Borland VisiBroker® 7.0

VisiTransact ガイド

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

使用権の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。適用される特許の一覧については、 製品 CD または ［バージョン情報］ダイアログ ボックスを参照してください。 このドキュメントの提供により、 これらの特許のいかなる使用権もユーザーに付与されるものではありません。

Copyright 1999–2006 Borland Software Corporation. All rights reserved.

Borland のブランド名および製品名はすべて、 米国 Borland Software 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。その他の商標は、 その所有者に帰属します。

Microsoft、 .NET ロゴ、および Visual Studio は米国 Microsoft Corporation の米国およびその他の国における登録商標または商標です。

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。

VB70Transact
2006 年 3 月

Borland®

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
VisiTransact の基礎 7
VisiTransact の概要 7
VisiTransact のアーキテクチャ 8
VisiTransact Transaction Service 8
データベース統合（Solaris のみ） 8
VisiBroker コンソール 9
VisiBroker ORB 9
VisiTransact の機能 10
VisiTransact CORBA 準拠 10
監視ツール 10
少ないフットプリントで最小限のオーバーヘッド 10
柔軟なデプロイメント 10
公開トランザクション処理標準のサポート . . . 11
マルチスレッドの完全なサポート 11
OMG 仕様の拡張機能 11
VisiTransact と CORBA サービス仕様 11

第 3 章
トランザクション処理の概要 13
分散環境におけるトランザクションについて . . . 13
CORBA の概要 14
CORBA のトランザクションサービスについて . . . 14
基本的なトランザクションのモデル 15
トランザクションの開始 16
トランザクションオブジェクトへの要求の発行 . 16
トランザクションの完了 17

第 4 章
VisiTransact の C++ クイックスタート 19
サンプルの概要. 19
C++ クイックスタートサンプルのファイル. 20
サンプルを実行するための前提条件 21
このサンプルの学習手順 21
クイックスタート IDL の記述 22
トランザクションオリジネータの記述（transfer クライアントプログラム） 23
ORB の初期化 23
Bank オブジェクトへのバインド 23
トランザクションの開始 24
トランザクションオブジェクト（送金元と送金先の Account）へのリファレンスの取得 25
トランザクション（Account）オブジェクトのメソッドの呼び出し（debit() および credit()）. . . . 26
トランザクションのコミットまたはロールバック 26
例外処理. 27
bank_server プログラムの記述 27
Bank オブジェクトの記述 29
BankImpl クラスの階層 29
Bank オブジェクトと get_account() メソッドの実装 29
トランザクションオブジェクト（Account）の記述 . 31
AccountImpl クラスの階層 31
Account オブジェクトをトランザクションオブジェクトにする. 31
Account オブジェクトとそのメソッドの実装. . . 32
サンプルのビルド 34
Makefile の選択. 34
make によるサンプルのコンパイル 34
サンプルの実行 35
スマートエージェント（osagent）の起動 35
VisiTransact Transaction Service の起動 35
storage_server プログラムの起動. 35
bank_server プログラムの起動. 35
トランザクションオリジネータの実行（transfer クライアントプログラム）. 35
結果 36
完全なサンプルコード. 36
クイックスタートサンプルの IDL 36
transfer クライアントプログラム 37
bank_server プログラム 39
Bank オブジェクトと Account（トランザクション）オブジェクト 40

第 5 章
トランザクションオブジェクトの作成 45
トランザクション対応オブジェクトインターフェースの継承 45
トランザクション対応オブジェクトインターフェースの実装 45
トランザクション対応 POA ポリシーインターフェース 46
OTSPolicy 46
InvocationPolicy 46
NonTxTargetPolicy 46
影響を受けるサーバーの動作 46
影響を受けるクライアントの動作 47
非共有（UNSHARED）トランザクションの処理 . . 47

第 6 章
トランザクションの構築方法の決定 49
トランザクション管理の方法 49

直接的なコンテキスト管理と間接的なコンテキスト管理 49
暗黙的な伝達と明示的な伝達 50
コンテキストの管理と伝達 51
暗黙的な伝達による間接的なコンテキスト管理51
明示的な伝達による間接的なコンテキスト管理51
暗黙的な伝達による直接的なコンテキスト管理51
明示的な伝達による直接的なコンテキスト管理51
インプロセスとアウトプロセスの VisiTransact Transaction Service. 52
マルチスレッド 52
既存のアプリケーションとトランザクションシステムの統合 52
構築方法の組み合わせ 53
Web 用のトランザクションの実装 53
C++ VisiTransact アプリケーションの構築. 53
VisiTransact Transaction Service スタンドアロンインスタンスの使用. 53
アプリケーションへの VisiTransact Transaction Service インスタンスの埋め込み. 54
VisiTransact Transaction Service 埋め込みインスタンスへのバインド. 55
VisiTransact から提供されるヘッダーファイルの使い方 55

第 7 章

VisiTransact 管理のトランザクションの作成および伝達 57

VisiTransact 管理のトランザクションで使用される Current の概要 57
Current の機能 58
Current オブジェクトリファレンスの取得 59
Current インターフェースとそのメソッドの使い方59
同じトランザクションに関与する複数のスレッド 61
コンテキストまたはスレッド内の複数トランザクションの使用. 61
VisiTransact Transaction Service のインスタンスの検索 62
VisiTransact 管理のトランザクションの伝達. 63
トランザクションが実行中かどうかの確認 63
ロールバックするトランザクションのマーク 64
トランザクション情報の取得 65
Current インターフェースの拡張機能 65

第 8 章

トランザクションを作成および伝達するほかの方法 67

はじめに 67
TransactionFactory によるトランザクションの作成. . 68
Control オブジェクトによるトランザクションの制御 69
オリジネータからの明示的なトランザクションの伝達 70
明示的な伝達から暗黙的な伝達への変更 71
Current からの明示的なコンテキストの取得. 72
Terminator によるトランザクションのコミットまたはロールバック 72
ロールバックするトランザクションのマーク 73
トランザクション情報の取得 73

第 9 章

トランザクションの完了 75

トランザクションの完了 75
VisiTransact Transaction Service が完了を実行するしくみ 75
VisiTransact Transaction Service が checked behavior を実行するしくみ. 76
ヒューリスティックな完了. 78
アプリケーションへのヒューリスティック情報の通知 79
OTS の例外. .80

第 10 章

リソースオブジェクトによるトランザクションの完了の調整 81

トランザクションの完了の概要. 81
トランザクションの完了への参加 82
リソースオブジェクトがトランザクションに登録される 83
トランザクションオリジネータがトランザクションの完了を開始する. 83
Terminator がリソースオブジェクトに準備を指示する 84
リソースオブジェクトが Terminator に提案を戻す 84
Terminator がコミットするかロールバックするかを決定する 85
リソースオブジェクトがトランザクションをコミットする 85
2 フェーズコミットのまとめ 86
1 フェーズコミットのまとめ 86
ロールバックのまとめ 86
障害後のトランザクション回復への関与. 87

第 11 章

ヒューリスティックな決定の管理 89

ヒューリスティックな決定の概要 89
heuristic.log ファイルの概要. 90
ヒューリスティックログの解釈. 91
問題を特定した後の処理 92

第 12 章

同期オブジェクトの実装 93

同期オブジェクトについて. 93
コミットプロトコルの前に同期オブジェクトを使用する 94
ロールバックやコミットの後で同期オブジェクトを使用する 94
同期オブジェクトの登録 94
障害が同期オブジェクトに及ぼす影響. 95
トランザクションオブジェクトにおける同期オブジェクトの役割 .95

第 13 章

下位互換性と移行 97

下位互換性 97
OTS1.1 クライアント対 OTS1.2 サーバー. 97

OTS1.1 サーバー対 OTS1.2 クライアント 97
移行 98

第 14 章
セッションマネージャの概要 99

データベースの VisiTransact アプリケーションへの統合方法 99
セッションマネージャの概要 100
データベースへの接続を開く 100
接続プロファイル 101
接続の設定 101
接続とトランザクションの関連付け 101
リソースの登録 101
接続の解放 102
接続のプール 102
スレッド要件の管理 102
XA プロトコルを使ったグローバルトランザクション 103
XA リソースディレクタの概要 103
分散トランザクションの回復 104
DirectConnect セッションマネージャ 104
リソースの登録 105
デプロイメントに関する問題 105
DirectConnect アクセストランザクションの制約 . .106
DirectConnect と XA アクセストランザクションの共存 106

第 15 章
セッションマネージャを使用した VisiTransact とデータベースの統合 107

XA を使って VisiTransact をデータベースと統合することによる影響を評価する 108
XA の使用によるオーバーヘッドの増加 108
高可用性の必要 108
ロック中または利用できないデータ 108
部分的な制御の取得 109
DirectConnect を使って VisiTransact をデータベースと統合することによる影響を評価する 109
データベースの準備 109
接続プロファイルセット 110
セッションマネージャクライアントが使用する接続プロファイルの変更 110
XA リソースディレクタが使用する接続プロファイルの変更 110
XA リソースディレクタの使用 111
XA リソースディレクタのデプロイメント . . .111
XA リソースディレクタの起動 111
XA リソースディレクタによる接続プロファイルの使用方法 112
クライアント側ライブラリのデプロイメント . .112
XA リソースディレクタをリモートからシャットダウンする 112
XA リソースディレクタを OAD に登録する . . .112
セッションマネージャベースのアプリケーションプロセスの起動 113
永続的ストアファイルのデフォルトパスの確認 .113
ヒューリスティックの適用 114
パフォーマンスのチューニング 114
XA のチューニング 114
セッションマネージャ設定サーバー 114
永続的ストアファイルのディレクトリ構造 . . .114
永続的ストアファイルのデプロイメント 116
オプション 1：共有ファイルシステム上の永続的ストアファイル 116
オプション 2：各ノード上の永続的ストアファイル 116
オプション 3：各ノードにコピーされた永続的ストアファイルのセット 116
手動によるセッションマネージャ設定サーバーの起動 117
設定サーバーのシャットダウン 117
セキュリティ 117

第 16 章
セッションマネージャを使用したデータアクセス 119

統合の準備 119
セッションマネージャの使い方：手順のまとめ . . .120
ConnectionPool オブジェクトリファレンスの取得 . .120
ConnectionPool オブジェクトリファレンスの使用 121
Connection プールからの Connection オブジェクトの取得 121
明示的なトランザクションコンテキストの使用 .121
接続プールの最適化 122
ネイティブ接続ハンドルの取得 122
ネイティブ接続ハンドルの使用 122
スレッドの要件 122
接続の解放 123
Connection インスタンスの割り当て解除 124
例外の表示 124
属性の表示 125
セッションマネージャ情報の取得 126
hold() と resume() の使用 126
hold() の使い方 127
resume() の使い方 127
簡単な統合の例 128
XA インプリメンテーションに関する問題 129
トランザクションの完了または回復 129
DirectConnect インプリメンテーションに関する問題129
トランザクションの完了または回復 130
DirectConnect から XA への変更 130

第 17 章
VisiTransact 向けプラグイン可能データベースリソースモジュール 131

概念 131
プラグイン可能データベースリソースモジュールの概要 131
構造の説明 132
接続管理 .133
プラグイン可能モジュールの作成 136
接続プロファイル 136
インターフェースの定義 137
唯一の関数 137
ITSDataConnection クラス 137
ネイティブハンドル取得インターフェース . .138

ローカルトランザクション接続／完了インターフェース 138
グローバルトランザクション接続／完了インターフェース 138
ビルドと実行 139
プラグイン可能モジュールを使用したアプリケーションの実行 139
プログラミングの制限事項 139
既知の制限 140

第 18 章

VisiBroker コンソールの使い方 141

VisiBroker コンソールの概要 141
［Transaction Services］セクション 141
［Session Manager Profile Sets］セクション . . . 142
VisiBroker コンソールの起動 142
VisiTransact Transaction Service の起動 142
セッションマネージャ設定サーバーの起動 . . 142
VisiBroker コンソールの起動 142
［Transaction Services］セクションの使用 143
トランザクションサービスのインスタンスの検索 143
トランザクションの監視 143
トランザクションリストの再表示 144
特定のトランザクションの詳細の表示 144
特定のトランザクションの制御 144
ハングまたは未確定トランザクションの解決 145
トランザクションリストのフィルタリング . . 145
ヒューリスティックによるトランザクションの表示 145
ヒューリスティックの詳細の表示 146
メッセージログの表示 146
メッセージログのフィルタリング 146
メッセージログの調整 147
［Session Manager Profile Sets］セクションの使用 . 147
接続プロファイルとは 147
セッションマネージャ設定サーバーへのアクセスの取得 148
新しい接続プロファイルの作成と設定 148
既存の接続プロファイルの編集 149
接続プロファイルのフィルタリング 149
接続プロファイルの削除 149
接続プロファイルのリストの更新 149

第 19 章

サーバーアプリケーションモデル 151

サーバーアプリケーショントランザクションとデータベース管理 151
このセクションの理解に必要な知識 152
概念と用語 152
グローバルトランザクションおよび PMT のシナリオ 153
クライアント開始グローバル 2PC および 1PC トランザクション 153
PMT による透過的なサーバー初期化トランザクション 154
PMT の概要 155
PMT トランザクション属性値 156
簡単な例 158
PMT::Current および接続名 159
XA リソースの設定 160
xa-resource-descriptor 160
xa-resource 160
xa-connection 161
xa-resource-alias 162
XA リソース記述子の例 163
VisiTransact のプロパティ 164
vbroker.its.its6xmode=<false|true>. 164
vbroker.its.verbose=<false|true> 164
vbroker.its.xadesc=<xa-resource xml file name> . . . 164
RM リカバリユーティリティ 165

第 20 章

XA Session Manager for Oracle OCI、version 9i Client 167

概要 167
この章の想定読者 168
Oracle9i のソフトウェア要件 168
クライアントの要件 168
サーバーの要件 168
Oracle9i のインストールと設定の問題 169
インストール要件 169
データベース設定 169
DBA_PENDING_TRANSACTIONS ビュー . . . 169
必要な環境変数 170
セッションマネージャの接続プロファイル属性 . . 170
OCI 9i API でのセッションマネージャの使用 . . . 171
プログラミングの制限事項 171
トラブルシューティング 172
VisiTransact メッセージログ 172
xa_trc ファイルの使い方 172
分散更新の問題 172
データアクセス障害 172
未確定トランザクションによるロック . . . 172
トランザクションタイムアウト 173
Oracle エラーメッセージ 173
ヒューリスティックな完了の強制 173

