System.out.println(value.get_wchar());
        break;
      }
      default:
        System.out.println("Invalid type");

    }
  }
}
```

第 30 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，403 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，402 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. `idl2java` 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. 必要に応じて，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（399 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，`idl2java` 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。

399 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。

- `Point.java`
- `PointDefaultFactory.java`
- `PointHelper.java`
- `PointHolder.java`
- `PointValueFactory.java`

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

```
public class PointImpl extends Point {
   public PointImpl() {}
   public PointImpl(int a_x, int a_y, String a_label) {
      x = a_x;
      y = a_y;
      label = a_label;
   }


   public void print () {
      System.out.println("Point is [" + label + ": (" + x + ", " + y + ")]");
   }
}
```

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，org.omg.CORBA.portable.ValueFactory 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。

```
public class PointDefaultFactory implements PointValueFactory {
   public java.io.Serializable read_value (org.omg.CORBA.portable.InputStream
is) {
       java.io.Serializable val = new PointImpl(); // インプリメンテーション
                                                   クラスと呼ばれます。
// 値を作成および初期化します。
       val = ((org.omg.CORBA_2_3.portable.InputStream)is).read_value(val);
       return val;
   }
   // valuetype を実装するのはユーザーの役割です
   public Point create (int x,
         int y,
         java.lang.String z) {
       // 実装
       return null;
   }
}
```

新しい valuetype を作成する PointImpl() が呼び出されます。これは，read_value によって InputStream から呼び出されます。

read_value を呼び出さなければファクトリは機能せず，その他のメソッドを呼び出すことができません。

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。valuetypenameDefaultFactory の形式でファクトリ名を指定している場合，この処理は不要です。ファクトリの登録の詳細については，403 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。

```
public class PointDefaultFactory implements PointValueFactory {
   public java.io.Serializable read_value (
          org.omg.CORBA_2_3.portable.InputStream is) {
```

```
        java.io.Serializable val = new PointImpl();
        // 値を作成および初期化します。
        // この呼び出しはたいへん重要です。
        val = ((org.omg.CORBA_2_3.portable.InputStream)is).read_value(val);
        return val;
        }
        public Point create (int x, int y, java.lang.String z) {
            // 実装
        return NO_IMPLEMENT;
        }
    }
```

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 org.omg.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，org.omg.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 `org.omg.CORBA.portable.CustomValue` を拡張します。標準の valuetype が拡張する `org.omg.CORBA.portable.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。

`org.omg.CORBA.portable.DataInputStream` と `org.omg.CORBA.portable.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 31 章

URL ネーミングの使い方

ここでは，ユニフォームリソースロケータ（Uniform Resource Locator : URL）を相互運用可能なオブジェクトリファレンス（Interoperable Object Reference : IOR）に関連付ける URL ネーミングサービスの使用方法について説明します。URL がオブジェクトにバインドされると，クライアントアプリケーションは，オブジェクト名のかわりに URL 文字列を指定してそのオブジェクトへのリファレンスを取得できます。クライアントアプリケーションで，osagent や CORBA ネーミングサービスを使用しないでオブジェクトを探す場合は，URL を指定する方法を利用します。

URL ネーミングサービス

URL ネーミングサービスは，サーバーオブジェクトの IOR をファイル内の文字列形式の URL に関連付ける単純なメカニズムです。クライアントプログラムは，Web サーバー上にあるこのファイルを指す URL を使ってオブジェクトを探すことができます。URL ネーミングサービスは，URL に基づいてオブジェクトを登録および検索する方法として，`http` URL スキームをサポートします。

URL ネーミングサービスでは，スマートエージェントや CORBA ネーミングサービスを使用しないでオブジェクトを探すことができます。また，クライアントアプリケーションは，どのベンダーから提供されるオブジェクトでも探すことができます。

メモ　VisiBroker の URL ネーミングサービスは，使用している Java 環境がサポートするすべての形式の URL 処理をサポートします。

URL ネーミングサービスの例

URL ネーミングサービスのサンプルコードは，VisiBroker の次のディレクトリにあります。

```
<install_dir>¥examples¥vbe¥basic¥bank_URL
```

次に示すのは，このサービスの IDL 仕様です。

```
// WebNaming.idl
#pragma prefix "borland.com"
module URLNaming {
   exception InvalidURL{string reason;};
   exception CommFailure{string reason;};
   exception ReqFailure{string reason;};
```

```
    exception AlreadyExists{string reason;};
    abstract interface Resolver {
    // 読み取りオペレーション
       Object locate(in string url_s)
             raises (InvalidURL, CommFailure, ReqFailure);
    // 書き込みオペレーション
       void force_register_url(in string url_s, in Object obj)
             raises (InvalidURL, CommFailure, ReqFailure);
       void register_url(in string url_s, in Object obj)
             raises (InvalidURL, CommFailure, ReqFailure, AlreadyExists);
      };
};
```

オブジェクトの登録

オブジェクトサーバーは，Resolver にバインドし，register_url メソッドまたは force_register_url メソッドで URL をオブジェクトの IOR に関連付けてオブジェクトを登録します。以前にそのオブジェクトの関連付けが行われていない場合は，register_url メソッドで URL をオブジェクトの IOR に関連付けます。force_register_url メソッドを使用すると，URL がオブジェクトにバインドされているかどうかに関係なく，URL がオブジェクトの IOR に関連付けられます。同じ状況で register_url メソッドを使用すると，AlreadyExists 例外が発生します。

サーバー側でこの機能を使用する例については，407 ページの「URL ネーミングサービスの例」を参照してください。この例では，force_register_url を使用します。force_register_url が正しく機能するには，Web サーバーで HTTP PUT コマンドを発行できる環境が必要です。

メモ Resolver へのリファレンスを取得するには，次のサンプルに示すように，ORB の resolve_initial_references メソッドを使用します。

```
. . .
public class Server {
   public static void main(String[] args) {
      if (args.length == 0) {
         System.out.println("Usage: vbj Server <URL string>");
         return;
      }
      String url = args[0];
      try {
         // ORB を初期化します。
         org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
         // ルート POA へのリファレンスを取得します。
         POA rootPOA =
POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
         // サーバントを作成します。
         AccountManagerImpl managerServant = new AccountManagerImpl();
         // サーバントの ID を決定します。
         byte[] managerId = "BankManager".getBytes();
         // その ID を使って myPOA でサーバントをアクティブ化します。
         rootPOA.activate_object_with_id(managerId, managerServant);
         // POA マネージャをアクティブ化
         rootPOA.the_POAManager().activate();
         // オブジェクトリファレンスを作成します。
         org.omg.CORBA.Object manager =
              rootPOA.servant_to_reference(managerServant);
         // URLNaming リゾルバを取得します。
         Resolver resolver = ResolverHelper.narrow(
              orb.resolve_initial_references("URLNamingResolver"));
         // オブジェクトリファレンスを登録します（既存のものは上書きされます）
         resolver.force_register_url(url, manager);
         System.out.println(manager + " is ready.");
         // 着信要求を待機します。
```

```
            orb.run();
         } catch (Exception e) {
            e.printStackTrace();
         }
      }
   }
```

このサンプルコードで，args[0] の形式は次のとおりです。

```
http://<host_name>:<http_server_port>/<ior_file_path>/<ior_file_name>
```

ior_file_name は，文字列化されたオブジェクトリファレンスが保存されているユーザー指定のファイル名です。HTTP サーバーのかわりに Gatekeeper を使用する場合は，ior_file_name の末尾を .ior とします。次に，デフォルトのポート番号を指定して Gatekeeper を使用するサンプルを示します。

```
http://mars:15000/URLNaming/Bank_Manager.ior
```

URL によるオブジェクトの検索

次のサンプルコードでわかるように，クライアントアプリケーションでは Resolver にバインドする必要はなく，クライアントアプリケーションは，bind メソッドの呼び出し時に URL を指定するだけです。bind はオブジェクト名として URL を受け取ります。URL が有効でない場合は，InvalidURL 例外が生成されます。bind メソッドは，ユーザーにかわって透過的に locate() を呼び出します。

```
// ResolverClient.java
import com.borland.vbroker.URLNaming.*;
public class ResolverClient {
   public static void main(String[] args) {
      if (args.length == 0) {
         System.out.println("Usage: vbj Client <URL string> [Account name]");
         return;
      }
      String url = args[0];
      try {
         // ORB を初期化します。
         org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
```

locate() の使用方法については，次のサンプルコードを参照してください。

```
         // URLNaming リゾルバを取得します。
         Resolver resolver = ResolverHelper.narrow(
         orb.resolve_initial_references("URLNamingResolver"));
         // オブジェクトを検索します。
         Bank.AccountManager manager =
         Bank.AccountManagerHelper.narrow(resolver.locate(url));
         // 口座名またはデフォルトとして args[0] を使用します。
         String name = args.length > 1 ? args[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 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。
         System.out.println("The balance in " + name + "'s account is $" +
balance);
      } catch(Exception e) {
         e.printStackTrace();
      }
   }
}
```

Resolver.locate メソッドによるオブジェクトリファレンスの取得：