第 21 章

DirectConnect Session Manager for Oracle OCI、version 9i Client 175

概要 175
この章の想定読者 176
Oracle9i のソフトウェア要件 176
クライアントの要件 176
サーバーの要件 176
Oracle9i のインストールと設定の問題 176
インストール要件 176
データベース設定 177
必要な環境変数 177
セッションマネージャの接続プロファイル属性 . . 177
OCI 9i API でのセッションマネージャの使用 . . . 178
プログラミングの制限事項 178
トラブルシューティング 178
VisiBroker VisiTransact メッセージログ 178
Oracle エラーメッセージ 179

第 22 章

コマンド、ユーティリティ、引数、および環境変数 181

VisiTransact コマンドの概要 181
vbconsole. 182
構文 182
サンプル 182
引数 182
ots 182
構文 182
サンプル 182
引数 182
smconfig_server 183
構文 183
サンプル 183
引数 183
vshutdown 184
構文 184
サンプル 184
引数 184
xa_resdir 185
構文 185
サンプル 185
引数 185
VisiTransact ユーティリティ 186
smconfigsetup 186
セッションマネージャで使用するプロファイルを作成する 186
アプリケーションのコマンドライン引数. 187
argc と argv を使用して、コマンドライン引数を ORB_init() に渡す 187
トランザクションを開始するアプリケーションの引数. 187
VisiTransact Transaction Service インスタンスが埋め込まれたアプリケーションの引数 188
セッションマネージャを使用するアプリケーションの引数. 189
環境変数 190

第 23 章

エラーコード 191

VisiTransact の一般的なエラーコード 191
VisiTransact トランザクションサービスのエラーコード 192
セッションマネージャのエラーコード 195
VisiTransact トランザクションログのエラーコード . 200

第 24 章

問題の判定 201

一般的な方法 201
トランザクションの問題の処理 201

索引 203

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デプロイメントで使用されます。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。

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

VisiTransact の基礎

ここでは、VisiBroker VisiTransact を紹介します。VisiBroker VisiTransact は、インターネットまたはイントラネットを介して CORBA アプリケーションを使用するトランザクションへの完全な C++ トランザクション管理ソリューションです。以下では、VisiTransact の機能とアーキテクチャコンポーネントについて説明します。

VisiTransact の概要

VisiTransact は、分散トランザクション CORBA アプリケーションに完全なソリューションを提供します。VisiTransact は VisiBroker ORB 上に実装され、基本的なサービスを提供して、分散トランザクションを単純化します。提供されるサービスには、トランザクションサービス、回復、ログ、データベースおよび既存のシステムとの統合（Solaris プラットフォームのみ）、管理機能などがあります。

VisiTransact は OMG OTS 1.2 準拠のトランザクションサービス機能である VisiTransact Transaction Service を提供します（C++ 版のみ）。Solaris プラットフォームでは、VisiTransact は統合されたトランザクションサービス（ITS）を提供します。これには、XA リソースディレクタ、セッションマネージャ設定サーバー、Oracle9i 向けセッションマネージャ、選択したデータベースをセッションマネージャが操作できるようにするためのプラグイン可能リソースインターフェースが含まれます。

メモ VisiTransact では、Java トランザクション型アプリケーションを作成する機能は提供されていません。この機能は、Borland AppServer で提供されます。

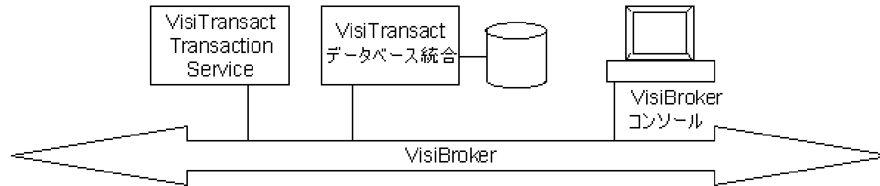
VisiTransact のアーキテクチャ

VisiTransact には、分散トランザクションの管理に対する完全なソリューションを提供する次のコンポーネントがあります。

- VisiTransact Transaction Service
- データベース統合（Solaris のみ）
- VisiBroker コンソール

次の図に示されているように、VisiTransact は VisiBroker ORB 上に実装され、分散トランザクション対応 CORBA アプリケーションで使用されるコンポーネントを提供します。

図 2.1 VisiTransact のアーキテクチャ



VisiTransact Transaction Service

VisiTransact Transaction Service は、トランザクションの完了を管理します。最終的な OMG トランザクションサービス仕様バージョン 1.2 文書に準拠しています。

VisiTransact は、1 つの共有ライブラリと 1 つの実行可能ファイルとして提供されます。この柔軟なアーキテクチャにより、VisiTransact Transaction Service のインスタンスをスタンドアロンプロセスとしてデプロイメントしたり、C++ アプリケーションに埋め込むことができます。ネットワーク上で VisiTransact Transaction Service の複数のインスタンスを使用すると、トランザクションの負荷を分散できます。

VisiTransact は osagent（スマートエージェント）を使用して、トランザクションサービスのインスタンスを起動し、インスタンスの数を 1 つだけに保ちます。また `vshutdown` ユーティリティも、トランザクションサービスの検索とそのシャットダウンに osagent を使用します。

データベース統合（Solaris のみ）

データベース統合コンポーネントを使用して、Solaris プラットフォーム上のトランザクション型アプリケーションをデータベースやほかのリソースマネージャに統合できます。データベース統合には、次のコンポーネントがあります。

- **XA セッションマネージャインプリメンテーション（Oracle9i のみ）。**アプリケーションは、セッションマネージャの XA インプリメンテーションを使用することにより、Oracle9i データベースへの VisiTransact 対応の接続を取得できます。セッションマネージャは、すべての XA 呼び出しを処理し、VisiTransact Transaction Service がリソース間のトランザクションを調整できるようにします。また、データベース接続プールも提供します。さらに、セッションマネージャ設定サーバーを使用すると、VisiBroker コンソールを使って接続プロファイルを作成できます。
- **DirectConnect セッションマネージャインプリメンテーション（Oracle9i のみ）。**セッションマネージャの DirectConnect インプリメンテーションは、リソースへの非 XA アクセスを提供します。これは、1 フェーズリソースが埋め込まれたセッションマネージャを含む単一のアプリケーションサーバープロセスで構成されます。このアーキテクチャは、1 フェーズコミットを実行するため、パフォーマンスが向上します。
- **セッションマネージャ設定サーバー。**セッションマネージャ設定サーバーを使用すると、VisiBroker コンソールを使ってセッションマネージャの接続プロファイルを作成できます。セッションマネージャ設定サーバーについては、114 ページの「セッションマネージャ設定サーバー」を参照してください。

- **プラグイン可能リソースインターフェース。**Pluggable Resource Interface では、選択したデータベースで動作するセッションマネージャを有効にできます。このコンポーネントに実装される定義済みインターフェースのセットを使用すると、トランザクション型アプリケーションは、VisiTransact によって管理されるトランザクションでの永続的ストレージとして Oracle9i 以外のデータベースを使用できます。プラグイン可能リソースインターフェースについては、131 ページの「VisiTransact 向けプラグイン可能データベースリソースモジュール」を参照してください。
- **XA リソースディレクタ。**XA リソースディレクタは、1 つ以上のトランザクションに関与している特定のリソースマネージャとのすべての対話を管理します。特定のリソースマネージャ（Oracle9i データベースなど）を使用して、すべてのトランザクションをネットワーク上で処理します。XA リソースディレクタは、VisiTransact と X/Open トランザクション環境の間を*仲介します*。これにより、VisiTransact のリソースモデルと X/Open 分散トランザクションプロトコル（DTP）のリソースマネージャモデルの間で相互運用性が実現します。VisiTransact のリソースモデルについては、103 ページの「XA リソースディレクタの概要」を参照してください。

VisiBroker コンソール

VisiBroker コンソールは、ネットワークを介して分散トランザクションを管理したり、特定のデータベースで使用する接続プロファイルを設定するために使用されるグラフィカルツールです。また、コンソールを使用して、トランザクションの状態や完了を監視および制御できます。コンソールを使用すると、セッションマネージャ設定サーバーでセッションマネージャ接続プロファイルを作成できます。VisiTransact の VisiBroker コンソールについては、141 ページの「VisiBroker コンソールの使い方」を参照してください。

VisiBroker ORB

VisiBroker ORB は、アプリケーションが VisiTransact を使って分散トランザクションを管理するための機能とインプリメンテーションを提供します。この ORB は、スレッドプーリング、接続の多重化と再利用、負荷分散、フォールトトレランスなどの多くの機能を VisiTransact アプリケーションに提供します。これらの機能の多くは、トランザクションプロセスモニタの一部になります。

VisiTransact は、VisiBroker ORB の強力な機能である OMG ポータブルインターセプタを使用して、ORB の機能を実装します。また、VisiTransact のユーザーは、インターセプタを利用して、トランザクション型アプリケーションをカスタマイズできます。

VisiTransact の機能

VisiTransact は、1 フェーズまたは 2 フェーズのコミットプロトコルにより、フラットなトランザクションの完了を管理します。トランザクションに関与するリソースが 1 つしかない場合は、1 フェーズコミットプロトコルが使用されます。

VisiTransact は、CORBA のトランザクションサービス仕様に記述されている分散トランザクション管理機能だけでなく、この仕様の拡張機能も提供します。以下では、これらの拡張機能などについて説明します。

VisiTransact CORBA 準拠

VisiTransact は、OMG（オブジェクトマネージメントグループ）の CORBA 2.6 仕様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。

VisiTransact は、OMG のトランザクションサービスバージョン 1.2 の CORBA サービス仕様にも準拠しています。11 ページの「VisiTransact と CORBA サービス仕様」には、この仕様で規定されたオプションに対する VisiTransact の決定内容の一覧が記載されています。

監視ツール

コンソールを使用すると、トランザクションの状態と完了を監視および制御したり、ログファイルのサイズや場所を管理することができます。

少ないフットプリントで最小限のオーバーヘッド

システムの必要条件に応じて、VisiTransact Transaction Service のインスタンスを必要な数だけネットワーク上に配置できます。ただし、VisiTransact Transaction Service を環境内のすべてのホストマシンに置く必要はありません。

セッションマネージャ（Solaris でのみ使用可能）も、データベース接続をプールし、要求に応じて接続を再利用することにより、システムリソースを節約できます。

柔軟なデプロイメント

VisiTransact では、デプロイメントを最適化するために次の 3 つの方法が用意されています。

- VisiTransact Transaction Service にアプリケーション／ビジネスオブジェクトを直接リンクします。
- VisiTransact Transaction Service と同じマシンにアプリケーション／ビジネスオブジェクトをデプロイメントします。
- VisiTransact Transaction Service の場所に関係なく、任意のマシンにアプリケーション／ビジネスオブジェクトをデプロイメントします。

スケーラビリティとフォールトトレランスを求める場合は、ビジネスオブジェクトの複数のインスタンスと VisiTransact Transaction Service の複数のインスタンスを複数のマシンにデプロイメントできます。

Solaris プラットフォームでは、Oracle9i データベースが 1 つだけ存在する場合、以下を 1 つのプロセスにリンクすると、パフォーマンスがさらに向上します。

- アプリケーションコード
- VisiTransact Transaction Service
- セッションマネージャ（Solaris プラットフォーム上の Oracle9i データベースの場合のみ）

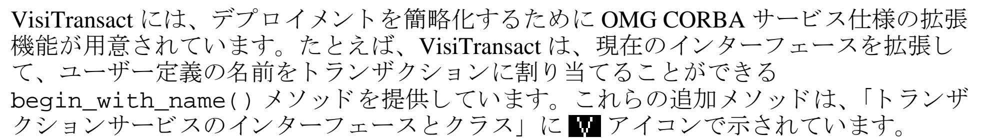
公開トランザクション処理標準のサポート

現在、VisiTransact は、OMG の CORBA サービストランザクションサービスと XA プロトコル公開トランザクション処理標準をサポートしています。

マルチスレッドの完全なサポート

VisiTransact はマルチスレッドをサポートします。したがって、ビジネスオブジェクトをマルチスレッド化して、複数の要求を同時に処理できます。

OMG 仕様の拡張機能



VisiTransact には、デプロイメントを簡略化するために OMG CORBA サービス仕様の拡張機能が用意されています。たとえば、VisiTransact は、現在のインターフェースを拡張して、ユーザー定義の名前をトランザクションに割り当てることができる `begin_with_name()` メソッドを提供しています。これらの追加メソッドは、「トランザクションサービスのインターフェースとクラス」に V アイコンで示されています。

VisiTransact と CORBA サービス仕様

次の表は、CORBA サービス仕様のいくつかのオプションが VisiTransact でどのように実装されているかを示します。

オプション	VisiTransact での決定
トランザクションサービスのインプリメンテーションでは、同期をサポートしなくてもよい	VisiTransact は同期オブジェクト（Synchronization インターフェース）を完全にサポートします。
トランザクションサービスのインプリメンテーションがトランザクションコンテキストの使用範囲を制限した場合は、Unavailable 例外が生成される	VisiTransact はトランザクションコンテキストの使用範囲を制限しないため、Unavailable 例外は生成されません。
トランザクションサービスのインプリメンテーションで、すべての要求ハンドラのトランザクションコンテキストを初期化する必要はない	VisiTransact のデフォルトの動作では、`TransactionalObject` から派生したインターフェースをサポートするオブジェクトについてのみ、トランザクションコンテキストの初期化が行われます。ただし、すべての要求のトランザクションコンテキストを初期化するようにも設定できます。57 ページの「VisiTransact 管理のトランザクションの作成および伝達」を参照してください。
トランザクションサービスのインプリメンテーションでは、トランザクションの整合性を保証するために、Coordinator、Terminator、および Control オブジェクトの一部または全部をほかの実行環境に送信したり、ほかの実行環境で使用できないように制限してもよい	VisiTransact では、一切の制限なしに、Coordinator、Terminator、および Control オブジェクトをほかの実行環境に送信したり、ほかの実行環境で使用することができます。checked behavior の実行方法については、75 ページの「VisiTransact Transaction Service が完了を実行するしくみ」を参照してください。

オプション	VisiTransact での決定
トランザクションサービス自身がトランザクションに関与する要素について障害や非アクティブ状態を監視するかどうかは、インプリメンテーションに依存する。トランザクションサービスの一部のインプリメンテーションでは、トランザクションサービスインターフェースの使用を制限することにより、X.Open DTP トランザクションモデルをサポートするインターフェースが提供するものと同等の整合性を保証してもよい。これは「checked behavior」と呼ばれる	VisiTransact は制限を設けませんが、VisiTransact によって管理されるトランザクションでは checked behavior がサポートされます。詳細については、76 ページの「VisiTransact Transaction Service が checked behavior を実行するしくみ」を参照してください。
トランザクションサービスの一部のインプリメンテーションでは、トランザクションを作成したトランザクションサービスクライアント以外のクライアントがトランザクションを終了することを許してもよい	VisiTransact では、トランザクション（VisiTransact によって管理されないトランザクションなど）の `Terminator` インターフェースを呼び出すことで、任意のオブジェクトからトランザクションを終了できます。なお、トランザクションを作成したクライアントスレッドへの `Current` インターフェースが使用されている場合は、トランザクションの終了が制限されます。
TransactionFactory は、ORB の `resolve_initial_references()` オペレーションではなく、存続期間サービスの FactoryFinder インターフェースによって検索される	VisiTransact の TransactionFactory を検索するには、`bind()` メソッドなどの VisiBroker ORB の検索機能を使用します。
トランザクションサービスのインプリメンテーションでは、オプションでイベントサービスを使用して、ヒューリスティックな決定を通知してもよい	VisiTransact がヒューリスティックな決定を通知するためにイベントサービスを使用することはありません。
トランザクションサービスのインプリメンテーションでは、ネストしたトランザクションをサポートしなくてもよい	現時点では、主要なデータベースはネストしたトランザクションをサポートしていません。そのため、VisiTransact は、ネストしたトランザクションをサポートしていません。

第 3 章

トランザクション処理の概要

ここでは、トランザクションとそれが処理される方法の概要を示します。トランザクション、CORBA、CORBA トランザクションサービスのコンポーネント、および基本的なトランザクションのプロセスについて説明します。

分散環境におけるトランザクションについて

分散オブジェクト環境のトランザクションは、オブジェクトに対する一連の操作をまとめた作業単位です。トランザクションの例として、ある銀行口座から別の銀行口座にお金を振り込む場合を考えます。振り込みは、一方の口座から引き出し、もう一方の口座に預け入れるという 2 つの動作からなる 1 つのトランザクションです。

図 3.1　トランザクションの例

このフラットなトランザクションモデルの例で、目的の結果を実現するには、両方の動作が完了する必要があります。動作 1 が完了して動作 2 が完了しなければ、顧客がお金を失います。動作 1 が完了しないで動作 2 が完了すれば、銀行がお金を失います。フラットなトランザクションには、完全な成功と完全な失敗という 2 種類の結果しかありません。つまり、トランザクションは、すべての手順が完了するか、すべての手順が完了しないかのいずれかである必要があります。

メモ　ネストしたトランザクションという別のタイプのトランザクションがあり、このトランザクションでは、一部の手順が完了しなくもかまいません。ただし、VisiTransact Transaction Manager は、ネストしたトランザクションをサポートしていません。

トランザクションの一部の手順の完了を妨げるものとして、アプリケーションロジックの誤り、サーバーの障害、ハードウェアの障害、ネットワーク割り込みなど、さまざまな原因が考えられます。これらの予測できない環境要因に関係なくアプリケーションの一貫性、信頼性、および整合性を維持するために、トランザクションは、ACID プロパティと呼ばれる次の性質を備えている必要があります。

- **原子性（Atomicity）**トランザクションが正常に完了した場合は、トランザクションに関連付けられているすべての動作が実行されます。つまり、トランザクションが*コミット*されます。トランザクションが正常に完了しなかった場合は、どの動作も実行されません。つまり、トランザクションが*ロールバック*されます。
- **一貫性（Consistency）**システムが一貫した状態から別の一貫した状態に移るために、トランザクションを構成するすべての動作が正確に実行される必要があります。先の銀行の例で考えると、トランザクションを開始する前の 2 つの銀行口座の合計金額と、トランザクションが完了した後の 2 つの銀行口座の合計金額は同じである必要があります。
- **分離性（Isolation）**トランザクション内の中間結果は、トランザクション全体が完了するまでトランザクションの外部から見えません。
- **耐久性（Durability）**トランザクションの結果は永続的です。

トランザクションは、銀行の例で紹介した送金に関するものだけではありません。トランザクションは、あらゆる種類のビジネス活動に必要です。たとえば、オンライン書店は、多くの処理をトランザクションとして行う必要があります。たとえば、出版社への書籍の注文、出版社からの書籍の納入、書籍の正確な在庫の更新、購入者への請求、注文の受け付けなどがあります。これらの多くの動作をトランザクションとして実行する必要があります。

CORBA の概要

CORBA（Common Object Request Broker Architecture）仕様は、分散オブジェクトを実装および管理する共通の方法を確立するために、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。各オブジェクトが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だけを提示します。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。

CORBA のトランザクションサービスについて

OMG の定義による CORBA トランザクションサービスは、トランザクションの整合性を提供することで、ミッションクリティカルなアプリケーションを分散環境内で実行できるようにします。そのために、複数の分散オブジェクトがトランザクションに参加したり、分散アプリケーションがインターネットやイントラネットを介してトランザクションを処理できるようにするための IDL インターフェースが定義されています。

Borland の CORBA トランザクションサービスは、VisiTransact Transaction Service（トランザクション管理アーキテクチャのコンポーネント）として実装されています。

基本的なトランザクションのモデル

VisiTransact Transaction Service を使用して、トランザクションの完了を*管理*できます。VisiTransact Transaction Service は ORB レベルでオブジェクトを操作して、トランザクションのコミットやロールバックを調整および管理します。ORB は、VisiTransact Transaction Service がトランザクションに関与する各オブジェクトに*トランザクションコンテキスト*を伝達できるようにします。そのために、VisiTransact Transaction Service は、トランザクション管理プロセスの特定の時点で、トランザクションに関与するオブジェクトと対話します。

分散アプリケーションでは、複数の要求を実行する複数のオブジェクトが 1 つのトランザクションに関与する場合があります。関与するオブジェクトは、異なるさまざまな役割を果たすことができます。トランザクションを開始したオブジェクトは、*トランザクションオリジネータ*と呼ばれます。次の表で、これらの役割について説明します。

参加者の役割	説明
トランザクション対応クライアント	トランザクション対応クライアントは、トランザクション型アプリケーションに対するユーザーのインターフェースです。トランザクション対応クライアントがトランザクションオリジネータになる場合もあります。
トランザクションオリジネータ	トランザクションオリジネータは、トランザクションを開始するオブジェクトです。 トランザクションオリジネータはトランザクション対応クライアントであるとは限りません。トランザクションサーバーがトランザクションを開始する場合もあります。
トランザクションオブジェクト	トランザクションオブジェクトは、トランザクションによって動作が左右されるが、それ自体では回復可能な状態を持たないオブジェクトです。トランザクションオブジェクトはトランザクションの完了に関与しませんが、トランザクションにロールバックを強制できます。 回復可能なオブジェクト（回復可能な状態がトランザクションに左右されるオブジェクト）の詳細については、81 ページの「リソースオブジェクトによるトランザクションの完了の調整」を参照してください。
トランザクションサーバー	トランザクションサーバーは、1 つ以上のトランザクションオブジェクトの集まりです。

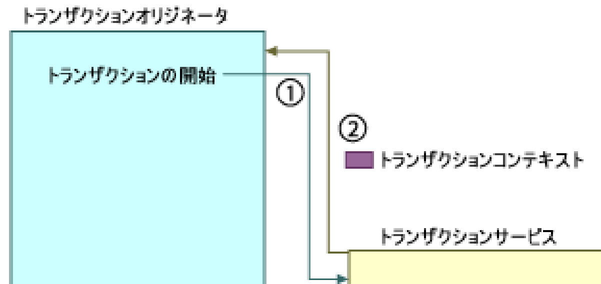
トランザクションオリジネータがトランザクションを開始すると、VisiTransact Transaction Service がアプリケーションと対話して、トランザクション情報をトランザクションオブジェクトに伝達します。最後にすべてのオブジェクトが調整され、トランザクションはコミットまたはロールバックによって完了します。

ここでは説明していませんが、ほとんどのトランザクションには、データベースなどの永続的データが関係しています。このようなトランザクションの場合は、ほかに「リソース」および「回復可能なサーバー」という 2 つの参加者の役割があります。これらの役割については、81 ページの「リソースオブジェクトによる トランザクションの完了の調整」で説明します。

トランザクションの開始

あるオブジェクトがトランザクションを開始すると、VisiTransact Transaction Service のインスタンスは、トランザクションオリジネータのためにトランザクションを開始し、トランザクションコンテキストを確立します。このトランザクションコンテキストは、VisiBroker ORB によって生成されたオリジネータの制御用スレッドに関連付けられます。トランザクションコンテキストには、トランザクションを一意に識別するオブジェクトトランザクション識別子（OTRID）などのトランザクション情報が格納されます。

図 3.2　トランザクションの開始

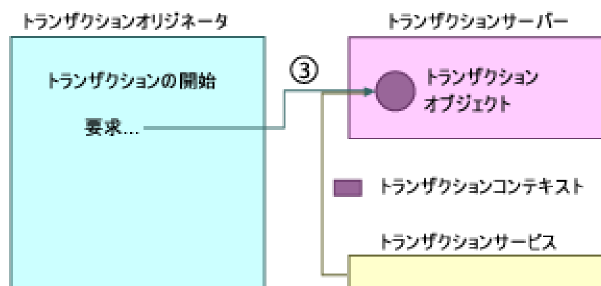


上の図のステップ 1 で、トランザクションオリジネータは、トランザクションの開始要求を VisiTransact Transaction Service に登録します。ステップ 2 で、VisiTransact Transaction Service は、トランザクションオリジネータにトランザクションコンテキストを戻すことによって要求に応答します。

トランザクションオブジェクトへの要求の発行

ステップ 3 で、トランザクションオリジネータがトランザクションオブジェクトに要求を発行すると、これらの要求もトランザクションコンテキストに関連付けられます。VisiTransact Transaction Service は、ORB を使用して、トランザクションに関与するすべてのオブジェクトにトランザクションコンテキストを伝達します。

図 3.3　トランザクションオブジェクトへの要求の発行と、トランザクションコンテキストの伝達



メモ　トランザクションコンテキストは、GIOP のリクエストヘッダーおよび応答ヘッダーのサービスコンテキストとして伝達されます。これにより、伝達は完全に透過的に行われます。また、トランザクションサービスインプリメンテーション間の相互運用性に関する CORBA サービス仕様に準拠しています。

トランザクションの完了

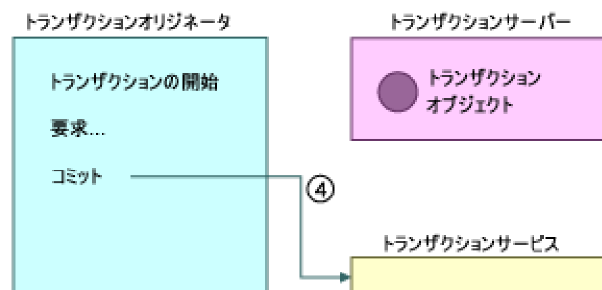
トランザクションは次のいずれかの形式で完了します。

- トランザクションオリジネータがトランザクションをコミットします。これが通常のシナリオです。
- Current が使用されていない限り、アプリケーション内の任意のコンポーネントがトランザクションを完了できます。
- トランザクションがタイムアウトになります。

コミットが要求され、関与するすべてのリソースがコミットに同意すると、変更がコミットされます。関与するリソースのいずれかがロールバックを提案すると、そのトランザクションはロールバックされます。

アプリケーションから完了が要求されない場合、VisiTransact Transaction Service は、タイムアウトになった時点でトランザクションをロールバックします。

図 3.4　トランザクションの完了



第 4 章

VisiTransact の C++ クイックスタート

ここでは、C++ のサンプルアプリケーションを使用して、VisiTransact を使用した分散オブジェクトベースのトランザクション型アプリケーションの開発について説明します。

サンプルの概要

C++ クイックスタートサンプルは、いくつかの口座を管理する銀行をモデルにしています。トランザクションの間に、クライアントプログラムに渡されるパラメータに基づいて、少なくとも 2 つの口座間で送金が行われます。

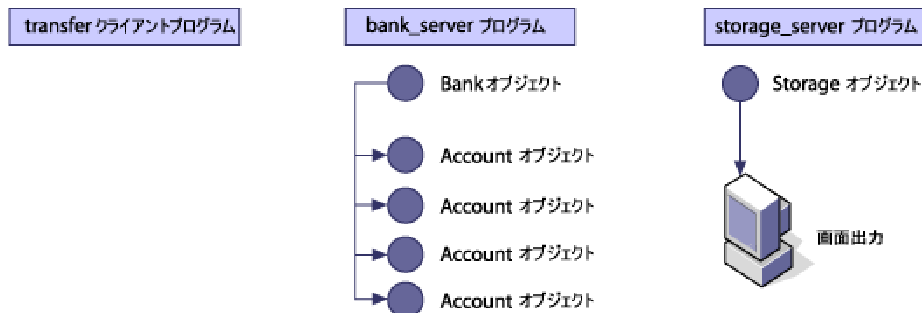
このクイックスタートには、次のプログラムがあります。

- **transfer**。このプログラムは、いくらのお金をどの口座間で送金するかをコマンドラインからの入力で受け取ります。次に、トランザクションを開始し、要求された送金を実行します。要求されたすべての送金が完了すると、トランザクションを終了（コミットまたはロールバック）するように要求します。
- **bank_server**。このプログラムは、Storage オブジェクトにバインドし、コマンドラインに入力された名前で Bank オブジェクトを作成します。
- **storage_server**。このプログラムは、データベースを使用しないクイックスタートのために Storage オブジェクトを実装します。トランザクション中に行われた残高の変更を永続的に格納するか（コミットされた場合）、残高をトランザクションの前の状態に戻します（ロールバックされた場合）。

このクイックスタートには、次のオブジェクトがあります。

- **Bank**。このオブジェクトは、既存の Account オブジェクトにアクセスします。また、Storage オブジェクト内に存在する口座に対する Account オブジェクトのインスタンスを作成します。
- **Account**。このオブジェクトを使用して、口座の残高を表示したり、口座に入金または出金を行うことができます。このオブジェクトは、Storage オブジェクトを使って永続的データとやり取りします。
- **Storage**。このオブジェクトの目的は、口座にかわってデータに変更を加えるためのデータアクセスを 1 つのオブジェクトに抽象化することです。
- **StorageServerImpl**。この Storage オブジェクトのインプリメンテーションには、メモリ上で残高を更新するだけの軽量のリソース（FakeResourceImpl）が含まれます。これは、VisiTransact を簡単に実行できるように用意されています。