```
// Client.java
public class Client {
```

```
   public static void main(String[] args) {
      if (args.length == 0) {
         System.out.println("Usage: vbj Client <URL string> [Account name]");
         return;
      }
      String url = args[0];
      // ORB を初期化します。
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
      // オブジェクトを検索します。
      Bank.AccountManager manager = Bank.AccountManagerHelper.bind(orb, url);
      // 口座名またはデフォルトとして args[0] を使用します。
      String name = args.length > 1 ? args[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 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。
      System.out.println("The balance in " + name + "'s account is $" +
balance);
   }
}
```

第 32 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 ガイド』を参照してください。

メモ　双方向 IIOP を有効にする前に，414 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。

- `enableBiDir` プロパティ（412 ページ参照）

- exportBiDi プロパティ（412 ページ参照）
- importBiDir プロパティ（412 ページ参照）

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 32.1　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）。

exportBiDi 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> vbj -Dvbroker.orb.enableBiDir=server Server &
```

Windows
```
prompt> start vbj -Dvbroker.orb.enableBiDir=server Server
```

3 クライアントを起動します。

```
prompt> vbj -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，413 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ています。

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

try {
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, null);
    org.omg.PortableServer.POA rootPOA =
        org.omg.PortableServer.POAHelper.narrow(
            orb.resolve_initial_references("RootPOA"));
    org.omg.CORBA.Any bidirPolicy = orb.create_any();
    bidirPolicy.insert_short(BOTH.value);
    org.omg.CORBA.Policy[] policies = {
        //Bidir ポリシーを設定します。
        orb.create_policy(BIDIRECTIONAL_POLICY_TYPE.value, bidirPolicy)
    };
    org.omg.PortableServer.POA callbackPOA =
```

```
        rootPOA.create_POA("bidir", rootPOA.the_POAManager(), policies);
    QuoteConsumerImpl c = new QuoteConsumerImpl();
    callbackPOA.activate_object(c);
    callbackPOA.the_POAManager().activate();
    QuoteServer serv =
        QuoteServerHelper.bind(orb, "/QuoteServer_poa",
                "QuoteServer".getBytes());
    serv=QuoteServerHelper.narrow(serv._set_policy_override(
                policies, org.omg.CORBA.SetOverrideType.ADD_OVERRIDE));
    serv.registerConsumer(QuoteConsumerHelper.narrow(
                callbackPOA.servant_to_reference(c)));
    System.out.println("Client: consumer registered");
    //60 秒スリープし，メッセージを受信します。
    try{
        Thread.currentThread().sleep(60*1000);
    }
    catch(java.lang.InterruptedException e){ }
    serv.unregisterConsumer(QuoteConsumerHelper.narrow(
        callbackPOA.servant_to_reference(c)));
    System.out.println("Client: consumer unregistered.Good bye.");
    orb.shutdown(true);
...
```

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> vbj -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，411 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 33 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 2.1) 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 BOA コードがあり，それを最新バージョンの VisiBroker でも使用する場合は，次の点に注意する必要があります。

- 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 idl2java ツールを使用する必要があります。idl2java 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 Java へのマッピング」を参照してください。
- BOA_init() は org.omg.CORBA.ORB の下では使用できなくなっているため，VisiBroker ORB を com.borland.vbroker.CORBA.ORB にキャストする必要があります。
- BOA クラスは，org.omg.CORBA パッケージではなく com.borland.vbroker.CORBA パッケージで提供されるようになりました。VisiBroker ORB パッケージの詳細については，Java API リファレンスの VisiBroker API 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。

BOA を使用する場合の制限事項

VisiBroker 4.x の BOA では，次の 2 つの機能がサポートされていません。

- 永続的 DSI オブジェクト
- DSI オブジェクトの _boa()

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 Java 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 34 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 33 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 DBObjectImpl オブジェクトに対して使用する activate メソッドと deactivate メソッドを実装できます。これで，BOA が DBObjectImpl 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。

このサンプルコードは Activator インターフェースを示します。BOA によって呼び出されるこれらのメソッドは，VisiBroker ORB オブジェクトをアクティブ化，非アクティブ化します。

```
package com.borland.vbroker.extension;
public interface Activator {
   public abstract org.omg.CORBA.Object activate(ImplementationDef impl);
   public abstract void deactivate(org.omg.CORBA.Object obj, ImplementationDef
impl);
}
```

次のサンプルコードは，DBObjectImpl インターフェースの Activator を作成します。

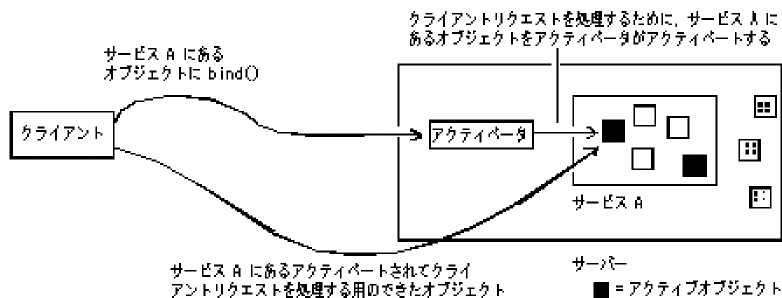
```
// Server.java
import com.borland.vbroker.extension.*;
. . .
class DBActivator implements Activator {
   private static int _count;
   private  com.borland.vbroker.CORBA.BOA _boa;
   public DBActivator(com.borland.vbroker.CORBA.BOA boa) {
      _boa = boa;
   }
   public org.omg.CORBA.Object activate(
         com.borland.vbroker.extension.ImplementationDef impl) {
      System.out.println("Activator called " + ++_count + " times");
      byte[] ref_data = ((ActivationImplDef) impl).id();
      DBObjectImpl obj = new DBObjectImpl(new String(ref_data));
      _boa.obj_is_ready(obj);
      return obj;
   }
   public void deactivate(org.omg.CORBA.Object obj,
         com.borland.vbroker.extension.ImplementationDef impl) {
      // ここでは何もしません ...
   }
}
. . .
```

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 34.1　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 34.1 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.java	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.java	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.java	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，Creator，Client

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，obj_is_ready に続けて 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，com.borland.vbroker.extention.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，com.borland.vbroker.extenstion.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。

```
// Server.java
class DBActivator implements Activator {
   private static int _count;
   private  com.borland.vbroker.CORBA.BOA _boa;
   public DBActivator(com.borland.vbroker.CORBA.BOA boa) {
      _boa = boa;
   }
   public org.omg.CORBA.Object activate(
         com.inprise.vbroker.extension.ImplementationDef impl) {
      System.out.printIn("Activator called " + ++_count + " times");
      byte[] ref_data = ((ActivationImplDef) impl).id();
      DBObjectImpl obj = new DBObjectImpl(new String(ref_data));
      _boa.obj_is_ready(obj);
      return obj;
   }
   public void deactivate(org.omg.CORBA.Object obj, ImplementationDef impl) {
      // ここでは何もしません ...
   }
}
```

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 ReferenceData を取得できます。

次のサンプルコードは，サーバーアクティベータの実装例です。

```
public org.omg.CORBA.Object activate(ImplementationDef impl) {
   System.out.println("Activator called " + ++_count + " times");
   byte[] ref_data = ((ActivationImplDef) impl).id();
   DBObjectImpl obj = new DBObjectImpl(new String(ref_data));
 _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は，メインサーバープログラムで impl_is_ready を呼び出して，DBActivator サービスアクティベータを作成および登録します。

```
public static void main(String[] args) {
   org.omg.CORBA.ORB orb = ORB.init(args, null);
   com.borland.vbroker.CORBA.BOA boa = ((com.borland.vbroker.orb.ORB
)orb).BOA_init();
   DB db = new DBImpl("Database Manager");
   boa.obj_is_ready(db);
   boa.impl_is_ready("DBService", new DBActivator(boa));
}
```

メモ　impl_is ready の呼び出しは，通常の 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，activate 内で 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 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。