図 4.1　クイックスタートサンプルのコンポーネント



C++ クイックスタートサンプルのファイル

VisiTransact パッケージの場所がわからない場合は、システム管理者に問い合わせてください。このサンプルに含まれているファイルを次の表に示します。

ファイル	説明
quickstart.idl	クイックスタートサンプルの IDL。オブジェクトに必要なインターフェースを定義します。
transfer.C	クライアントプログラム。ユーザーから入力を受け取り、VisiTransact 管理のトランザクションのオリジネータになります。トランザクションの一環としてトランザクションサーバーオブジェクトを呼び出します。
storage_server.C	Storage オブジェクトを作成するサーバープログラム。このサンプルの Storage オブジェクトは、メモリ上で残高を更新し、残高を出力する単純なインプリメンテーションです。storage_server は、簡単に実行できるように用意されています。
storage_server.h	Storage オブジェクトの仕様が記載されています。
bank_server.C	サーバープログラム。**storage_server** または **storage_ora** プログラムから受け取った情報を使って Bank オブジェクトを作成し、それをクライアントプログラムが使用できるようにします。
bank.h	Bank オブジェクトと Account オブジェクトの仕様が記載されています。
bank.C	Bank インターフェースと Account インターフェースのインプリメンテーションが格納されています。Bank オブジェクトは、トランザクションオブジェクト（Account オブジェクト）を作成します。Account オブジェクトは、Storage オブジェクトを呼び出して、口座への入金または出金を行うトランザクションオブジェクトです。
Makefile	すべてのテストターゲットをビルドするために使用します。
Makefile.cpp	すべてのテストターゲットをビルドするために使用します。
../itsmk	サポートされるプラットフォーム固有のメイク定義です。

メモ　可搬性を高めるため、これらのサンプルファイルには Windows と UNIX の両方で C 拡張子が使用されており、共通の `Makefile` を使用できます。

サンプルを実行するための前提条件

VisiTransact 製品と VisiBroker C++ Developer（ORB）をインストールする必要があります。また、VisiTransact Transaction Service のインスタンスを起動する必要があります。35 ページの「サンプルの実行」を参照してください。

このサンプルの学習手順

C++ クイックスタートは、次の手順で進めます。

1. トランザクション型アプリケーションに必要な 3 つのオブジェクト（Bank、Account、Storage）を定義する単純なインターフェースを IDL で実装します。22 ページの「クイックスタート IDL の記述」を参照してください。
2. クライアントプログラムとトランザクションオリジネータ（transfer）を実装します。手順としては、使用する口座と送金する金額に関する入力をユーザーから受け取ります。ORB を初期化し、トランザクションを開始します。Bank オブジェクトにバインドし、トランザクションオブジェクト（Account）へのリファレンスを取得します。トランザクションオブジェクト（Account）を使ってアクションを実行し、トランザクションをコミットまたはロールバックし、例外を処理します。23 ページの「トランザクションオリジネータの記述（transfer クライアントプログラム）」を参照してください。
3. **bank_server** プログラムを実装します。手順としては、ORB を初期化し、Bank オブジェクトを作成し、Storage オブジェクトにアクセスします。Bank オブジェクトを POA に登録し、要求を受信する準備を行います。27 ページの「bank_server プログラムの記述」を参照してください。
4. Bank を実装します。要求時にトランザクションオブジェクト（Account）をインスタンス化して戻します。29 ページの「Bank オブジェクトの記述」を参照してください。
5. トランザクションオブジェクト（Account）を実装します。要求を処理して口座残高を表示し、口座への入金または出金を行います。31 ページの「トランザクションオブジェクト（Account）の記述」を参照してください。
6. Storage オブジェクトを実装します。ビジネス（Account）オブジェクトからの要求に応じて、データにアクセスして更新します。
7. サンプルをビルドします。クライアントプログラムを作成するには、クライアントプログラムのコードをクライアントスタブと一緒にコンパイルおよびリンクします。サーバープログラムを作成するには、サーバーのコードをクライアントおよびサーバースケルトンと一緒にコンパイルおよびリンクします。34 ページの「サンプルのビルド」を参照してください。
8. サンプルを実行します。スマートエージェント、VisiTransact Transaction Service、サーバープログラム、クライアントプログラムを起動します。35 ページの「サンプルの実行」を参照してください。

クイックスタート IDL の記述

VisiTransact を使用するトランザクション型アプリケーションを作成するには、最初に、CORBA インターフェース定義言語（IDL）を使用してすべてのインターフェースを指定します。IDL は言語に依存しません。構文は C++ に似ていますが、さまざまなプログラミング言語にマッピングできます。

次のサンプルは、クイックスタートサンプルで必要な 3 つのオブジェクト（Bank、Account、Storage）を定義する **quickstart.idl** ファイルの内容です。

```
// quickstart.idl
#include "CosTransactions.idl"
#pragma prefix "visigenic.com"

module quickstart
{
  // 必須
  interface Account
  {
    float balance();
    void credit(in float amount);
    void debit(in float amount);
  };

  exception NoSuchAccount
  {
    string account_name;
  };

  interface Bank
  {
    Account get_account(in string account_name)
      raises(NoSuchAccount);
  };

  typedef sequence<string> AccountNames;

  // アダプタ
  interface Storage
  {
    float balance(in string account)
      raises(NoSuchAccount);
    void credit(in string account, in float amount)
      raises(NoSuchAccount);
    void debit(in string account, in float amount)
      raises(NoSuchAccount);
    AccountNames account_names();
  };

};
```

IDL で作成したインターフェース仕様は、クライアントアプリケーションの C++ スタブルーチンとオブジェクトのスケルトンコードを生成するために、VisiBroker ORB の `idl2cpp` コンパイラによって使用されます。スタブルーチンは、すべてのメソッド呼び出しでクライアントプログラムによって使用されます。スケルトンコードは、オブジェクトを実装するサーバープログラムを作成するために、ユーザーが記述したコードとともに使用されます。

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

トランザクションオリジネータの記述（transfer クライアントプログラム）

transfer.C という名前のファイルには、トランザクションオリジネータとなるクライアントプログラムのインプリメンテーションが含まれています。ただし、13 ページの「トランザクション処理の概要」で説明したように、トランザクションオリジネータはクライアントプログラムでなくてもかまいません。**transfer** クライアントプログラムは、1 つの VisiTransact 管理のトランザクションを実行します。詳細については、57 ページの「VisiTransact 管理のトランザクションの作成および伝達」を参照してください。別の方法でトランザクションを管理する方法については、67 ページの「トランザクションを作成および伝達するほかの方法」を参照してください。

クライアントプログラムは、次の手順を実行します。

1. ORB を初期化します。
2. コマンドラインで指定された Bank オブジェクトにバインドします。
3. トランザクションを開始します。
4. コマンドラインで指定されたトランザクションオブジェクト（送金元と送金先の Account オブジェクト）へのリファレンスを取得します。
5. **transfer** クライアントプログラムに入力された各組の送金元／送金先／金額ごとに、Account オブジェクトの `debit()` メソッドと `credit()` メソッドを呼び出します。各口座の送金前と送金後の残高を出力します。
6. トランザクションをコミットまたはロールバックします。
7. 例外を処理します。

ORB の初期化

次のサンプルで示すように、トランザクションオリジネータは、最初のタスクとして ORB を初期化する必要があります。VisiBroker のコンポーネントとして、VisiTransact のコマンドライン引数は、VisiBroker ORB の初期化呼び出し `ORB_init()` を介して VisiTransact に提供されます。したがって、コマンドラインで指定された引数を特定のアプリケーションプロセスの VisiTransact オペレーションで利用するには、アプリケーションのメインプログラムから元の `argc` 引数と `argv` 引数を `ORB_init()` に渡す必要があります。

```
...
int main(int argc, char* const* argv)
{
  try
  {
   // ORB を初期化します。
  CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
...
```

`ORB_init()` 関数は、ORB の引数と VisiTransact の引数の両方を解析し、それらを `argv` ベクタから削除してから戻ります。

Bank オブジェクトへのバインド

transfer クライアントプログラムは、トランザクション（Account）オブジェクトのメソッドを呼び出す前に、最初に `_bind()` メソッドを使用して、Bank オブジェクト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、`idl2cpp` コンパイラが自動的に生成します。`_bind()` メソッドは、Bank オブジェクトを検索して接続を確立するように ORB に要求します。

次のサンプルは、**transfer** クライアントプログラムの起動時にコマンドラインで渡された bank_name パラメータで指定されたとおりに、Bank オブジェクトにバインドする方法を示します。メモリ管理のために _var を使用する方法に注目してください。

```
const char *bank_name = argv[1];
 //Bank オブジェクトを検索します。
 Quickstart::Bank_var bank;
 //Bank ID を取得します。
 PortableServer::ObjectId_var bankId =
   PortableServer::string_to_ObjectId(bank_name);
 try
 {
   bank = quickstart::Bank::_bind("/bank_agent_poa", bankId);
   //bank = quickstart::Bank::_bind(bank_name);
 }
 catch (CORBA::Exception &ex)
 {
   const char *name;
   (bank_name == 0) ? name="NULL" : name=bank_name;
   cerr << "Unable to bind to Bank ¥"" << name << "¥": " << ex << endl;
   return 1;
 }
```

トランザクションの開始

トランザクションを開始する前に、トランザクションコンテキストを取得する必要があります。VisiTransact 管理のトランザクションは、アクティブなスレッドごとに一意のトランザクションを管理するための Current オブジェクトを使用して、アプリケーションには透過的に処理されます。VisiTransact 管理のトランザクションを使用するには、この Current オブジェクトへのリファレンスを取得する必要があります。Current オブジェクトは、それを作成したプロセス全体で有効です。つまり、任意のスレッドで使用できます。

次のサンプルは、VisiTransact 管理のトランザクションを取得します。最初に、CORBA::ORB::resolve_initial_references() メソッドを使用して、TransactionCurrent オブジェクトのオブジェクトリファレンスを取得します。このメソッドから戻された Current オブジェクトは、narrow() メソッドを使用して、特定の CosTransactions::Current オブジェクトにナローイングされます。resolve_initial_references() メソッドと narrow() メソッドの完全な説明については、VisiBroker のマニュアルを参照してください。

```
// トランザクションを開始します。
CosTransactions::Current_var current;
{
 CORBA::Object_var initRef =
   orb->resolve_initial_references("TransactionCurrent");
 current = CosTransactions::Current::_narrow(initRef);
}
...
```

VisiTransact によって管理される作業を実行するには、最初に Current インターフェースの begin() メソッドを使用して、トランザクションを開始する必要があります。スレッド内では、一度に 1 つのトランザクションだけをアクティブ化できます。次のサンプルは、VisiTransact 管理のトランザクションを開始します。

```
...
CosTransactions::Current_var current;
...
current->begin();
...
```

トランザクションオブジェクト（送金元と送金先のAccount）へのリファレンスの取得

Bank オブジェクトにバインドしたら、**transfer** プログラムの起動時に指定されたトランザクション（Account）オブジェクトへのリファレンスを取得できます。**transfer** プログラム内では、これらのリファレンスは、Bank インターフェースの `get_account()` メソッドを使って取得されます。次のサンプルは、**transfer** プログラムから抜粋した関連のコードです。

```
...
try
{
 for(CORBA::ULong i = 2; i < (CORBA::ULong)argc; i += 3)
 {
  const char* srcName = argv[i];
  const char* dstName = argv[i + 1];
  float amount = (float)atof(argv[i + 2]);

  quickstart::Account_var src = bank->get_account(srcName);
  quickstart::Account_var dst = bank->get_account(dstName);
  ...
 }
}
catch(const quickstart::NoSuchAccount& e)
{
 cout << "Exception: " << e << endl;
 commit = 0;
}
catch(const CORBA::SystemException& e)
{
 cout << "Exception: " << e << endl;
 commit = 0;
}
...
```

上のサンプルでは、**transfer** クライアントプログラムは、入力引数（プログラムの起動時にコマンドラインで入力）全体をループし、入力された送金元および送金先の口座名ごとに `get_account()` を呼び出します。入力された口座名が有効な場合、Bank オブジェクトは、対応する Account オブジェクトを戻します。Bank オブジェクトの `get_account()` メソッドの詳細については、29 ページの「Bank オブジェクトと get_account() メソッドの実装」を参照してください。

無効な口座名が入力された場合は、エラーメッセージが出力され、`commit` 変数の値が false に設定されます。同様に、`get_account()` 呼び出しの実行でシステム例外が生成された場合は、エラーメッセージが出力され、`commit` 変数の値が false に設定されます。トランザクションの完了で `commit` 変数を使用する方法については、26 ページの「トランザクションのコミットまたはロールバック」を参照してください。

トランザクション（Account）オブジェクトのメソッドの呼び出し（debit() および credit()）

transfer クライアントプログラムが送金元および送金先の Account オブジェクトとの接続を確立したら、**transfer** プログラムの起動時に入力された送金元／送金先／金額の組ごとに `Account` インターフェースの `debit()` メソッドと `credit()` メソッドを呼び出すことができます。

`debit()` メソッドと `credit()` メソッドは、前のサンプルで示した `get_account()` 呼び出しで `src` 変数と `dst` 変数に戻された情報を使用して、**transfer** プログラムのメイン `try()` 節から呼び出されます。次のサンプルは、`credit()` と `debit()` を呼び出す `try()` 節の一部です。

```
try
{
 for(CORBA::ULong i = 2; i < (CORBA::ULong)argc; i += 3)
 {
  ...
  src->debit(amount);
  dst->credit(amount);
  ...
 }
}
...
```

トランザクションのコミットまたはロールバック

トランザクションを開始したら、コミットまたはロールバックによってトランザクションを完了する必要があります。VisiTransact 管理のトランザクションのオリジネータがトランザクションを完了しない場合、VisiTransact Transaction Service は、タイムアウトになった後でトランザクションをロールバックします。ただし、ハングしたトランザクションがシステムリソースを消費しないように、トランザクションをコミットまたはロールバックすることが重要です。

次のサンプルは、**transfer** プログラムが `commit` 変数を使用して、トランザクションをコミットするかロールバックするかを決定する方法を示します。`commit` 変数が 1（true）の場合、トランザクションはコミットされます。`commit` 変数が 0（false）の場合、トランザクションはロールバックされます。次のサンプルの commit() に送られる 0 は、ヒューリスティックが報告されないという意味です。ヒューリスティックの詳細については、75 ページの「トランザクションの完了」を参照してください。

```
...
CORBA::Boolean commit = 1;
...
if(commit)
{
 cout << "*** Committing transaction ***" << endl;
 current->commit(0);
}
else
{
 cout << "*** Rolling back transaction ***" << endl;
 current->rollback();
}
...
```

例外処理

次のサンプルは、**transfer** クライアントプログラムの外側の try 文と catch 文です。これらの文を使用して、エラー（CORBA またはアプリケーションの例外）を検出したり、メッセージを出力したり、制御を戻す方法を確認してください。

```
try
{
  ...
}
catch(const CORBA::Exception& e)
{
  cerr << "Exception: " << e << endl;
  return 1;
}
catch(...)
{
  cerr << "Unknown Exception caught" << endl;
  return 1;
}
return 0;
...
```

bank_server プログラムの記述

bank_server プログラムは、main ルーチンで次の手順を実行します。

1. ORB を初期化します。
2. Storage オブジェクトを取得し、そのオブジェクトを使って Bank オブジェクトをインスタンス化します。
3. Bank オブジェクトを ORB と POA に登録します。
4. クライアント要求を待機するループに入ります。

ORB_init() メソッドに渡される argc パラメータと argv パラメータは、main ルーチンに渡されるパラメータと同じです。これらのパラメータを使用して、ORB のオプションを指定できます。

```
int main(int argc, char* const* argv)
{
  try
  {
  // ORB を初期化します。
  CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
...
```

次に、Storage オブジェクトのアクティブ化に使用される **myPOA** が作成されます。**bank_server** プログラムは、Storage オブジェクトを取得し、そのオブジェクトから口座情報を取得します。**bank_server** プログラムは、その口座情報を使って Bank オブジェクトをインスタンス化します。最後に、**bank_server** プログラムは、orb->run() メソッドを呼び出して、receiveclient が要求するイベントループを開始します。

```
const char* bank_name = argv[1];
  // ルート POA へのリファレンスを取得します。
  CORBA::Object_var obj = orb->resolve_initial_references("RootPOA");
  PortableServer::POA_var rootPOA = PortableServer::POA::_narrow(obj);

  CORBA::PolicyList policies;
  policies.length(1);
  policies[(CORBA::ULong)0] = rootPOA->create_lifespan_policy(
                                PortableServer::PERSISTENT);
  // POA マネージャを取得します。
  PortableServer::POAManager_var poa_manager = rootPOA->the_POAManager();

  // 適切なポリシーで myPOA を作成します。
  PortableServer::POA_var myPOA = rootPOA->create_POA("bank_agent_poa",
                                     poa_manager,
                                     policies);
  // Bank ID を取得します。
```

```
PortableServer::ObjectId_var bankId =
          PortableServer::string_to_ObjectId(bank_name);

// この Bank オブジェクトのための Storage オブジェクトを取得します。
quickstart::Storage_var storage = quickstart::Storage::_bind("/
bank_storage_poa", bankId);

// Bank サーバントを作成します。
PortableServer::ServantBase_var bankServant = new BankImpl(bank_name,
storage, orb);

// サーバントの ID を決定します。
PortableServer::ObjectId_var managerId =
           PortableServer::string_to_ObjectId(bank_name);

// 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, bankServant);

// POA マネージャをアクティブ化します。
poa_manager->activate();

CORBA::Object_var reference = myPOA->servant_to_reference(bankServant);
cout << reference << " is ready" << endl;

// 着信要求を待機します。
orb->run();
```

Bank オブジェクトの記述

Bank オブジェクトを実装するには、いくつかのタスクを実行する必要があります。

- POA_quickstart::Bank スケルトンクラスから BankImpl クラスを派生します。
- トランザクション（Account）オブジェクトを生成する Bank オブジェクトを実装します。

BankImpl クラスの階層

実装する BankImpl クラスは、idl2cpp コンパイラによって生成された POA_quickstart::Bank クラスから派生されます。次のサンプルは、BankImpl クラスを示します。

```
class BankImpl : public POA_quickstart::Bank
{
private:
  quickstart::AccountNames_var _account_names;
  quickstart::Storage_var _storage;
  AccountRegistry _accounts;
  PortableServer::POA_var _account_poa;
public:
  BankImpl(const char* bank_name,
    quickstart::Storage* storage, CORBA::ORB* orb);
  virtual ~BankImpl();
  virtual quickstart::Account* get_account(const char* account_name);
};
```

Bank オブジェクトと get_account() メソッドの実装

BankImpl インターフェースは、コンストラクタとデストラクタを定義します。コンストラクタは、**bank_server** プログラムの起動時に指定された名前（bank_name）で Bank オブジェクトを作成します。また、インスタンス化されたすべての Account オブジェクトを追跡するために使用される AccountRegistry のインスタンスも作成します。口座名は、Storage オブジェクトから取得されます。

```
BankImpl::BankImpl(const char* bank_name,
     quickstart::Storage* storage, CORBA::ORB* orb)
{
  _account_names = storage->account_names();
  _storage = quickstart::Storage::_duplicate(storage);

  PortableServer::POA_var root_poa =
    PortableServer::POA::_narrow(orb->resolve_initial_references("RootPOA"));
  CORBA::PolicyList policies;
  policies.length(2);
  CORBA::Any policy_value;
  policy_value <<= CosTransactions::REQUIRES;
  policies[0] = orb->create_policy(CosTransactions::OTS_POLICY_TYPE,
    policy_value);
  policies[1] =
    root_poa-
>create_implicit_activation_policy(PortableServer::IMPLICIT_ACTIVATION);
  _account_poa = root_poa->create_POA("account_poa",
                                PortableServer::POAManager::_nil(),
                                policies);
  _account_poa->the_POAManager()->activate();
  return;
}

BankImpl::~BankImpl()
{
}
```

次のサンプルは、Bank オブジェクトの get_account() メソッドを示します。get_account() メソッドは、口座が存在するかどうかをチェックし、新しい口座オブジェクトを作成します。存在しない場合は、NoSuchAccount 例外が生成されます。

```
quickstart::Account_ptr
BankImpl::get_account(const char* account_name)
{
  // account ディレクトリ内で口座を検索します。
  PortableServer::ServantBase_var servant = _accounts.get(account_name);
  CORBA::Boolean foundAccount = 0;

  if (servant == PortableServer::ServantBase::_nil()) {
    for(CORBA::ULong i = 0; !foundAccount && i < _account_names->length(); i++)
{
      if (!strcmp(_account_names[i], account_name)) {
        servant = new AccountImpl(account_name, _storage);

     // 新しい口座を出力します。
     cout << "Created " << account_name << "'s account." << endl;

        // 口座を account ディレクトリに保存します。
        _accounts.put(account_name, servant);
 foundAccount = 1;
      }
    }
    if (!foundAccount) {
      throw quickstart::NoSuchAccount(account_name);
      return 0;
    }
  }

  try {
    CORBA::Object_var ref = _account_poa->servant_to_reference(servant);
    quickstart::Account_var account = quickstart::Account::_narrow(ref);
    cout << "account generated." << endl;
    return quickstart::Account::_duplicate(account);
  }
  catch(const CORBA::Exception& e) {
    cerr << "_narrow caught exception: " << e << endl;
    return quickstart::Account::_nil();
  }
  throw quickstart::NoSuchAccount(account_name);
  return 0;
}
```

トランザクションオブジェクト（Account）の記述

トランザクション（Account）オブジェクトを実装するには、いくつかのタスクを実行する必要があります。

- POA_quickstart::Account クラスから AccountImpl クラスを派生します。
- Storage オブジェクトを呼び出す balance()、credit()、および debit() メソッドのインプリメンテーションを使用して、Account オブジェクトを実装します。

AccountImpl クラスの階層

実装する AccountImpl クラスは、idl2cpp コンパイラによって生成された POA_quickstart::Account クラスから派生されます。前の節の最初のサンプルコードを参照してください。*account*_poa には、REQUIRE という OTS_POLICY_TYPE ポリシーが定義されています。したがって、この poa でアクティブ化されるすべてのオブジェクトは、トランザクションオブジェクトになり、account も同様です。

```
class AccountImpl : public POA_quickstart::Account
{
private:
  CORBA::String_var _account_name;
  quickstart::Storage_var _storage;
public:
  AccountImpl(const char* account_name,
      quickstart::Storage* storage);
  virtual CORBA::Float balance();
  virtual void credit(CORBA::Float amount);
  virtual void debit(CORBA::Float amount);
private:
  virtual void markForRollback();
};
```

Account オブジェクトをトランザクションオブジェクトにする

オブジェクトをトランザクション対応にするには、次の 2 つの手順を実行する必要があります。

- REQUIRE または ADAPT 値の OTS_POLICY_TYPE を使って poa を作成します。
- この poa を使ってオブジェクトをアクティブ化します。

_*account*_poa は、BankImpl オブジェクトの構築時に作成されます。29 ページの「Bank オブジェクトと get_account() メソッドの実装」で示した最初のサンプルコードを参照してください。get_account() 関数では、新しい口座が _*account*_poa を使って必要に応じてアクティブ化されます。これで、Account オブジェクトがトランザクションオブジェクトになります。

Account オブジェクトとそのメソッドの実装

次のサンプルに示すように、AccountImpl クラスは、Bank オブジェクトから提供される account_name パラメータと storage パラメータを使って Account オブジェクトを作成するコンストラクタを定義します。

```
AccountImpl::AccountImpl(const char* account_name,
   quickstart::Storage* storage)
{
  _account_name = CORBA::strdup(account_name);
  _storage = quickstart::Storage::_duplicate(storage);
}
```

次のサンプルに示すように、Account クラスは、markForRollback() メソッドも実装します。このメソッドは、呼び出されると、rollback_only() を呼び出して、トランザクションオリジネータがトランザクションをロールバックするように強制します。

```
void AccountImpl::markForRollback()
{
  try
  {
   CORBA::ORB_var orb = CORBA::ORB_init();
   CORBA::Object_var initRef =
     orb->resolve_initial_references("TransactionCurrent");
   CosTransactions::Current_var current =
     CosTransactions::Current::_narrow(initRef);
   current->rollback_only();
  }
  catch(const CosTransactions::NoTransaction&)
  {
   throw CORBA::TRANSACTION_REQUIRED();
  }
}
```

markForRollback() メソッドが TransactionCurrent オブジェクトのハンドルを取得した後で、current->rollback_only() を呼び出すことができるように、取得したハンドルを Current オブジェクトにナローイングする方法に注目してください。Account オブジェクトはトランザクションオリジネータではないため、rollback() を呼び出すことができません。VisiTransact 管理のトランザクションでは、トランザクションオリジネータだけがトランザクションを完了できます。

次のサンプルに示すように、Account オブジェクトは、balance()、credit()、および debit() メソッドも実装します。

- balance() メソッドは、Storage オブジェクトに Account オブジェクトの現在の残高を要求します。
- credit() メソッドは、amount パラメータの金額だけ残高を増やすように Storage オブジェクトに要求します。
- debit() メソッドは、amount パラメータの金額だけ残高を減らすように Storage オブジェクトに要求します。

メモ　クイックスタートサンプルの Account オブジェクトはデータベース自体とも簡単に対話できますが、このサンプルは、バックエンドのデータアクセスオブジェクトが複数のビジネスロジックオブジェクトによって使用される現実的なシナリオを反映するように設計されています。これにより、後で必要に応じてデータベースを変更することが容易になります。

```
CORBA::Float AccountImpl::balance()
{
  try
  {
    return _storage->balance(_account_name);
  }
  catch(const quickstart::NoSuchAccount& e)
  {
    cerr << "Account::balance: " << e << endl;
    markForRollback();
    return 0;
  }
}

void
AccountImpl::credit(CORBA::Float amount)
{
  if(amount < 0)
  {
    cerr << "Account::credit: Invalid amount: " << amount << endl;
    markForRollback();
  }
  try
  {
    _storage->credit(_account_name, amount);
  }
  catch(const quickstart::NoSuchAccount& e)
  {
    cerr << "Account::credit: " << e << endl;
    markForRollback();
  }
}

void
AccountImpl::debit(CORBA::Float amount)
{
  if(amount < 0 || balance() - amount < 0)
  {
    cerr << "Account::debit: Invalid amount: " << amount << endl;
    markForRollback();
  }
  try
  {
    _storage->debit(_account_name, amount);
  }
  catch(const quickstart::NoSuchAccount& e)
  {
    cerr << "Account::debit: " << e << endl;
    markForRollback();
  }
}
```

サンプルのビルド

作成した **transfer.C** ファイルは、生成された **quickstart_c.C** ファイルとともにコンパイルしてリンクし、クライアントプログラムを作成します。作成した **bank_server.c** ファイルは、生成された **quickstart_s.C**、**quickstart_c.C**、および **bank.C** ファイルとともにコンパイルしてリンクし、**bank_server** プログラムを作成します。Current は擬似オブジェクトであり、VisiTransact 管理のトランザクションは Current オブジェクトを使用するため、クライアントプログラムとサーバープログラムは、VisiTransact の **its_support** ライブラリにリンクする必要があります。

Makefile の選択

VisiTransact リリースの **<install_dir>/examples/vbe/its/** ディレクトリには、このサンプル用の **Makefile** が含まれています。また、このディレクトリには **itsmk** ファイルもあります。このファイルは **Makefile** にインクルードされ、サイト固有のすべての設定を定義します。**itsmk** ファイルは、必要に応じてカスタマイズできます。**itsmk** ファイルは、VisiTransact が VisiBroker のデフォルトのインストールディレクトリにインストールされているとしています。

make によるサンプルのコンパイル

Windows: VisiBroker ORB と VisiTransact が C:¥vbroker にインストールされている場合は、次のコマンドを使用します。

```
prompt> C:
prompt> cd c:¥vbroker¥examples¥vbe¥its
prompt> nmake cpp
```

UNIX Visual C++ 標準の `nmake` コマンドは、`idl2cpp` コンパイラを実行し、各ファイルをコンパイルします。

VisiBroker ORB と VisiTransact が /usr/local/vbroker にインストールされている場合は、次のコマンドを使用します。

```
prompt> cd /usr/local/vbroker/examples/vbe/its
prompt> make cpp
```