```
public DBObject create_object(String name) {
   System.out.println("Creating: " + name);
   DBObject dbObject = new DBObjectImpl(name);
 _boa().obj_is_ready(dbObject, "DBService", name.getBytes());
   return dbObject;
}
```

2 度めの obj_is_ready 呼び出しは，activate 内で明示的に行う必要があります。

第 35 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。

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。 **null** リファレンスが渡されると，VisiBroker ORB がこの例外を生成します。Any に **null** 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。	クライアントから要求が送信された後で，サーバーからクライアントに応答が返される前の処理中に通信が失われると，この例外が生成されます。
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::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。
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。

例外	説明	考えられる原因
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。
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。

例外	説明	考えられる原因
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。 CORBA::TIMEOUT には，次のマイナーコードがあります。 • 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。	
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。
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。

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。

システム例外	マイナーコード	説明
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，『VisiBroker VisiTransact ガイド』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 35.1　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。

表 35.1　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（続き）

例外	説明	考えられる原因
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，『VisiBroker VisiTransact ガイド』を参照してください。

表 35.2　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。
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。

表 35.2　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:: 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 36 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 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。

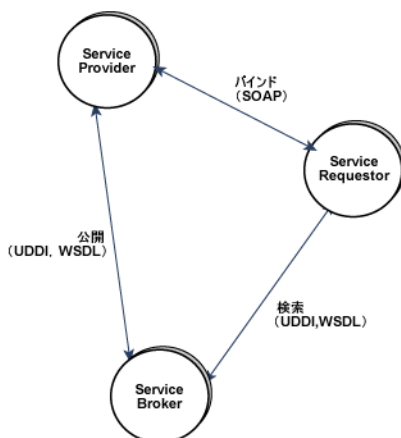
- *Service Broker* は，Service Requestor がアクセスする Web サービスを公開します。公開される情報の内容は，Web サービスとその場所です。また，Web サービスを公開するほかに，Web サービスのホスティングを調整します。

  Service 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。

- *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 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。

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）（バージョン 1.1）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/HTTPS 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。

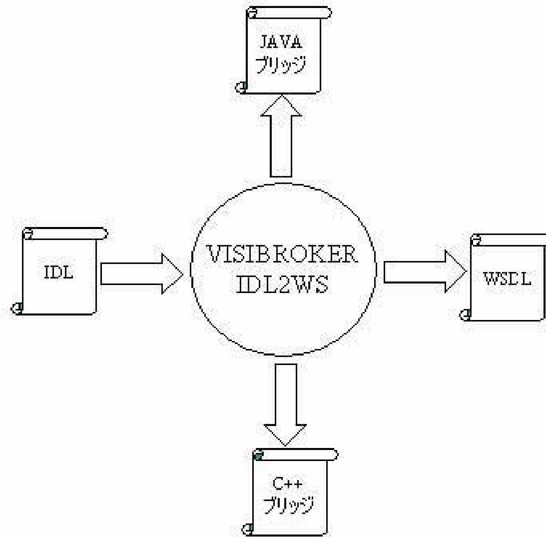
Web サービスランタイムを提供するために，VisiBroker は，Apache Axis テクノロジーを使用して Service Broker の複雑な部分を処理します。独自の型固有ブリッジ（ツールが生成）を使用して，配布されたステートレス CORBA オブジェクトにアクセスできます。型固有ブリッジインスタンスは，CORBA オブジェクトバックエンドの機能を Service Requestor に公開する Service Provider として動作します。

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。



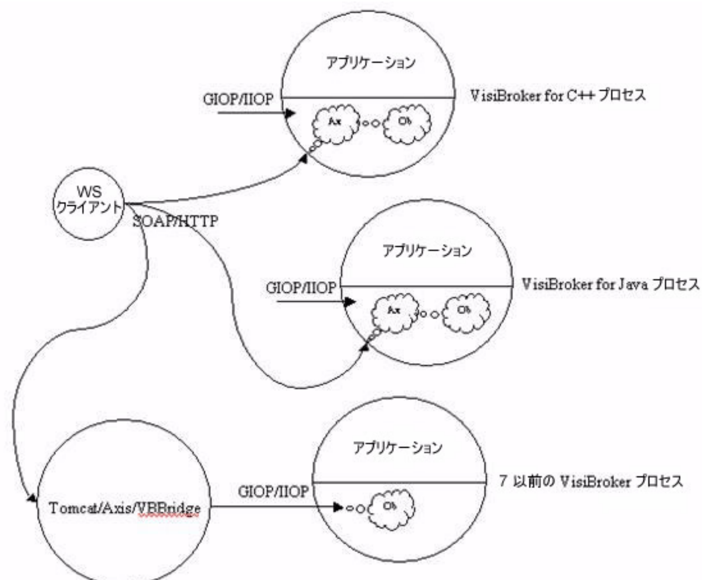
Web サービスランタイム

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 または SOAP/HTTPS 呼び出しを行う SOAP クライアントを示します。

VisiBroker for Java プロセスは，HTTP/SOAP および HTTPS/SOAP リスナーのインフラストラクチャを備えていますが，デフォルトでは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，HTTP/SOAP 実行時インフラストラクチャを開始できます。Web サービスが有効な場合は，プロパティ vbroker.security.disable=false を使用して，VisiBroker for Java の HTTPS/SOAP インフラストラクチャを起動できます。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

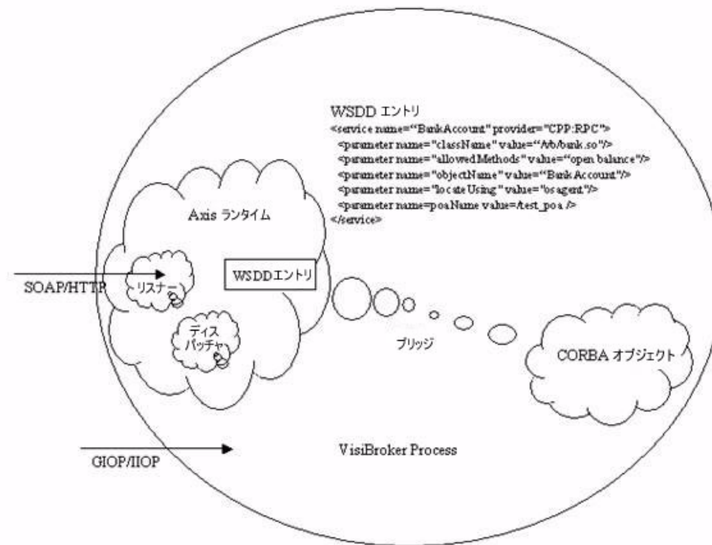
CORBA クライアントは，何の影響もなく通常どおり，GIOP/IIOP リスナーを通じて引き続き GIOP over IIOP 呼び出しを行うことができます。



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 Java 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。

1 開発

  a サーバー側サーバントスケルトンを生成する

  b IDL ファイルからインターフェース型固有の Java ブリッジを生成する

  c 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 配布

  a Web サービスランタイムを有効化／設定する

  b Axis WSDD メカニズムを使用して，VisiBroker プロセスでブリッジクラスを配布する

ここでは，examples ディレクトリの vbe/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（VisiBroker for Java サーバープロセスの SOAP over HTTPS の例は，security/ws/animal ディレクトリにあります）。この例は，vbe/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

1 サーバー POA サーバントコードを生成します。

idl2java コンパイラを使用して，CORBA インターフェース（Bank.idl の Account と AccountManager および BankWebService.idl の AccountManagerWebService）のサーバー側スケルトンクラスを生成します。

```
prompt> idl2java Bank.idl
prompt> idl2java BankWebService.idl
```

2 Java インターフェース型固有のブリッジを生成します。

-gen_java_bridge オプションを付けて idl2wsj コンパイラを使用すると，すべてのインターフェースの Java ブリッジを生成できます。次のコマンドは，BankWebService.idl のブリッジコードを AccountManagerWebService.java という名前のファイルに生成します。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> idl2wsj -gen_java_bridge BankWebService.idl
```

3 IDL から WSDL を生成します。

手順 2 の idl2wsj は，ブリッジコードに加えて，OMG の「CORBA to WSDL/SOAP Inter-working specification」（CORBA および WSDL/SOAP 間相互作用仕様）（バージョン 1.1）にしたがって，IDL ファイルの WSDL ドキュメントも生成します。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。idl2wsj を次のように使用して，WSDL ドキュメントだけを生成することもできます。

```
prompt> idl2wsj BankWebService.idl
```

生成されたブリッジコードを Web サービスとして配布します。

使用できるオプションのリストについては，「Java 対応プログラマツール」の章の idl2wsj のセクションを参照してください。

配布

1 最初に AXIS ランタイムに WSDD ドキュメントを配布します。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 WSDD（deploy.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，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。

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment>
    <service
        name="BankWebService.AccountManagerWebServicePort"
        provider="java:VISIBROKERPROVIDER">
        <namespace>
            http://BankWebService.AccountManagerWebService
        </namespace>
        <parameter
            name="className"
            value="[package].BankWebService_AccountManagerWebService "/>

          <operation name="openAndQueryBalance">
      <parameter qname="name" type="tns:string"
             xmlns:tns="http://www.w3.org/2001/XMLSchema/"/>
          </operation>
    </service>
</deployment>
```

2 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.se.ws.host=143.186.141.54
vbroker.se.ws.scm.ws_ts.listener.port=19000
vbroker.se.ws.scm.ws_ts.manager.connectionMax=30
vbroker.se.ws.scm.ws_ts.manager.connectionMaxIdle=300
vbroker.se.ws.scm.ws_ts.dispatcher.threadMin=0
vbroker.se.ws.scm.ws_ts.dispatcher.threadMax=300
vbroker.se.ws.scm.ws_ts.dispatcher.threadMaxIdle=300
```

3 次のようにサーバーを実行します。

```
prompt> vbj -DORBpropStorage=server.prop Server
```

4 生成されたブリッジコードは，次のように Axis ユーティリティ AdminClient を使用して，Axis ランタイムに deploy.wsdd（ブリッジとともに生成される）で配布できます。

```
prompt> java org.apache.axis.client.AdminClient
            -lhttp://143.186.141.54:19000/axis/deploy.wsdd
```

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 278
*_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
_get_policy 142
_is_a() メソッド 140
_is_bound() メソッド 140
_is_local() メソッド 140
_is_remote() メソッド 140
_set_policy_override メソッド 142
_tie クラス 133
  サンプル 134
  デリゲータインプリメンテーション 133
| 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 317
activate() メソッド 419
Activator クラス
  ORB オブジェクトの非アクティブ化 419
  オブジェクトのアクティブ化の遅延 419, 421
ActiveObjectLifeCycleInterceptor 347
  クラス 346
Agent
  レポート 170
Agent インターフェース 175
agentaddr ファイル
  IP アドレスの指定 168
Any
  Any 型のマッピング 59
  クラス 306
Any オブジェクト 300
Any 型
  DSI 317