このサンプルの `make` は、標準の UNIX 機能です。

サンプルの実行

必要なコンポーネントのコンパイルが完了し、最初の VisiTransact アプリケーションを実行する準備ができました。

スマートエージェント（osagent）の起動

VisiTransact トランザクション型アプリケーションを実行する前に、ローカルネットワークの少なくとも 1 つのホストで、VisiBroker スマートエージェントを起動する必要があります。

Windows スマートエージェントが Windows NT サービスとして設定されていない場合は、次のコマンドを使用して、スマートエージェントを起動してください。

```
prompt> osagent
```

UNIX UNIX の場合は、次のコマンドを使用します。

```
prompt> osagent
```

サンプルの実行中に、スマートエージェントを起動するのは一度だけです。

VisiTransact Transaction Service の起動

ネットワーク全体でトランザクションを有効にするには、VisiTransact Transaction Service のインスタンスを起動する必要があります。それには、次のコマンドを使用します。

prompt> ots

サンプルの実行中に、VisiTransact Transaction Service を起動するのは一度だけです。

storage_server プログラムの起動

コマンドラインで次のように入力して、**storage_server** プログラムを起動します。

prompt> storage_server MyBank

引数 `MyBank` には、銀行名を指定します。

bank_server プログラムの起動

コマンドラインで次のように入力して、**bank_server** プログラムを起動します。

prompt> bank_server MyBank

上のサンプルの引数は銀行名です。

メモ `PATH` 環境変数に VisiTransact ディレクトリ（バイナリが存在するディレクトリ）のパスが含まれていることを確認してください。Solaris の場合は、`LD_LIBRARY_PATH` 環境変数に VisiTransact の共有ライブラリのパスが含まれていることを確認してください。

トランザクションオリジネータの実行（transfer クライアントプログラム）

コマンドラインで、銀行名の後に送金元口座、送金先口座、および送金する金額を入力して、**transfer** プログラムを起動します。

prompt> transfer MyBank Paul John 20

1 回の transfer プログラムで複数の送金を実行できます。それには、送金ごとに送金元口座、送金先口座、および金額を順番に指定します。

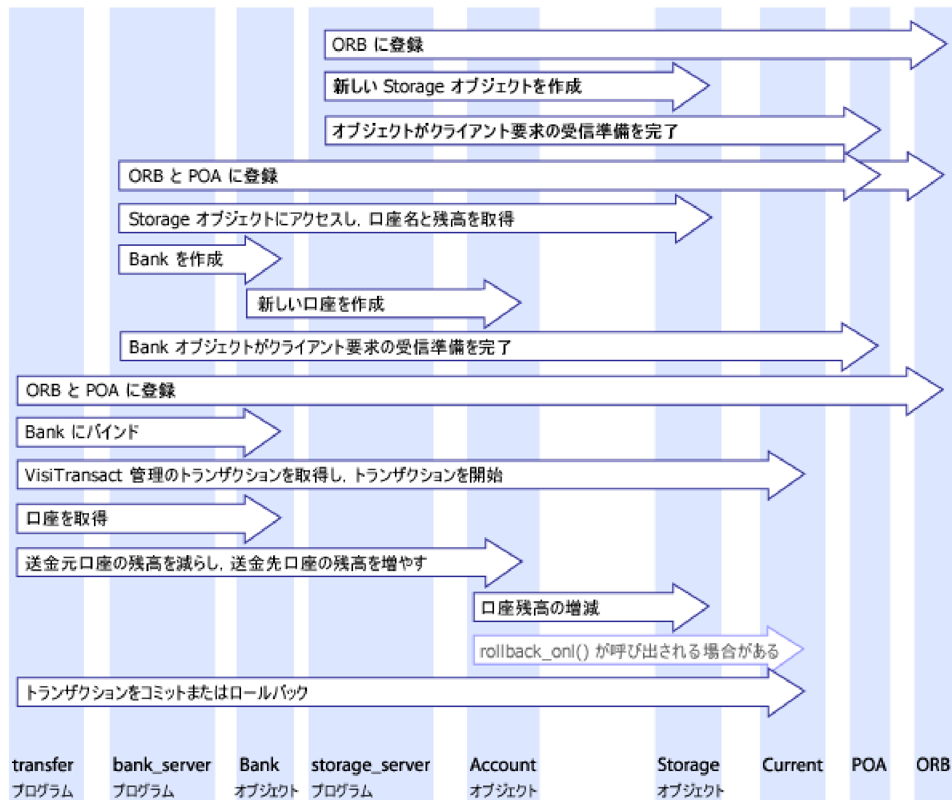
prompt> transfer MyBank Paul John 20 Ringo George 40

結果

「**MyBank Paul John 20**」で **transfer** クライアントプログラムを実行すると、**transfer** クライアントプログラムから次のように出力されます。

```
Account Balance
======= =======
Paul    100.0
John    100.0
*** Transfer $20.0 from Paul's account to John's account ***
Account Balance
======= =======
Paul    80.0
John    120.0
*** Committing transaction ***
```

図 4.2　次は、クイックスタートサンプルでアプリケーションが行う呼び出しを図で表したものです。



完全なサンプルコード

次に、クイックスタートアプリケーションのすべてのコードを示します。

クイックスタートサンプルの IDL

```
// quickstart.idl
#include "CosTransactions.idl"
#pragma prefix "visigenic.com"

module quickstart
{
  // 必須
  interface Account
  {
```

```
    float balance();
    void credit(in float amount);
    void debit(in float amount);
  };

  exception NoSuchAccount
  {
    string account_name;
  };

  interface Bank
  {
    Account get_account(in string account_name)
      raises(NoSuchAccount);
  };

  typedef sequence<string> AccountNames;

  // アダプタ
  interface Storage
  {
    float balance(in string account)
      raises(NoSuchAccount);
    void credit(in string account, in float amount)
      raises(NoSuchAccount);
    void debit(in string account, in float amount)
      raises(NoSuchAccount);
    AccountNames account_names();
  };
};
```

transfer クライアントプログラム

次のサンプルは、**transfer.C** ファイルに含まれる完全な **transfer** クライアントプログラムのコードです。

```
// transfer.C

#include "quickstart_c.hh"

USE_STD_NS

int
main(int argc, char* const* argv)
{
  try
  {
    // ORB を初期化します。
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

    // コマンドライン引数をチェックします。
    if (argc % 3 != 2)
    {
      cerr << "Usage: " << argv[0] <<
 " <bank-name> [<src> <dst> <amount>] ..." << endl;
      return 1;
    }

    // 最初の引数を解析します。
    const char *bank_name = argv[1];

    // Bank オブジェクトを検索します。
    quickstart::Bank_var bank;

    // Bank ID を取得します。
    PortableServer::ObjectId_var bankId =
                PortableServer::string_to_ObjectId(bank_name);

    try
    {
      bank = quickstart::Bank::_bind("/bank_agent_poa", bankId);
```

```
    // bank = quickstart::Bank::_bind(bank_name);
  }
  catch (CORBA::Exception &ex)
  {
    const char *name;
    (bank_name == 0) ? name="NULL" : name=bank_name;
    cerr << "Unable to bind to Bank ¥"" << name << "¥": " << ex << endl;
    return 1;
  }

  // トランザクションを開始します。
  CosTransactions::Current_var current;
  {
    CORBA::Object_var initRef =
      orb->resolve_initial_references("TransactionCurrent");
    current = CosTransactions::Current::_narrow(initRef);
  }

  current->begin();

  CORBA::Boolean commit = 1;
  try
  {
    for(CORBA::ULong i = 2; i < (CORBA::ULong)argc; i += 3)
    {
const char* srcName = argv[i];
const char* dstName = argv[i + 1];
float amount = (float)atof(argv[i + 2]);

quickstart::Account_var src = bank->get_account(srcName);
quickstart::Account_var dst = bank->get_account(dstName);

cout << "Account¥tBalance" << endl;
cout << "=======¥t=======" << endl;
cout << srcName << "¥t" << src->balance() << endl;
cout << dstName << "¥t" << dst->balance() << endl;
cout << "¥n*** Transfer $" << amount << " from " <<
  srcName << "'s account to " <<
  dstName << "'s account ***¥n" << endl;

src->debit(amount);
dst->credit(amount);

cout << "Account¥tBalance" << endl;
cout << "=======¥t=======" << endl;
cout << srcName << "¥t" << src->balance() << endl;
cout << dstName << "¥t" << dst->balance() << endl;
    }
  }
  catch(const quickstart::NoSuchAccount& e)
  {
    cout << e << endl;
    commit = 0;
  }
  catch(const CORBA::SystemException& e)
  {
    cout << "Exception: " << e << endl;
    commit = 0;
  }

  // トランザクションをコミットまたはロールバックします。
  if(commit)
  {
    cout << "*** Committing transaction ***" << endl;
    current->commit(0);
  }
  else
  {
    cout << "*** Rolling back transaction ***" << endl;
    current->rollback();
  }
 }
 catch(const CORBA::Exception& e)
```

```
  {
    cerr << "Exception: " << e << endl;
    return 1;
  }
  catch(...)
  {
    cerr << "Unknown Exception caught" << endl;
    return 1;
  }
  return 0;
}
```

bank_server プログラム

次のサンプルは、**bank_server.C** ファイルに含まれる **bank_server** プログラムのコードです。

```
// bank_server.C

#include "bank.h"

USE_STD_NS

int
main(int argc, char* const* argv)
{
  try
  {
    // ORB を初期化します。
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

    // コマンドライン引数をチェックします。
    if(argc != 2)
    {
      cerr << "Usage: " << argv[0] << " <bank-name>" << endl;
      return 1;
    }
    const char* bank_name = argv[1];

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

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

    // POA マネージャを取得します。
    PortableServer::POAManager_var poa_manager = rootPOA-
>the_POAManager();

    // 適切なポリシーで myPOA を作成します。
    PortableServer::POA_var myPOA = rootPOA->create_POA("bank_agent_poa",
                                      poa_manager,
                                      policies);
    // Bank ID を取得します。
    PortableServer::ObjectId_var bankId =
                PortableServer::string_to_ObjectId(bank_name);

    // この Bank オブジェクトのための Storage オブジェクトを取得します。
    quickstart::Storage_var storage = quickstart::Storage::_bind("/
bank_storage_poa", bankId);

    // Bank サーバントを作成します。
    PortableServer::ServantBase_var bankServant = new BankImpl(bank_name,
storage, orb);

    // サーバントの ID を決定します。
    PortableServer::ObjectId_var managerId =
                PortableServer::string_to_ObjectId(bank_name);
```

```
    // 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, bankServant);

    // POA マネージャをアクティブ化します。
    poa_manager->activate();

    CORBA::Object_var reference = myPOA->servant_to_reference(bankServant);
    cout << reference << " is ready" << endl;

    // 着信要求を待機します。
    orb->run();
  }
  catch(const CORBA::Exception& e)
  {
    cerr << "Exception: " << e << endl;
    return 1;
  }
  catch(...)
  {
    cerr << "Unknown Exception caught" << endl;
    return 1;
  }
  return 0;
}
```

Bank オブジェクトと Account（トランザクション）オブジェクト

次のサンプルは、**bank.h** ファイルに含まれる AccountRegistry、Bank、および Account クラスのコードです。

```
// bank.h

#include "quickstart_s.hh"
#include <vport.h>

// AccountRegistry は、銀行口座インプリメンテーションのホルダーです
class AccountRegistry
{
  public:
    AccountRegistry() : _count(0), _max(16), _data((Data*)NULL)
    {
      _data = new Data[16];
    }

    ~AccountRegistry() { delete[] _data; }

    void put(const char* name, PortableServer::ServantBase_ptr servant) {

      VISMutex_var lock(_lock);

      if (_count + 1 == _max) {
        Data* oldData = _data;
        _max += 16;
        _data = new Data[_max];
        for (CORBA::ULong i = 0; i < _count; i++)
          _data[i] = oldData[i];
        delete[] oldData;
      }

      _data[_count].name = name;
      servant->_add_ref();
      _data[_count].account = servant;
      _count++;
    }

    PortableServer::ServantBase_ptr get(const char* name) {

      VISMutex_var lock(_lock);
```

```
    for (CORBA::ULong i = 0; i < _count; i++) {
      if (strcmp(name, _data[i].name) == 0) {
        _data[i].account->_add_ref();
        return _data[i].account;
      }
    }
    return PortableServer::ServantBase::_nil();
  }

 private:
  struct Data {
    CORBA::String_var name;
    PortableServer::ServantBase_var  account;
  };

  CORBA::ULong _count;
  CORBA::ULong _max;
  Data* _data;
  VISMutex _lock;   // 同期のためにロック
};

class BankImpl : public POA_quickstart::Bank
{
private:
  quickstart::AccountNames_var _account_names;
  quickstart::Storage_var _storage;
  AccountRegistry _accounts;
  PortableServer::POA_var _account_poa;
public:
  BankImpl(const char* bank_name,
    quickstart::Storage* storage, CORBA::ORB* orb);
  virtual ~BankImpl();
  virtual quickstart::Account* get_account(const char* account_name);
};

class AccountImpl : public POA_quickstart::Account
{
private:
  CORBA::String_var _account_name;
  quickstart::Storage_var _storage;
public:
  AccountImpl(const char* account_name,
       quickstart::Storage* storage);
  virtual CORBA::Float balance();
  virtual void credit(CORBA::Float amount);
  virtual void debit(CORBA::Float amount);
private:
  virtual void markForRollback();
};
```