B

backingStoreType 73
BAD_CONTEXT 例外 425
BAD_INV_ORDER 例外 425
BAD_OPERATION 例外 425
BAD_PARAM 例外 425
BAD_TYPECODE 例外 425
BiDirectional ポリシー 414
bind
  nsutil 189
  共通オブジェクトリファレンス 303
  プロセス 138
bind()
  osagent 163
bind_context
  nsutil 189
bind_new_context
  nsutil 189
BindInterceptor
  クラス 346
BOA
  VisiBroker における ... の使い方 417
  オブジェクトアクティベータ 418, 419
  下位互換サポート 419
  コードのコンパイル 417
  削除されたクラス 417
  サポートされるオプション 417
  使用する場合の制限事項 417
  ネーミングオブジェクト 418
  バインディング 171
BOA のオプション 417
BOA_init
  パッケージの変更 417
boolean 型
  マッピング 45
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

Caffeine コンパイラ
  説明 32
ChainUntypedObjectWrapper 364
char 型
  マッピング 45
CLASSPATH 34
classpath 36
ClientRequestInterceptor 346
  クラス 323, 346
  実装 333
ClusterManager 203
ClusterManager インターフェース 206
Codec 326
  インターフェース 326
  クラス 326
CodecFactory 326
  インターフェース 326
  クラス 326
COMM_FAILURE 例外 425
ConnEventListener インターフェース 379
connID 379
ConnInfo 379
  connID 379
  ipaddress 379
  ポート 379
container
  サーバーマネージャ 238
Container クラス 239
CORBA
  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, 133
CORBA 例外 425
corbaloc URL 191
corbaname URL 191
CosNaming
  コマンドラインからの呼び出し 188
CosNaming 操作
  VisiNaming によるサポート 189

CreationImplDef クラス 282
activation_policy プロパティ 282
args プロパティ 282
env プロパティ 282
path_name プロパティ 282
CreationImplDef 構造体
オブジェクトのアクティブ化 283
Current
インターフェース 326
custom valuetype 404

D

-D_VIS_INCLUDE_IR フラグ 295
DATA_CONVERSION 例外 425
DataExpress アダプタ 197
deactivate() メソッド 419
DefaultBindInterceptor
クラス 349
DefaultClientInterceptor
クラス 349
DefaultServerInterceptor
クラス 349
destroy
nsutil 189
DII 10
_request メソッドの使い方 304
Any オブジェクト 300
create_request メソッドの使い方 304
DII 要求の作成 304
idl2java コンパイラの使い方 302
NamedValue インターフェース 306
NamedValue クラス 306
NVList オブジェクト 301
Reply の受信オプション 300
Request インターフェース 303
Request オブジェクト 300
Request オブジェクトの使用 300
Request の送信オプション 300
send_deferred メソッド 309
send_oneway メソッド 310
Typecode オブジェクト 300
インターフェースリポジトリ 289, 310
応答の受信 302
概念 300
概要 299
可搬性のあるスタブの生成 32
機能の概要 10
共通オブジェクトリファレンス 303
クライアント 302
クライアントの構築 299
結果の受信 308
欠点 299
サンプル 302
非同期要求 310
複数の要求の受信 310
複数の要求の送信 310
要求の作成 303
要求の初期化 303
要求の送信 301, 308
要求の引数の設定 305
DSI
AccountManager インターフェース 317
Any 型 317
BAD_OPERATION 例外 317
DSI での入力処理 317
DynamicImplementation クラスから派生 314
idl2java コンパイラの使い方 313
ServerRequest クラス 317
オブジェクトインプリメンテーションの動的な作成 314
オブジェクトサーバーのコンパイル 314
オブジェクトのアクティブ化 318
オブジェクトの動的な作成 314
概要 313
クラスの派生 314
サーバーオブジェクトの実装 317
サンプル 314
スコープ解決演算子 316
入力パラメータ 317
プロトコル間ブリッジ 313
プロトコルのブリッジ 313
戻り値 317
DynamicImplementation クラス 314
… から派生するサンプル 314
DynAny
アクセスと初期化 390
値の初期化とアクセス 390
概要 389
作成 390
タイプ 389
DynAny インターフェース 389
current_component メソッド 391
DynAnyFactory オブジェクト 390
DynArray データ型 392
DynEnum インターフェース 391
DynSequence データ型 392
DynStruct インターフェース 391, 392
DynUnion インターフェース 391
NameValuePair 392
next メソッド 391
rewind メソッド 391
seek メソッド 391
to_any メソッド 392
構造データ型 391
サンプル 389
サンプル IDL 392
サンプルアプリケーション 392
サンプルクライアントアプリケーション 392
サンプルサーバーアプリケーション 393
制限 390
DynArray データ型 392
DynEnum インターフェース 391
DynSequence データ型 392
DynStruct インターフェース 391
DynUnion インターフェース 391

E

enableBiDir プロパティ 411
enum
マッピング 47
EventChannel 225
EventLibrary 234, 235
EventListener 379
接続の実装 381
登録 381
EventQueueManager インターフェース 379
exportBiDi プロパティ 411

F

factory_name 189
FREE_MEM 例外 425

G

get_listeners 379

H

Helper クラス
  マッピング 45
Holder クラス
  マッピング 42

I

id フィールド
  NameComponent 186
IDL
  DynAny サンプル 392
  IR に含まれる情報 289
  Java からの作成 31
  Java コードの生成 29
  Java による IDL の定義 32
  Java へのマッピング 16, 39
  Java へのマッピング名 39
  OAD インターフェース 286
  typedef のマッピング 59
  一方向メソッドの定義 156
  インターフェースの継承 156
  インターフェースリポジトリで表現される構造 290
  オブジェクトの指定 16
  型拡張 41
  構造型のマッピング 47
  コンパイラ 17
  サーバーマネージャ 239
  仕様のサンプル 152
  ネストした型のマッピング 59
  マッピング，インターフェース 53, 56
  マッピング，型 41
  マッピング，定数 46
  マッピング，パラメータ 55
  マッピング，モジュール 40
  予約語 40
  予約名 40
IDL 型
  boolean 45
  char 45
  Holder クラス 42
  octet 45
  wstring 45
  基本型 41
  整数型 45
  単純 59
  複雑 59
  浮動小数点数 45
  文字列 45
IDL ファイル
  #pragma メカニズム 278
idl2cpp コンパイラ 17
  一方向メソッドの定義 156
  インターフェースの継承 156
  コードの生成 152
  属性メソッド 156
idl2ir 28
  オプション 28
  コマンド情報 28, 29
  説明 28, 29
idl2ir コンパイラ 292
  コマンド情報 12
  説明 12
idl2java
  DII 用の可搬性のあるスタブの生成 29
  オプション 29
  コマンド情報 29
idl2java コンパイラ
  DII を使ったスタブコードの生成 302
  DSI を使ったスタブコードの生成 313
  -portable フラグ 302, 313
IIOP
  双方向 … サンプル 412, 414
  双方向 … の使用 411
  双方向 … の有効化 413
IMP_LIMIT 例外 425
impl_rep ファイル 275
import ステートメント
  VisiNaming サービス 214
  ネーミングサービス 214
importBiDi プロパティ 411
importBiDir 414
INITIALIZE 例外 425
Interceptor
  インターフェース 322
  クラス 322
  デフォルトのインターセプタクラス 349
InterfaceDef オブジェクト
  インターフェースリポジトリ 290
INTERNAL 例外 425
INTF_REPOS 例外 425
INV_FLAG 例外 425
INV_INDENT 例外 425
INV_OBJREF 例外 425
INVALID_TRANSACTION 例外 425
InvalidPolicy 例外 414
InvalidURL 例外 409
invoke() メソッド 313, 314
  実装のサンプル 314
IOR インターセプタ 321
IORCreationInterceptor 348
  クラス 346
IORInfoExt
  クラス 330
IORInterceptor
  インターフェース 325
  クラス 325
IP サブネットマスク
  localaddr ファイル 166
  スコープを指定するブロードキャストメッセージ 165
ipaddress 379
IR
  オブジェクト情報へのアクセス 295
  オブジェクトの識別 293
  継承元のインターフェース 294
  構造体 292
  サンプル 295
  説明 289
  内容 293
  保存できるオブジェクトの型 293
IR → インターフェースリポジトリ 10
ir2idl 29
  オプション 29
ir2idl ユーティリティ
  IR の内容の表示 292
irep ツール
  インターフェースリポジトリの作成 291
  インターフェースリポジトリの表示 292

J

Java
  IDL インターフェースの定義 32
  IDL からのマッピング 39
  IDL ファイルからコードを生成 29
  IIOP を介した RMI のプロパティ 61
  Java 開発者キット（JDK）12
  Java からの IDL ファイルの作成 31
  null 44
  vbj によるインタープリタの起動 34
  実行時環境ジッコウジカンキョウ 13
Java アプレット
  RMI-IIOP の設定 383
java2idl 383
  オプション 31
  コマンド情報 31
  説明 31
java2iiop 383
  DII 用の可搬性のあるスタブの生成 32
  オプション 32
  コマンド情報 32
  複合データ型のマッピング 387
  プリミティブ型のマッピング 387
JDBC アダプタ 197
JDBC アダプタのプロパティ 73
jdbcDriver 73
JVM 36

K

kind フィールド
  NameComponent 186

L

list
  nsutil 189
localaddr ファイル
  インターフェースの用法の指定 166
loginPwd 73

M

MARSHAL 例外 425
maxQueueLength 233
method 142
ModuleDef オブジェクト
  インターフェースリポジトリ 290