次のサンプルは、**bank.C** ファイルに含まれる `BankImpl` および `AccountImpl` クラスのコードです。

```
// bank.C

#include "bank.h"

USE_STD_NS

BankImpl::BankImpl(const char* bank_name,
     quickstart::Storage* storage, CORBA::ORB* orb)
{
  _account_names = storage->account_names();
  _storage = quickstart::Storage::_duplicate(storage);

  PortableServer::POA_var root_poa =
    PortableServer::POA::_narrow(orb->resolve_initial_references("RootPOA"));
  CORBA::PolicyList policies;
  policies.length(2);
  CORBA::Any policy_value;
  policy_value <<= CosTransactions::REQUIRES;
  policies[0] = orb->create_policy(CosTransactions::OTS_POLICY_TYPE,
    policy_value);
  policies[1] = root_poa-
>create_implicit_activation_policy(PortableServer::IMPLICIT_ACTIVATION);
  _account_poa = root_poa->create_POA("account_poa",
                                   PortableServer::POAManager::_nil(),
                                   policies);
  _account_poa->the_POAManager()->activate();
  return;
}

BankImpl::~BankImpl()
{
}

quickstart::Account_ptr
BankImpl::get_account(const char* account_name)
{
  // account ディレクトリ内で口座を検索します。
  PortableServer::ServantBase_var servant = _accounts.get(account_name);
  CORBA::Boolean foundAccount = 0;

  if (servant == PortableServer::ServantBase::_nil()) {
    for(CORBA::ULong i = 0; !foundAccount && i < _account_names->length(); i++)
{
      if (!strcmp(_account_names[i], account_name)) {
        servant = new AccountImpl(account_name, _storage);

     // 新しい口座を出力します。
     cout << "Created " << account_name << "'s account." << endl;

        // 口座を account ディレクトリに保存します。
        _accounts.put(account_name, servant);
 foundAccount = 1;
      }
    }
    if (!foundAccount) {
      throw quickstart::NoSuchAccount(account_name);
      return 0;
    }
  }

  try {
    CORBA::Object_var ref = _account_poa->servant_to_reference(servant);
    quickstart::Account_var account = quickstart::Account::_narrow(ref);
    cout << "account generated." << endl;
    return quickstart::Account::_duplicate(account);
  }
  catch(const CORBA::Exception& e) {
    cerr << "_narrow caught exception: " << e << endl;
    return quickstart::Account::_nil();
  }
```

```
  throw quickstart::NoSuchAccount(account_name);
  return 0;
}

AccountImpl::AccountImpl(const char* account_name,
    quickstart::Storage* storage)
{
  _account_name = CORBA::strdup(account_name);
  _storage = quickstart::Storage::_duplicate(storage);
}

void
AccountImpl::markForRollback()
{
  try
  {
    CORBA::ORB_var orb = CORBA::ORB_init();
    CORBA::Object_var initRef =
      orb->resolve_initial_references("TransactionCurrent");
    CosTransactions::Current_var current =
      CosTransactions::Current::_narrow(initRef);
    current->rollback_only();
  }
  catch(const CosTransactions::NoTransaction&)
  {
    throw CORBA::TRANSACTION_REQUIRED();
  }
}

CORBA::Float
AccountImpl::balance()
{
  try
  {
    return _storage->balance(_account_name);
  }
  catch(const quickstart::NoSuchAccount& e)
  {
    cerr << "Account::balance: " << e << endl;
    markForRollback();
    return 0;
  }
}

void
AccountImpl::credit(CORBA::Float amount)
{
  if(amount < 0)
  {
    cerr << "Account::credit: Invalid amount: " << amount << endl;
    markForRollback();
  }
  try
  {
    _storage->credit(_account_name, amount);
  }
  catch(const quickstart::NoSuchAccount& e)
  {
    cerr << "Account::credit: " << e << endl;
    markForRollback();
  }
}

void
AccountImpl::debit(CORBA::Float amount)
{
  if(amount < 0 || balance() - amount < 0)
  {
    cerr << "Account::debit: Invalid amount: " << amount << endl;
    markForRollback();
  }
  try
  {
```

```
      _storage->debit(_account_name, amount);
    }
    catch(const quickstart::NoSuchAccount& e)
    {
      cerr << "Account::debit: " << e << endl;
      markForRollback();
    }
  }
```

第 5 章

トランザクションオブジェクトの作成

C++ のサーバーとクライアント向けのトランザクション対応オブジェクトは、2 つの方法で（トランザクションオブジェクトインターフェースを継承して、または直接インプリメンテーションによって）作成できます。

トランザクション対応オブジェクトインターフェースの継承

C++ サーバー／クライアントで、オブジェクトのインターフェースで CosTransactions::TransactionalObject インターフェースを継承することで、オブジェクトをトランザクションオブジェクトとして定義します。

トランザクション対応オブジェクトインターフェースの実装

この方法は、OMG OTS 1.2 仕様に準拠しており、C++ のサンプルで示されています。この方法では、C++ サーバー／クライアントがトランザクションオブジェクトを定義してトランザクションの制御を拡張できます。この新しい拡張機能を使用して、サーバーは、ターゲットオブジェクトに適切なポリシーを設定することでトランザクションの要件を適用できます。これに対して、クライアントは、クライアントの動作を規定するいくつかの新しいポリシーを使用し、ターゲットオブジェクトの要件にしたがって対応する呼び出しを実行する必要があります。これにより、強力なセマンティクス制御が保証されます。

また、クライアント側とサーバー側の両方で、ポリシーの作成とポリシーのチェックがサポートされます。これは、分散トランザクション環境でのトランザクションオブジェクトリファレンスの作成とトランザクション対応の呼び出しを保護します。

トランザクション対応 POA ポリシーインターフェース

OTSPolicy

このポリシーは、ターゲットオブジェクトの共有トランザクション動作を記述するために使用されます。次の 3 つの値があります。

- `REQUIRES` - ターゲットオブジェクトは受信呼び出しでトランザクションの存在を必要とします。
- `FORBIDS` - ターゲットオブジェクトの呼び出しでトランザクションが存在してはなりません。
- `ADAPTS` - ターゲットオブジェクトは現在のトランザクションが存在するかどうかに適応します。

InvocationPolicy

このポリシーは、ターゲットオブジェクトがサポートするトランザクションの種類を指定します。ターゲットオブジェクトは、呼び出しポリシーを設定することで、共有（SHARED）トランザクションモデル、非共有（UNSHARED）トランザクションモデル、またはそれらのどちらか（EITHER）をサポートするように選択できます。

ターゲットオブジェクトで OTSPolicy と InvocationPolicy の両方を定義する場合、いくつかの組み合わせは無効になります。詳細については、OMG OTS 仕様バージョン 1.2 を参照してください。無効な組み合わせでポリシーを作成すると、InvalidPolicy 例外が発生します。

NonTxTargetPolicy

このポリシーは、アクティブなトランザクションの間の非トランザクションターゲットオブジェクトに対するクライアント呼び出しを許可（PERMIT）または禁止（PREVENT）するために使用されます。このポリシーと矛盾するクライアント呼び出しを行うと、INVALID_TRANSACTION 例外が発生します。

影響を受けるサーバーの動作

新しいトランザクションサーバーは、OTSPolicy と InvocationPolicy（オプション）を使用して、作成するオブジェクトのトランザクション動作を制御する必要があります。新しいサーバーでは、前に指定された TransactionalObject を使用してはなりません。

必要なトランザクション動作を実行するオブジェクトを作成するために、サーバーは、適切なポリシーを使って POA を作成する必要があります。POA は、そのポリシー値を使用して、オブジェクトリファレンスの作成を制御します。VisiTransact Transaction Manager は、これらのポリシーの有効性を検証し、次のいずれかを実行します。

1. すべてのポリシーが有効な場合は、オブジェクトのアクティブ化とリファレンスの作成のために、指定されたポリシーで POA が作成されます。
2. ポリシーが無効な場合は、例外が生成されます。
3. POA の作成時に OTSPolicy がない場合、VisiTransact Transaction Manager は、デフォルト値（FORBIDS）を提供します。

InvocationPolicy がない場合、作成されたオブジェクトは、EITHER の InvocationPolicy をサポートするとして処理する必要があります。

影響を受けるクライアントの動作

クライアントは、ターゲットオブジェクトの要件に合った状況で呼び出しを実行します。それ以外の状況で呼び出しを実行すると、VisiTransact Transaction Manager から例外を受け取ります。

トランザクションを必要とするオブジェクト（REQUIRES）の場合は、アクティブなトランザクションのスコープ内でオブジェクトを呼び出す必要があります。たとえば、呼び出し元スレッドは、アクティブなトランザクションに関連付けられている必要があります。

トランザクションを禁止するオブジェクト（FORBIDS）の場合は、アクティブなトランザクションのスコープ外でオブジェクトを呼び出す必要があります。たとえば、呼び出し元スレッドは、どのトランザクションにも関連付けられていません。

トランザクションに適応するオブジェクト（ADAPTS）の場合は、どちらの場合でもオブジェクトを呼び出すことができます。ただし、ターゲットオブジェクトの動作は、受信呼び出しがアクティブなトランザクションに関連付けられているかどうかによって異なります。

アクティブなトランザクションの間に、クライアントは、NonTxTargetPolicy を使用して、非トランザクションオブジェクトへの呼び出しを操作できます。クライアントがこのポリシーを設定していない場合、デフォルト値は PERMIT になります。

非共有（UNSHARED）トランザクションの処理

現在の Visibroker は、OMG AMI モデルとは異なる NativeMessaging という非同期メソッド呼び出しモデルを採用しているため、このリリースでは非共有（UNSHARED）トランザクションが十分にサポートされていません。したがって、VisiTransact Transaction Manager のサーバーとクライアントは、非共有（UNSHARED）トランザクションに直接参加できません。

ただし、この制約とは無関係に、サーバー側で POA に対して任意の有効な値の InvocationPolicy を作成できます。

第 6 章

トランザクションの構築方法の決定

ここでは、VisiTransact Transaction Manager を使用するトランザクション型アプリケーションを構築する際に選択できる方法について説明します。

トランザクション管理の方法

プログラムでは、使用するコンテキスト管理のタイプとコンテキスト伝達（トランザクションコンテキストを別のオブジェクトに転送すること）の方法を選択できます。あるタイプのコンテキスト管理を使用しても、トランザクション伝達の選択は制限されません。

直接的なコンテキスト管理と間接的なコンテキスト管理

OMG による CORBA サービストランザクションサービス仕様には、次のタイプのコンテキスト管理が定義されています。

- **間接的なコンテキスト管理。** 間接的なコンテキスト管理の場合、アプリケーションは、トランザクションサービスから提供される Current オブジェクトを使用して、トランザクションコンテキストをアプリケーションの制御用スレッドに関連付けて管理します。
- **直接的なコンテキスト管理。** 直接的なコンテキスト管理の場合、アプリケーションは、トランザクションに関連付けられている Control オブジェクトなどのオブジェクトを操作します。

間接的なコンテキスト管理を使用すると、プログラミングが簡単になります。また、VisiTransact Transaction Service がトランザクションコンテキストを制御することで、アプリケーションのパフォーマンスの向上と最適化の機能を利用できます。たとえば、VisiTransact 管理のトランザクションは、基底の VisiBroker ORB を活用して、リモート呼び出しを最小限にします。さらに、VisiTransact 管理のトランザクションは、伝達コンテキストとトランザクションコンテキストをアプリケーションエンドでキャッシュしてシステムリソースを節約するため、このデータを取得するためのリモート呼び出しを削減できます。

直接的なコンテキスト管理は、明示的な伝達を使用したり、複数の VisiTransact Transaction Service のインスタンスを使ってトランザクションを生成する場合に便利なことがあります。また、VisiTransact ライブラリにリンクしない場合は、直接的なコンテキスト管理を使用する必要があります。まれに、VisiTransact ライブラリを使用しないで、独自の IDL ファイルに基づく独自のスタブを使用する場合があります。独自のスタブを使用できるのは、直接的なコンテキスト管理を使用する場合だけです。間接的なコンテキスト管理を使用する場合は、Current オブジェクトを使用します。Current オブジェクトを使用するには、VisiTransact ライブラリを使用します。

直接的なコンテキスト管理を使用する場合、または両方のコンテキスト管理モードを併用する場合は、アプリケーションでトランザクションの整合性を保証する必要があります。

いったん直接的なコンテキスト管理が使用されると、VisiTransact Transaction Service は、トランザクションの完了をチェックする機能を失います。checked behavior の詳細については、76 ページの「VisiTransact Transaction Service が checked behavior を実行するしくみ」を参照してください。

暗黙的な伝達と明示的な伝達

OMG による CORBA サービストランザクションサービス仕様には、次の伝達タイプが定義されています。

- **暗黙的な伝達** 暗黙的な伝達の場合、要求はアプリケーションのトランザクションに暗黙的に関連付けられます。つまり、それらの要求がアプリケーションのトランザクションコンテキストを共有します。トランザクションコンテキストは、VisiTransact Transaction Service によって関与するオブジェクトに暗黙的に転送されます。トランザクションオリジネータが直接介入することはありません。暗黙的な伝達をサポートするオブジェクトは、通常、トランザクションサービスオブジェクトを明示的なパラメータとしては受け取りません。
- **明示的な伝達** 明示的な伝達の場合、トランザクションオリジネータ（および関与するトランザクションオブジェクト）は、トランザクションサービスによって定義されたオブジェクトを明示的なパラメータとして渡すことで、トランザクションコンテキストを伝達します。

暗黙的な伝達の主な利点は、VisiTransact Transaction Service が自動的にトランザクションの伝達を処理することです。もう 1 つの利点は、既存のメソッドのシグニチャを変更しなくてもトランザクションをサポートできることです。オブジェクトをトランザクション対応にすることで、オブジェクトのすべてのメソッドをトランザクションの一部として実行できます。

明示的な伝達にも利点があります。1 つめの利点として、1 つのオブジェクトでトランザクションメソッドと非トランザクションメソッドを併用できます。これは、トランザクション内のあるメソッドではトランザクションのセマンティクスが必要だが、別のメソッドでは必要ない場合に便利です。

2 つめの利点として、CORBA 1.x インプリメンテーション（VisiBroker 2.0 など）と相互運用性が必要な場合に、明示的な伝達を使用できます。明示的な伝達には、ORB とトランザクションサービスの連係が不要なため、このような下位互換性のために使用できます。

3 つめの利点として、明示的な伝達を使用すると、ほかのオブジェクトがトランザクションを完了できるようになります。つまり、明示的な伝達では、Terminator をトランザクションの別の参加者に渡すことができます。これで、その参加者がトランザクションを完了できます。

コンテキストの管理と伝達

クライアントは、暗黙的または明示的な伝達とともに、直接的または間接的なコンテキスト管理を使用できます。結果として、クライアントアプリケーションは、次の方法でトランザクションオブジェクトと通信できます。

- 暗黙的な伝達による間接的なコンテキスト管理
- 明示的な伝達による間接的なコンテキスト管理
- 暗黙的な伝達による直接的なコンテキスト管理
- 明示的な伝達による直接的なコンテキスト管理

暗黙的な伝達による間接的なコンテキスト管理

クライアントアプリケーションは、Current オブジェクトのメソッドを使用して、トランザクションを作成および制御します。トランザクションオブジェクトに対して要求を発行すると、現在のスレッドに関連付けられているトランザクションコンテキストが暗黙的にオブジェクトに伝達されます。

VisiTransact 管理のトランザクションは、このカテゴリに入ります。VisiTransact 管理のトランザクションの場合は、VisiTransact が checked behavior を保証します。checked behavior の詳細については、76 ページの「VisiTransact Transaction Service が checked behavior を実行するしくみ」を参照してください。