N

name
  解像度 186
  定義 186
NameComponent
  id フィールド 186
  kind フィールド 186
  定義 186
NamedValue
  オブジェクト 305
  クラス 305
  ペア 305
NamingContext
  ファクトリ 185
  ブートストラップ 185
NamingContextExt 192
NamingContexts
  オブジェクトインプリメンテーションによる使用 185
  クライアントアプリケーションによる使用 185
  定義 185
new_context
  nsutil 189
nmake
  ... でコンパイル 21
nmake コンパイラ 21
NO_IMPLEMENT 例外 425
NO_MEMORY 例外 425
NO_PERMISSION 例外 425
NO_RESOURCES 例外 425
NO_RESPONSE 例外 425
nsutil 188
  bind 189
  bind_context 189
  bind_new_context 189
  destroy 189
  list 189
  new_context 189
  rebind 189
  rebind_context 189
  resolve 189
  shutdown 189
  unbind 189
null
  Java 44
null valuetype 400
null セマンティクス 403
NVList オブジェクト 301
NVList クラス 317
  ARG_IN パラメータ 317
  ARG_INOUT パラメータ 317
  ARG_OUT パラメータ 317
  引数リストの実装 305

O

OAD
  IDL のインターフェース 286
  impl_rep ファイル 275
  oadutil list 278
  osagent 162
  アクティブ化ポリシーの設定 283
  インターフェース名 277
  インプリメンテーションリポジトリ 275
  オブジェクトの登録 279, 283
  オブジェクトの登録解除 284
  オブジェクトのリスト 278
  概要 276
  起動 276
  スマートエージェント 162
  タイムアウトの指定 276
  登録されたオブジェクトの移行 170
  登録されたオブジェクトの複製 169
  登録情報 275
  登録情報の保存 278
  とスマートエージェント 276
  プログラミングインターフェース 286
  プロパティ 77
  リポジトリ ID 277
  渡される引数 284
OAD コマンド
  環境変数の設定 276
oadj
  レポート 170
oadutil
  OAD に登録されたオブジェクトの一覧表示 278

インプリメンテーションの登録解除 284
oadutil list 278
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 286
オブジェクトインプリメンテーションの登録 275
OBJ_ADAPTOR 例外 425
OBJECT_NOT_EXIST 例外 425
ObjectWrapper 370
octet
マッピング 45
OMG 7
イベントサービス 223
コモンオブジェクトサービス仕様 224
通知サービス 223
OMG 準拠の機能
VisiNaming サービス 214
ネーミングサービス 214
open() メソッド 317
OpenLDAP 202
OperationDef オブジェクト
インターフェースリポジトリ 290
ORB
resolve_initial_references 190
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 10
オブジェクトインプリメンテーション 278
オブジェクトへのバインド 138
機能 7
初期化 89, 137
相互運用性 13
定義 171
ドメイン 164
バインド処理中のオブジェクトへの接続 138
プロキシの作成 171
プロパティ 63
ORB, クライアントランタイム 149
ORB_init() メソッド
-ORBshmsize 34
ORBDefaultInitRef のプロパティ 191
ORBInitializer
インターフェース 328
クラス 328
実装 333
登録 328, 332
ORBInitInfo
インターフェース 328
クラス 328
ORBInitRef 188
ORBInitRef のプロパティ 191
ORInfoExt
インターフェース 330
OSAgent
vbj コマンドによる検索 34
osagent
bind() 163
オブジェクトの検索 161
オブジェクト名 418
可用性の確認 163
起動 162
クライアントの確認（ハートビート）163
詳細出力 162
スマートエージェント 159
スマートエージェントの起動 21
バインディング 171
別のエージェントの検出 165
無効化 162, 163
レポート 170
osagent ログファイル
オプション 163
OSAGENT_ADDR 34
OSAGENT_ADDR 環境変数 168
OSAGENT_LOCAL_FILE
環境変数 166
OSAGENT_PORT 34
OSAgent（スマートエージェント）
VisiBroker のアーキテクチャ 8
osfind
コマンド情報 170

P

PDF マニュアル 3
PERSIST_STORE 例外 425
PICurrent
クラス 326
POA
BiDirectional ポリシー 414
ObjectID 96
POA の管理 109
POA マネージャ 96, 109
rootPOA 96, 100
servant 96
ServantLocators 107
アクティブオブジェクトマップ 96
アクティブ化 100
アダプタアクティベータ 96, 117
一時的オブジェクト 96
オブジェクトのアクティブ化 101, 104
オブジェクトの非アクティブ化 103
具現化 96
サーバーエンジン 112
サーバー接続マネージャ 113
サーバントの使用 104
サーバントマネージャ 96, 104
作成 90, 97, 99
双方向 IIOP の有効化 414
定義 95
ディスパッチャのプロパティ 111, 114
デフォルトサーバントによるアクティブ化 102
プロパティ 68
ポリシー 96, 97
要求の処理 117
リスナーのプロパティ 111, 114
リスナーポートプロパティ 115
霊化 96
POALifeCycleInterceptor 347
クラス 346
poolSize 73
ProxyPullConsumer 226
ProxyPullSupplier 226
ProxyPushConsumer 225
ProxyPushSupplier 225
PullConsume 231
PullModel 229
PullSupplierPOA
クラス 229
PullSupply 229, 231
PushConsumer
サンプル 228
PushModel 229
PushModel クラス 228
PushSupplier
実装 228
PushSupplier インターフェース 228

Q

QoS 141
　インターフェース 142
Quality of Service（Qos）
　プロパティ 79

R

rebind
　nsutil 189
rebind_context
　nsutil 189
rebinds
　スマートエージェントで有効 168
ref_data パラメータ 282
register_listener 379
Reply の受信オプション 300
Repository クラス 295
Request オブジェクト 300
　DII 300
Request クラス 303
Request の送信オプション 300
RequestInterceptor
　実装 333
REQUIRE_AND_TRUST 414
resolve
　nsutil 189
Resolver インターフェース
　URL をオブジェクトに関連付ける 408
RMI 383
RMI over IIOP 383
RMI over IIOP →「RMI-IIOP」383
RMI-IIOP 383
　IDL から Java クラスへのマッピング 384
　Java アプレット 383
　java.policy ファイル 383
　java2idl 383
　java2iiop 383
　java2iiop のインターフェース 384
　java2iiop の実行 384
　アプレット権限の設定 383
　概要 383
　サポートされる Java クラス 387
　サンプル 384, 385
root NamingContext 185
rootPOA 100
RoundRobin
　VisiNaming サービス 207
　ネーミングサービス 207

S

SCM
　双方向 IIOP 411
sequence
　マッピング 47
ServerRequest クラス 317
ServerRequestInterceptor 348
　インターセプトポイント 324
　クラス 346
　実装 333
ServiceInit クラス 350
ServiceLoader インターフェース 350
ServiceResolverInterceptor 349
shutdown
　nsutil 189
SSL
　VisiNaming 212
　VisiNaming の設定 212
　双方向 IIOP 415
　ネーミングサービス 212
　ネーミングサービスの設定 212
Storage インターフェース 241
　サーバーマネージャ 239
string_to_object() メソッド 139
SVCnameroot 188
SVCnameroot のプロパティ 190

T

TRANSACTION_MODE 例外 425
TRANSACTION_REQUIRED 例外 425
TRANSACTION_ROLLEDBACK 例外 425
TRANSACTION_UNAVAILABLE 例外 425
TRANSIENT 例外 425
truncatable valuetype 405
Typecode オブジェクト 300
TypeCode クラス 307
typedef 宣言
　マッピング 59

U

UDP プロトコル 161
unbind
　nsutil 189
UNKNOWN 例外 425
unregistered_listener 379
UntypedObjectWrapper
　post_method 364
　pre_method 364
URL
　ネーミングサービス 407
URL ネーミング
　プロパティ 79
URL ネーミングサービス 407
　InvalidURL 例外 409
　URL をオブジェクトに関連付ける 408
　オブジェクトの検索 409
　サンプル 407

V

valuetype 399
　CustomMarshal インターフェース 404
　Factory クラスの実装 402
　IDL ファイルのコンパイル 401
　isomorphic 400
　marshal メソッド 404
　null 400
　null セマンティクス 403
　truncatable 405
　unmarshal メソッド 404
　valuetype 基底クラスの継承 401
　アンマーシャリング 404
　インプリメンテーションクラス 401
　概要 399
　書き込みメソッド 405
　カスタム 404
　基底クラス 401
　共有セマンティクス 403
　共用 400
　具象 400
　実装 400
　抽象 400

抽象インターフェース 404
定義 401
登録 403
派生 399
ファクトリ 399, 400, 403
ファクトリの実装 402
ファクトリを ORB に登録 402
ボックス化 403
マーシャリング 404
読み取りメソッド 405
valuetype の実装 400
vbj コマンド
説明 34
vbmake
… でコンパイル 21
vbroker.naming.backingStore 73
vbroker.naming.cache 202
vbroker.naming.enableSlave プロパティ 209
vbroker.naming.jdbcDriver 73
vbroker.naming.loginName 73
vbroker.naming.loginPwd 73
vbroker.naming.poolSize 73
vbroker.naming.propBindOn 206
vbroker.naming.serverAddresses プロパティ 209
vbroker.naming.serverClusterName プロパティ 209
vbroker.naming.serverNames プロパティ 209
vbroker.naming.slaveMode プロパティ 209
vbroker.naming.url 73
vbroker.orb.dynamicLibs プロパティ 350
vbroker.orb.enableBiDir プロパティ 411
vbroker.orb.enableServerManager プロパティ 242
vbroker.security.peerAuthenticationMode 414
vbroker.serverManager.enableOperations プロパティ 242
vbroker.serverManager.enableSetProperty プロパティ 242
vbroker.serverManager.name プロパティ 238
VisiBroker
BOA の下位互換性 417
CORBA 準拠 11
機能 8
サンプルアプリケーション 15
説明 8
VisiBroker ORB
初期化 137
VisiBroker ORB 拡張
VisiNaming サービス 214
ネーミングサービス 214
VisiBroker インターセプタ
サンプル 350
VisiBroker インターセプタ（インターセプタ）345
VisiBroker の機能 8
IDL コンパイラ 9
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 9
スマートエージェントのアーキテクチャ 8
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 202
SSL の使い方 212
SSL 用のプロパティ（C++）212
SSL 用のプロパティ（Java）212
SSL を使用するように設定 212
キャッシング機能 202
ブートストラップ 213
メソッドレベル承認 213
VisiNaming サービス
import ステートメント 214
Java で初期化 193
nsutil ユーティリティ 188
OMG 準拠の機能 214
VisiBroker ORB 拡張 214
アダプタ 199
インストール 187
概要 183
起動 187, 188
クライアント認証 212
クラスタ 203
クラスタの作成 206
サポートされている CosNaming 操作 189
サンプル 215
サンプルプログラム 215
シャットダウン 190
セキュリティ 212
設定 187
デフォルトネーミングコンテキスト 193
取り替え可能なバックストア 197
ブートストラップ 190
フェイルオーバー 208
フォールトトレランス 209
負荷分散 207
プロパティ 70, 193
プロパティファイル 199
マスター／スレーブモード 210
メソッドレベル承認 212
VISObjectWrapper
UntypedObjectWrapperFactory 364
ChainUntypedObjectWrapper 364
UntypedObjectWrapper 364
Visual C++ nmake コンパイラ 21

W

Web サイト
CORBA 仕様 11
Web ネーミング
URL をオブジェクトに関連付ける 407
Web サイト
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 162
コンソールモード 162
wstring
マッピング 45

あ

アクティブ化 9
サービスのアクティブ化 420
アダプタ
DII 299

VisiNaming サービス 199
ネーミングサービス 199
アプリケーション
vbj コマンドによる osagent の検索 34
オブジェクトインターフェースの定義 16
クライアントプログラムの起動 21
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 122
セッションごとのスレッド 124
双方向 IIOP の有効化 413
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
OAD に登録されたオブジェクト 170
インスタンス化されたオブジェクト 170
オブジェクト 169
状態を持つオブジェクト 169
ホスト間のオブジェクト 169
一方向メソッド
定義 156
イベントキュー 379
ConnEventListener インターフェース 379
EventListener インターフェース 379
EventListener の登録 381
EventQueueManager インターフェース 379
イベントタイプ 379
イベントリスナー 379
概要 379
サンプルコード 381
接続 EventListener 381
接続イベント 379
イベントサービス
インプロセスイベントチャネル 235
概要 223
起動 233
キューの長さの設定 233
サンプル 228
通信モデル 225
プッシュモデル 225
プルモデル 226
イベントタイプ 379
接続タイプ 379
イベントチャネル 226
インプロセスインプリメンテーション 234
イベントリスナー 379
ConnInfo 379
インスタンス
オブジェクトリファレンスの ... の判定 140
ロケーションサービスを使った検索 173
インターセプタ
ActiveObjectLifeCycleInterceptor 347
API クラス 346
BindInterceptor 346
ClientRequestInterceptor 346
IOR 321
IORCreationInterceptor 348
ORB によるインターセプタの登録 349
ORB のカスタマイズ 10
POALifeCycleInterceptor 347
ServerRequestInterceptor 348
ServiceResolverInterceptor 349
インターセプタオブジェクトの作成 350
インターフェース 346
概要 345
クライアント 346
クライアントインターセプタ 345
クライアント側のポータブルインターセプタ 358
サーバー 347
サーバーインターセプタ 345
サーバー側のポータブルインターセプタ 358
サンプルプログラム 350
使い方 345
データの受け渡し 357
ポータブルインターセプタの使用 357
マネージャ 346
ロード中 350
インターセプタインターフェース
ORB に登録 349
サンプル 350
インターセプタオブジェクト
作成 350
インターセプトポイント
ServerRequestInterceptor 324
要求インターセプトポイント 323, 324
呼び出しの順序 357
インターフェース
Codec 326
CodecFactory 326
ConnEventListeners 379
Current 326
EventListener 379
EventQueueManager 379
IDL での定義 16
Interceptor 322
IORInterceptor 325
java2iiop の使い方 387
NamingContextExt 192
ORBInitializer 328
ORBInitInfo 328
ORInfoExt 330
QoS 142
インターフェースリポジトリの説明 289
継承 156
継承する ... の指定 156
検索 295
属性 156
マッピング 53, 56
レポート 170
インターフェース定義言語（IDL）16
インターフェースの継承
指定 156
インターフェースのスコープ
マッピング 57
インターフェース名
OAD からのオブジェクトの登録解除 284
取得 139
定義 152
リポジトリ ID に変換 277
インターフェースリポジトリ 10
_get_interface() メソッド 290
idl2ir による記入 12, 28, 29
idl2ir を使った内容の更新 292
オブジェクト情報へのアクセス 295
オブジェクトの識別 293
機能の概要 10
継承元のインターフェース 294
構造体 292
作成 291
サンプル 295
説明 289

内容 290, 293
内容の表示 292
プロパティ 77
保存できるオブジェクトの型 293
インターフェースリポジトリ（IR）289
インプリメンテーション
OAD からの登録解除 284, 285
アクティブ化 9
サポート 9
ステートレス，メソッドの呼び出し 169
スマートエージェントへの接続 159
セッションごとのスレッドの使い方 124
デリゲーションの使用 56
バインディング 171
フォールトトレランス 168
レポート 170
インプリメンテーションリポジトリ 10
impl_rep ファイル 275
OAD 278
OAD からの登録解除時の削除 284
OAD によるディレクトリの指定 276
OAD の使用 276
オブジェクトの登録解除 284
機能の概要 10
内容の一覧表示 286
保存された登録情報 275
インプロセスイベントチャネル 234, 235
インメモリアダプタ 197

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 316

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 283
DSI を使った動的な作成 314
IDL での指定 16
OAD からの登録解除 284
OAD による接続 162
URL による検索 407
URL の関連付け 407
アクティブ化 421
アクティブ化ポリシーの設定 283
インターフェースリポジトリの情報へのアクセス 295
実行可能ファイルのパス 283
ステートレス，メソッドの呼び出し 169
スマートエージェントへの接続 159
登録 282, 283
特性の動的な変更 282
ネットワーク上のオブジェクトのレポート 170
バインディング 171
複数のインスタンス 282
複製 169
メソッドを呼び出す状態 169
リスト 278
ロケーションサービスを使った検索 173
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）162
オブジェクトアクティベータ 419
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 169
動的な変更 282
フォールトトレランス 168
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, 282
deferred メソッドサンプル 421
OAD によって渡される引数 284
サービスアクティベータを使った … の遅延 421
サービスのアクティブ化 420
サポート 9
遅延 419
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 421
オブジェクトの移行 169
オブジェクトの登録
using oadutil 279
変更 282
オブジェクトの登録解除
OAD 284
using oadutil 284
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 138
オブジェクトラッパー
idl2java 要件 362
ORB のカスタマイズ 10
post_method 363
pre_method 363
概要 361
型付き 361, 367
型付き … の呼び出し順序 369
型付きラッパーと型なしラッパーの併用 372
型付きラッパーの削除 372
型付きラッパーの追加 370
型付きラッパーの派生 370
型なし 361, 362
型なしのインストール 364
型なしの実装 364
型なしの使用 364
型なしファクトリ 364
型なしファクトリの削除 367
共用クライアント／サーバー 369
サンプルアプリケーションの実行 375
サンプルプログラム 362
説明 361
ファクトリの追加 365
複数の型付き … の使用 368
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 140
インスタンスの型の判定 140
インターフェース名の取得 139
永続的 418
オブジェクト名の取得 139
型の判定 140
型の変換 141
サブタイプ 140
状態の判定 140
スーパータイプへの変換 141
操作 139
等価のインプリメンテーションのチェック 140
ナローイング 141
場所の判定 140

ハッシュ値の取得 140
文字列への変換 139
リポジトリ ID の取得 139
ワイドニング 141
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 223
開発者サポート，連絡 4
概要 1
VisiNaming サービス 183
型拡張 41
型なしオブジェクトラッパー 362
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 131
環境変数
OAD 276
OSAGENT_ADDR 168
OSAGENT_LOCAL_FILE 166
管理コマンド
oadutil list 278
oadutil unreg 284
osfind 170
完了状態 84
システム例外の取得 84

き

記号
省略符 ... 4
縦線｜ 4
ブラケット [ ] 4
起動機能の概要 10
基本型
IDL 型 41
キャッシング機能 202
キューの長さ
設定値 233
共通オブジェクトのテスタ
DII 299
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 403
共用体
マッピング 47

く

クライアント
DII の使用 302
ORB の初期化 137
vbj コマンドによる osagent の検索 34
サーバーへの一方向接続 414
サーバーへの双方向接続 414
サーバーマネージャの参照 238
実装 18
スレッドプールの使い方 122
セッションごとのスレッドの使い方 124
双方向 IIOP 411
動的起動インターフェースを使った ... の構築 299
非同期情報の受信 411
クライアントインターセプタ 345
クライアント側インプロセス接続
プロパティ 79
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, 22
クライアント認証
ネーミングサービス 212
クライアントランタイム 149
クライアントリクエストインターセプタ
サンプル 334
クラス
_tie 133
ActiveObjectLifeCycleInterceptor 346
Any 306
BindInterceptor 346
ClientRequestInterceptor 323, 346
Codec 326
CodecFactory 326
CreationImplDef 282
DefaultBindInterceptor 349
DefaultClientInterceptor 349
DefaultServerInterceptor 349
DynamicImplementation 314
Interceptor 322
IORCreationInterceptor 346
IORInterceptor 325
NamedValue 305
NVList 317
NVList ARG_IN パラメータ 317
NVList ARG_INOUT パラメータ 317
NVList ARG_OUT パラメータ 317
ORBInitializer 328
ORBInitInfo 328
ORInfoExt 330
PICurrent 326
POALifeCycleInterceptor 346
PullConsume 231
PullConsumerPOA 231
PullSupplierPOA 229
ServerRequest 317
ServerRequestInterceptor 346
TypeCode 307
デフォルトのインターセプタ 349
ネーミングコンテキスト 192
要求 303
リポジトリ 295
クラスタ 203
ネーミングサーバーで作成 206
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 159

け

継承
インターフェース 156
インプリメンテーションから 133
サーバーの実装 56

こ

語
予約 40
構造型
マッピング 47
構造体
マッピング 47
コード
BOA のコンパイル 417
nmake でビルド 21
vbmake でビルド 21
ビルド 20

コード生成 17
コードセットサポート 148
タイプ 148
転送コードセット 148
ネイティブコードセット 148
ネゴシエーション 148
分類 148
変換コードセット 148
コマンド
idl2ir 28, 29
idl2java 29
java2idl 31
java2iiop 32
vbj 34
コマンド，規約 4
コンパイラ
IDL, 機能の概要 9
java2idl 383
java2iiop 383
nmake 21
vbmake 21
コンパイル
BOA コード 417

さ

サーバー
GateKeeper なしのコールバック 411
OAD に登録 283
tie メカニズムのサンプル 134
アクティブ化ポリシーの設定 283
クライアントへの一方向接続 414
クライアントへの接続の開始 411
クライアントへの非同期情報の送信 411
クライアント要求の受け取り 89
クライアント要求の待機 92
継承の使用 56
サーバーマネージャ 238
実装 20
スレッドにおける留意点 128
セットアップ 89
双方向 IIOP 411
サーバーインターセプタ 345
サーバーエンジン
POA 112
サーバー側サーバーエンジン
プロパティ 79
サーバー側スレッドセッション BOA_TS 接続
プロパティ 80
サーバー側スレッドセッション IIOP_TS 接続
プロパティ 80
サーバー側スレッドプール BOA_TP 接続
プロパティ 82
サーバー側スレッドプール IIOP_TP 接続
プロパティ 81
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 113
サーバーマネージャ
Container インターフェース 239
Container クラス 239
Container のメソッド（Java）239
IDL 定義 242
Storage インターフェース 239, 241
アクセス可能性 242
概要 237
カスタムコンテナ 247
コンテナ 238
サンプル 244
はじめに 237
プロパティ 68
有効化 237
リファレンスの取得 238
カスタムコンテナの書き込み 247
サーバーマネージャ IDL 239
サーバーリクエストインターセプタ
POA スコープ付き 330
サンプル 334, 338
サービス
ネットワーク上のサービスのレポート 170
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 421
サンプル 421
... の遅延の実装 421
作業スレッド 120
作成
ソフトウェアコンポーネント 7
サブネットマスク 165, 166
サプライヤ
EventChannel への接続 226
サプライヤ／コンシューマ通信モデル 223
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 134
DII の使用 302
DSI 314
DynAny IDL 392
IR 295
odb 421
RMI-IIOP 384, 385
URL ネーミングサービス 407
VisiBroker インターセプタ 350
VisiNaming サービス 215
アクティブ化 421
インターセプタ 350
インターフェースリポジトリ 295
オブジェクトのアクティブ化 419, 422
オブジェクトのアクティブ化における deferred メソッド 421
オブジェクトラッパー 362
サーバーマネージャ 244
スマートエージェントの localaddr ファイル 166
双方向 IIOP 412
ネーミングサービス 215
プッシュコンシューマ 228
プッシュサプライヤ 228
ポータブルインターセプタ 331
リクエストインターセプタ 334
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 16
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 16
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 20

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 21
サンプルプログラム
VisiNaming サービス 215
ネーミングサービス 215

し

システム例外
BAD_CONTEXT 425
BAD_INV_ORDER 425
BAD_OPERATION 425
BAD_PARAM 425
BAD_QOS 425
BAD_TYPECODE 425
COMM_FAILURE 425
CompletionStatus 値 84
CORBA 定義 83
DATA_CONVERSION 425
FREE_MEM 425
IMP_LIMIT 425
INITIALIZE 425
INTERNAL 425
INTF_REPOS 425
INV_FLAG 425
INV_INDENT 425
INV_OBJREF 425
INVALID_TRANSACTION 425
Java へのマッピング 58
MARSHAL 425
NO_IMPLEMENT 425
NO_MEMORY 425
NO_PERMISSION 425
NO_RESOURCES 425
NO_RESPONSE 425
OBJ_ADAPTOR 425
OBJECT_NOT_EXIST 425
PERSIST_STORE 425
SystemException クラス 83
TRANSACTION_MODE 425
TRANSACTION_REQUIRED 425
TRANSACTION_ROLLEDBACK 425
TRANSACTION_UNAVAILABLE 425
TRANSIENT 425
UNKNOWN 425
完了状態の取得 84
キャッチ 85
処理 85
ポータブルインターセプタ 330
例外のナローイング 85
システム例外の処理 85
実装
サーバー 20
指定
IP アドレス 168
状態完了
システム例外の取得 84
承認
VisiNaming のメソッドレベル 213
VisiNaming メソッドレベル 212
ネーミングサービスのメソッドレベル 212, 213

す

スケルトン 17
スコープ解決演算子 316
スタブ
DII 用の可搬性のある ... の生成 29, 32
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 169
スマートエージェント
bind() 163
OAD 162, 276
OAD によるオブジェクトへの接続 162
osagent 159
OSAGENT_ADDR 環境変数 168
OSAGENT_LOCAL_FILE ファイル 166
インターフェースの用法の指定 166
オブジェクトの自動再登録 163
オブジェクトのフォールトトレランス 168
オブジェクト名 418
概要 159
可用性 163
起動 162
機能の概要 8
クライアントの確認（ハートビート）163
検索 161
異なるネットワーク上の接続 165
最善の方法 161
削除されるオブジェクト 284
詳細出力 162
通信 161
ネーミングサービスの負荷分散 207
バインディング 171
複数のインスタンスの起動 161
複数のドメインでの実行 164
プロパティ 62
別のエージェントの検出 165
ポイントツーポイント通信
通信 167
ほかのエージェントとの協力 161
マルチホームホスト 166
無効化 162, 163
ロケーションサービス 161, 173
スマートエージェント（OSAgent）
アーキテクチャ 8
スレッド
ガベージコレクション 131
作業スレッド 120, 121, 124
スレッドの使い方 119
スレッドプールポリシー 121
スレッドポリシー 120
セッションごとのスレッドポリシー 124
使い方 119
ディスパッチのポリシーとプロパティ 126
同期ブロックの使用 128
プロパティ 129
マルチスレッド , 機能概要 9
リスナースレッド 120
スレッド管理 9
スレッドプールディスパッチポリシー 126
スレッドポリシー 120

せ

整数
マッピング 45
製品のバージョン 12, 29, 31, 32, 34
セキュリティ
VisiNaming サービス 212
VisiNaming サービスクライアント認証 212
VisiNaming サービスのメソッドレベル承認 212
双方向 IIOP 414
ネーミングサービス 212

ネーミングサービスクライアント認証 212
ネーミングサービスのメソッドレベル承認 212
セキュリティ（C++）
VisiNaming での有効化 212
ネーミングサービスでの有効化 212
セキュリティ（Java）
VisiNaming での有効化 212
ネーミングサービスでの有効化 212
セッションごとのスレッドのインプリメンテーション 124
セッションごとのスレッドのディスパッチポリシー 127
接続
ガベージコレクション 131
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 165
ポイントツーポイント通信 167
接続管理 9, 125
プロパティ 128

そ

相互運用性 13
ORB 相互運用性 13
VisiBroker for C++ との 13
VisiBroker for Java との 13
ほかの ORB 製品との 13
双方向 IIOP 411
InvalidPolicy 例外 414
POA 414
一方向接続 414
既存のアプリケーションでの有効化 413
サンプル 412, 414
セキュリティ 414
双方向 SCM 411, 414
双方向のプロパティ 411
ソフトウェアの更新 5

た

タイプ
Any 317
DynAny 389
インスタンスの判定 140
オブジェクトリファレンスの … の判定 140
サブタイプの判定 140
マッピング 41
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 290
単純な名前 186

ち

遅延
オブジェクトのアクティブ化 421
抽象 valuetype 400
抽象インターフェース 54, 404

つ

ツール
CORBA サービス 12
idl2cpp 17
idl2ir 12, 28, 29
idl2java 29
java2idl 31
java2iiop 32
oadutil 279
oadutil unreg 284
osfind 170
vbj 34
管理 12
プログラミング 12

て

定数
マッピング 46
ディスパッチのポリシーとプロパティ 126
ディスパッチポリシー
スレッドプール 126
セッションごとのスレッド 127
ディスパッチャのプロパティ 114
テクニカルサポート，連絡 4
デフォルトのファクトリ 403
デリゲーション
サーバーインプリメンテーション 56
転送コードセット 148

と

動的起動インターフェース →「DII」299
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」313
登録
OAD インプリメンテーションリポジトリ 275
スマートエージェント 159
ドメイン
複数の … の実行 164
トリガー 175, 177
作成 177
取り替え可能なバックストア
種類 197
設定 198
プロパティファイル 198

な

名前
オブジェクトへの名前のバインド 183
単純 186
複雑 186
文字列化 186
名前空間 183
名前の解決 186
ナローイング
例外をシステム例外に 85

に

ニュースグループ 5
入力パラメータ
DSI での処理 317
入力／出力引数
メソッド呼び出し要求 305
認証
VisiNaming クライアント 212
双方向 IIOP 415
ネーミングサービスクライアント 212

ね

ネイティブコードセット 148
ネイティブメッセージング 249
ネーミングコンテキスト
クラス 192

デフォルト 193
ネーミングサービス
import ステートメント 214
Java で初期化 193
OMG 準拠の機能 214
SSL の使い方 212
SSL 用のプロパティ（C++）212
SSL 用のプロパティ（Java）212
SSL を使用するように設定 212
URL 407
VisiBroker ORB 拡張 214
アダプタ 199
インストール 187
起動 187, 188
キャッシング機能 202
クライアント認証 212
クラスタ 203
クラスタの作成 206
サポートされている CosNaming 操作 189
サンプル 215
サンプルプログラム 215
シャットダウン 190
セキュリティ 212
セキュリティの有効化（C++）212
セキュリティの有効化（Java）212
設定 187
デフォルトネーミングコンテキスト 193
取り替え可能なバックストア 197
ブートストラップ 190, 213
フェイルオーバー 208
フォールトトレランス 209
負荷分散 207
プロパティ 70, 193
プロパティファイル 199
メソッドレベル承認 212, 213
ネストした型
マッピング 59
ネットワーク
オブジェクトとサービスのレポート 170

は

配布
オブジェクトインターフェースの定義 16
説明 22
配列
マッピング 47
バインディング
ORB のタスク 171
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 138
オブジェクトへのバインド 138
確立されたオブジェクトへの接続 138
作成されるプロキシオブジェクト 138
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 197
パフォーマンスの向上 202
パラメータ
マッピング 55

ひ

引数
-ORBshmsize 34
非同期通信 411
ヒューリスティックな例外 429

ふ

ファイル
idl コンパイラによる生成 17
impl_rep 275
localaddr 166
コンパイルによる生成 17
ファイル拡張子 17
ファクトリ 400
valuetype 402
実装 402
デフォルト 403
ファクトリクラス 402
フェイルオーバー
VisiNaming サービス 208
ネーミングサービス 208
フォールトトレランス 8
OAD に登録されたオブジェクトの複製 169
VisiNaming サービス 209
ネーミングサービス 209
負荷分散
VisiNaming サービス 207
ネーミングサービス 207
ホスト間のオブジェクトの移行 169
ロケーションサービスの使用 174
複雑な名前 186
プッシュコンシューマ 225
プッシュサプライヤ 225
サンプル 228
実装 228
プッシュモデル 225
浮動小数点
マッピング 45
ブリッジ
DII 299
プルコンシューマ 226
プルサプライヤ 226
プルモデル 226
ブロードキャストアドレス 166
ブロードキャストメッセージ 161
プロキシオブジェクト
バインディング 171
バインド処理中に作成 138
プロキシコンシューマ 224
プロキシサプライヤ 224
プログラマツール
idl2ir 28
ir2idl 29
プロセス
bind 138
プロパティ
DataExpress アダプタ 73
enableBiDir 411
IIOP を介した Java RMI 61
JDBC アダプタ 73
JNDI アダプタ 73
listener 114
OAD 77
ORB 63
ORBDefaultInitRef 191
ORBInitRef 191
POA 68
POA ディスパッチャ 111
POA リスナー 111
QoS 79
SVCnameroot 190

URL ネーミング 79
vbroker.naming.cache 202
vbroker.naming.enableSlave 209
vbroker.naming.propBindOn 206
vbroker.naming.serverAddresses 209
vbroker.naming.serverClusterName 209
vbroker.naming.serverNames 209
vbroker.naming.slaveMode 209
vbroker.orb.dynamicLibs 350
vbroker.orb.enableBiDir 411
vbroker.orb.enableServerManager 242
vbroker.serverManager.enableOperations 242
vbroker.serverManager.enableSetProperty 242
vbroker.serverManager.name 238
VisiBroker BiDirectional 411
VisiNaming サービス 70, 193
インターフェースリポジトリ 77
クライアント側インプロセス接続 79
サーバー側サーバーエンジン 79
サーバー側スレッドセッション BOA_TS 接続 80
サーバー側スレッドセッション IIOP_TS 接続 80
サーバー側スレッドプール BOA_TP 接続 82
サーバー側スレッドプール IIOP_TP 接続 81
サーバーマネージャ 68
スマートエージェント 62
スレッド管理 129
接続管理の設定 128
ディスパッチャ 114
ネーミングサービス 70, 193
ロケーションサービス 69
プロパティファイル
VisiNaming サービス 199
分散アプリケーション
開発手順 15

ヘルプトピック，アクセス 3
変換コードセット 148

ほ

ポイントツーポイント通信 167
ポータブルインターセプタ
Current 326
Interceptor 322
IOR インターセプタ 321, 325
PICurrent 326
POA スコープ付きサーバー要求 330
ServerRequestInterceptor 324
インターセプトポイント 324
概要 321
拡張機能 330
作成 327
サンプル 331
システム例外 330
制限 330
タイプ 321
登録 328
要求インターセプトポイント 323
リクエストインターセプタ 321, 323
ポータブルオブジェクトアダプタ
ポリシー 97
ポータブルオブジェクトアダプタ（POA）
定義 95
ポート番号
listener 115
ボックス化 valuetype 403
ポリシー 141
POA 97
有効な 141
ポリシーオーバーライド 141

ま

マーシャリング
java2iiop の使い方 387
マッピング 45
Any 型 59
boolean 型 45
char 型 45
enum 47
Holder クラス 42
IDL 型 41
IDL 名 39
octet 45
sequence 47
typedef 宣言 59
インターフェース 53
インターフェースのスコープ 57
共用体 47
構造型 47
構造体 47
整数 45
抽象インターフェース 54
定数 46
ネストした型 59
配列 47
パラメータの受け渡し 55
浮動小数点 45
モジュール 40
文字列 45
予約語 40
予約名 40
例外 57
ローカルインターフェース 54
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 119
機能の概要 9
マルチホームホスト 166
インターフェースの用法の指定 166

む

無効化
スマートエージェント 162

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
_get_policy 142
_is_a() 140
_is_bound() 140
_is_local() 140
_is_remote() 140
_set_policy_override メソッド 142
activate() 419
boa.obj_is_ready() 314
deactivate() 419
invoke() 313, 314
invoke() の実装のサンプル 314
ORB_init()
-ORBshmsize 34
string_to_object() 139
一方向 … の定義 156
状態を保持するオブジェクト 169
ステートレスオブジェクト，呼び出し 169
メソッドレベル承認
ネーミングサービス 212
メッセージ
vbj コマンドによるブロードキャスト 34
ブロードキャスト 161

も

モジュール
マッピング 40
文字列
オブジェクトリファレンスへの変換 139
マッピング 45
文字列化
object_to_string() メソッドの使用 139
文字列化された名前 186

ゆ

有効なポリシー 141
ユーザー例外
UserException クラス 86
キャッチするためのオブジェクトの変更 87
定義 86
フィールドの追加 87
フィールドへの追加 87
例外を生成するためのオブジェクトの変更 86
ユーザー例外の生成 86
ユーザー例外へのフィールドの追加 87
ユーティリティ
idl2ir 292
irep 291
osagent 21

よ

様態
オブジェクトリファレンスの … の判定 140
予約
キーワード 40
予約語
マッピング 40
予約名
マッピング 40

ら

ランタイム , クライアント 149

り

リクエストインターセプタ 321, 323
POA スコープ付きサーバー要求 330
ServerRequestInterceptor 324
インターセプトポイント 323, 324
サンプル 334, 338
リスナースレッド 120
リスナーのプロパティ 114
リファレンスデータ 282
リポジトリ ID
取得 139, 277
リモートメソッド起動 →「RMI」383

れ

例
oadutil unreg ユーティリティ 285
例外
CORBA 425
CORBA 定義のシステム例外 83
CORBA の概要 83
InvalidPolicy 414
Java IDL システム 58
システム 58
SystemException クラス 83
システム例外にナローイング 85
処理 85
生成 86
ヒューリスティック 429
マッピング 57
ユーザー定義 58
ユーザー例外のキャッチ 87
ユーザー例外へのフィールドの追加 87
例外の完了状態 84
例外のキャッチ
オブジェクトの変更 87
システム例外 85
ユーザー例外 87
例外を生成するためのオブジェクトの変更 86

ろ

ローカルインターフェース 54
ロケーションサービス 173
Agent インターフェース 175
エージェントのコンポーネント 175
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 161
トリガー 175, 177
プロパティ 69