メモ　暗黙的な伝達による間接的なコンテキスト管理は、VisiTransact 管理のトランザクションと正確に同じではありません。VisiTransact 管理のトランザクションの場合は、暗黙的な伝達の前に `Current::begin` の使用を特別に指定します。

VisiTransact 管理のトランザクションの使用方法については、57 ページの「VisiTransact 管理のトランザクションの作成および伝達」を参照してください。

明示的な伝達による間接的なコンテキスト管理

クライアントは、Current オブジェクト、Control オブジェクト、およびトランザクションの状態を記述するほかのオブジェクトを組み合わせて使用します。`Current` オブジェクトを使用するクライアントアプリケーション（自動的に暗黙的な伝達も使用する）は、`Current::getControl()` メソッドによって Control オブジェクトへのアクセスを取得することで、明示的な伝達を使用できます。VisiTransact Transaction Service オブジェクトをトランザクションオブジェクトへの明示的なパラメータとして使用できます。これは、明示的な伝達です。

暗黙的な伝達による直接的なコンテキスト管理

クライアント は、Current オブジェクト 、Control オブジェクト 、およびト ランザクショ ンの状態を記述するほかのオブジェクト を組み合わせて使用します。VisiTransact Transaction Service オブジェクト に直接アク セスするクライアント は、`Current::resume()` メソッド を使用して、そのスレッド に関連付けられている暗黙的なト ランザクショ ンコンテキスト を設定できます。これにより 、クライアント は、ト ランザクショ ンコンテキスト を暗黙的に伝達する必要があるオブジェクト のメソッド を呼び出すこと ができます。

明示的な伝達による直接的なコンテキスト管理

クライアントアプリケーションは、Control オブジェクトとトランザクションの状態を記述するほかのオブジェクトに直接アクセスします。トランザクションをオブジェクトに伝達するため、クライアントは、メソッドの明示的なパラメータとして適切な VisiTransact Transaction Service オブジェクトを含める必要があります。

アプリケーションからトランザクションを管理する方法については、67 ページの「トランザクションを作成および伝達するほかの方法」を参照してください。

インプロセスとアウトプロセスの VisiTransact Transaction Service

ほとんどのトランザクションが単一のプロセスに分離されて使用される場合は、VisiTransact Transaction Service のインプロセスのインスタンスを使用できます。ただし、トランザクションの高可用性の必要条件（通常は、VisiTransact Transaction Service のスタンドアロンインスタンスによって処理される）を満たすには、トランザクションの処理中にアプリケーションプロセスが実行されたままである必要があります。この必要条件は、アプリケーションプロセス内に組み込まれた VisiTransact Transaction Service のインスタンスをほかのアプリケーション（プロセスの外部）が使用する場合に特に重要になります。54 ページの「アプリケーションへの VisiTransact Transaction Service インスタンスの埋め込み」を参照してください。

ネットワークでは、VisiTransact Transaction Service の複数のインスタンスを使用できます。トランザクションの動作を予測しやすくするために、トランザクションオリジネータが使用する VisiTransact Transaction Service のインスタンスを指定できます。

- 使用される VisiTransact Transaction Service のインスタンスは、`ORB_init()` に渡される引数や Current インターフェースの属性を設定して制御できます。Current の属性は、`ORB_init()` に渡されるどの引数より優先します。この属性は、`Current::begin()` を使用する後続のトランザクションにだけ適用されます。
- 直接的なコンテキスト管理の場合は、TransactionFactory の適切なインスタンスに名前でバインドします。

`-Dvbroker.ots.currentName` 引数の設定方法については、62 ページの「VisiTransact Transaction Service のインスタンスの検索」を参照してください。

マルチスレッド

VisiTransact はマルチスレッド対応です。マルチスレッドアプリケーションは、スレッドプーリング機能や接続管理機能など、基底の VisiBroker ORB の機能を活用できます。

VisiBroker ORB によるスレッドと接続の管理を利用するとシステムリソースを節約できますが、特定のトランザクションにどのスレッドを割り当てるかを制御する必要がある場合は、スレッドプーリング機能が欠点になる可能性があります。スレッドプーリングモデルの場合は、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、そのスレッドは、その要求の存続期間だけ有効です。より細かな制御が必要な場合は、VisiBroker が提供するほかのスレッドモデルを検討してください。また、ほかのライブラリがスレッドセーフでない場合は、スレッドの安全性に問題が発生する場合もあります。

既存のアプリケーションとトランザクションシステムの統合

ほかの CORBA トランザクションサービスを使用する外部トランザクションシステムを統合できます。VisiTransact は、CORBA 2.6 に完全に準拠しているため、OMG CORBA トランザクションサービス仕様のほかの CORBA 2.6 準拠インプリメンテーションと相互運用できます。VisiTransact は、CORBA サービス仕様に対して、ほかのトランザクションサービスインプリメンテーションでは処理できない優れた拡張機能（`begin_with_name()` などの便利なメソッドやその他の機能）を備えています。

さらに、ユーザー自身、サードパーティ、またはデータベースベンダーから提供される CORBA サービス準拠の任意のリソースを使用できます。

`Resource` インターフェースを使用して、独自のリソースを実装することもできます。この場合は、ログ、回復、ヒューリスティックなどの必要なコードが自動的に処理されないため、複雑なプログラミングを行う必要があります。

構築方法の組み合わせ

作成する分散トランザクション型アプリケーションの目的に合わせて、この章で説明する構築方法を適切に組み合わせることができます。

- **さまざまなタイプのトランザクション構築方法を組み合わせる。**たとえば、明示的な伝達を使用するトランザクションがある場合は、暗黙的な伝達に切り替えることができます。詳細については、71 ページの「明示的な伝達から暗黙的な伝達への変更」を参照してください。
- **複数のシステムを VisiTransact アプリケーションに統合する。**たとえば、トランザクション型アプリケーションでデータベース、トランザクションプロセスモニタ、およびメッセージングソフトウェアを使用し、それらをすべて VisiTransact に統合できます。

Web 用のトランザクションの実装

Web ベースのトランザクション型アプリケーションを開発する場合は、アプリケーションのフロントエンドとして Web ブラウザを使用し、トランザクションの開始などのロジックはサーバーベースのオブジェクトに置くことができます。

VisiTransact トランザクションを Web サーバーのローカルネットワーク内に収めると、VisiTransact Transaction Service とトランザクション参加者の局所性によってパフォーマンスが向上します。また、1 つの会社が制御できる範囲でトランザクションの局所自律性を提供できます。このアプリケーションアーキテクチャでは、外部ネットワークにまたがる通信の問題がトランザクションの完了や整合性に影響を及ぼしません。

C++ VisiTransact アプリケーションの構築

VisiTransact を使用する C++ アプリケーションを設計する場合は、C++ アプリケーションコンポーネントで VisiTransact Transaction Service のスタンドアロンインスタンスまたは VisiTransact Transaction Service の埋め込みインスタンスを使用できます。

以下では、これらのインスタンスの使用方法について詳しく説明します。

VisiTransact Transaction Service スタンドアロンインスタンスの使用

ほとんどの VisiTransact アプリケーションでは、プロセス内にインスタンスを埋め込むのではなく、ネットワーク上で実行されている VisiTransact Transaction Service のインスタンスを使用します。アプリケーションは、実行されると、任意の有効な VisiTransact Transaction Service インスタンスを使用できます。つまり、使用されている VisiTransact Transaction Service のインスタンスを制御できます。

VisiTransact Transaction Service インターフェースを使用する C++ プログラムは、its_support.lib（Solaris の場合は its_support.so）にリンクされている必要があります。

メモ　プログラムが明示的な伝達による直接的なコンテキスト管理だけを使用している場合は、**CosTransactions.idl** ファイルまたは **VISTransactions.idl** ファイルから生成されたスタブとヘッダーファイルを使用できます。

アプリケーションへの VisiTransact Transaction Service インスタンスの埋め込み

C++ 実行可能プログラムに VisiTransact Transaction Service のインスタンスを埋め込むには、アプリケーションを **ots_r.lib**、**its_support.lib**（Solaris の場合は **ots_r.so**、**its_support.so**）の各ライブラリにリンクします。これらのライブラリをリンク行に追加することで、VisiTransact Transaction Service のインスタンスがアプリケーションのプロセスに埋め込まれます。

VisiTransact ライブラリにリンクする場合は、VisiTransact から提供される _c.hh ファイルと _s.hh ファイルをインクルードする必要があります。独自のスタブファイルを生成することはできません。これは、VisiTransact ライブラリに埋め込まれたオブジェクトと互換性のあるバージョンのヘッダーを正しく使用するためです。VisiTransact ライブラリにリンクする場合は、この手順を実行する必要があります。

また、次の手順にしたがって、アプリケーションから VisiTransact Transaction Service のインスタンスを明示的に初期化および終了する必要があります。

1. C++ アプリケーションで **visits.h** ファイルをインクルードします。
2. `ORB_init()` を使用して、VisiTransact サーバーコンポーネントを初期化します。`VISTransactionService::init()` を呼び出して、VisiTransact Transaction Service のインスタンスを初期化します。これは、`ORB_init()` 呼び出しの後で行う必要があります。次に例を示します。

   ```
   CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
   VISIts::init(argc, argv);
   ```

3. `VISTransactionService::terminate()` を呼び出して、VisiTransact Transaction Service のインスタンスをシャットダウンします。
4. リンク行に次のファイルが必要です。

   UNIX：`ots_r.so`

   WinNT：`ots_r.lib`

メモ　64 ビットプラットフォームでは、`otsinit64.o`、`ots64_r.so`、および `its_support64.so` が必要です。AIX では `ots_r.a`（`ots64_r.a`）と `its_support.a`（`its_support64.a`）、HP-UX では `ots_r.sl`（`ots64_r.sl`）と `its_support.sl`（`its_support64.sl`）が必要です。

5. **osfind** を使用して、VisiTransact Transaction Service が実行されていることを確認します。

次のサンプルは、VisiTransact Transaction Service を埋め込むアプリケーションです。

```
// アプリケーションメイン
#include <visits.h> // VISIts
#include <corba.h>

int main(int argc, char** argv)
{
  CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
  VISTransactionService::init(argc, argv);
  // ここでアプリケーションのメインの作業を実行します。
  ...
  VISTransactionService::terminate();
}
```

VisiTransact Transaction Service 埋め込みインスタンスへのバインド

アプリケーションサーバーに VisiTransact Transaction Service を埋め込んだ場合は、クライアントが VisiTransact Transaction Service の正しいインスタンスにバインドされるようにする必要があります。それには、クライアントアプリケーションの起動時に、特定のコマンドライン引数を使って VisiTransact Transaction Service の名前を指定する必要があります。この名前は、アプリケーションサーバーに埋め込まれた名前と一致する必要があります。

メモ　`Current` を使用するのではなく、`TransactionFactory` から直接トランザクションを作成する場合、クライアントは正しい `TransactionFactory` にバインドされる必要があります。CORBA オブジェクトにバインドするためのセマンティクスを参照して、クライアントを正しいオブジェクトにバインドしてください。

VisiTransact から提供されるヘッダーファイルの使い方

its_support.lib または **ots_r.lib** にリンクされる C++ ソースファイルをコンパイルするには、**CosTransactions.idl** または **VISTransactions.idl** から生成される IDL クライアントスタブヘッダーファイル*ではなく*、VisiTransact から提供される **CosTransactions_c.hh** または **VISTransactions_c.hh** のバージョンをインクルードする必要があります。VisiTransact 提供のライブラリにリンクされるオブジェクトは、それらの構築に使用されたヘッダーファイルに対してのみ有効になります。`Current` インターフェースを使用するすべてのアプリケーションは、これらのライブラリにリンクされます。

第 7 章

VisiTransact 管理のトランザクションの作成および伝達

ここでは、VisiTransact 管理のトランザクションにおける Current インターフェースの使用方法を取り上げます。`Current` を使って VisiTransact 管理のトランザクションにアクセスする方法と、`Current` インターフェースのメソッドを使ってトランザクションを開始、ロールバック、およびコミットする方法について説明します。VisiTransact 管理のトランザクションでトランザクションオブジェクトを共有する方法についても説明します。

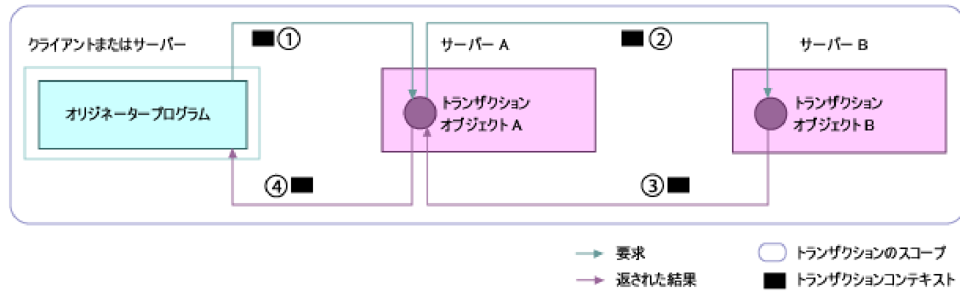
VisiTransact 管理のトランザクションで使用される Current の概要

VisiTransact 管理のトランザクションでは、Current インターフェースを使用してすべてのトランザクションを管理します。Current を使ってトランザクションを開始したり、Current を使ってトランザクションを暗黙的に伝達します。つまり、トランザクションを開始する際は、常に `Current::begin()` を使用します。

Current は、プロセス全体に対して有効なオブジェクトとして、各スレッドのトランザクションコンテキストの関連付けを管理します。各スレッドは、1 つのトランザクションコンテキストと個別に関連付けられています。

VisiTransact 管理のトランザクションでは、VisiTransact がトランザクションコンテキストを各参加者に透過的に転送するため、トランザクションの参加者が同じトランザクションコンテキストを共有します。つまり、オリジネータがほかのオブジェクトにアクションの実行を要求し、結果としてほかのオブジェクトが呼び出されても、トランザクションの状態は維持されます。

図 7.1　VisiTransact がトランザクションコンテキストを転送してトランザクションを管理するしくみ



1　トランザクションオリジネータは、オブジェクト A に doWork() メソッドを実行するように要求します。
2　オブジェクト A は、オブジェクト B に doMoreWork() メソッドを実行するように要求します。
3　オブジェクト B は、その結果をオブジェクト A に戻します。
4　オブジェクト A は、その結果をトランザクションオリジネータに戻します。

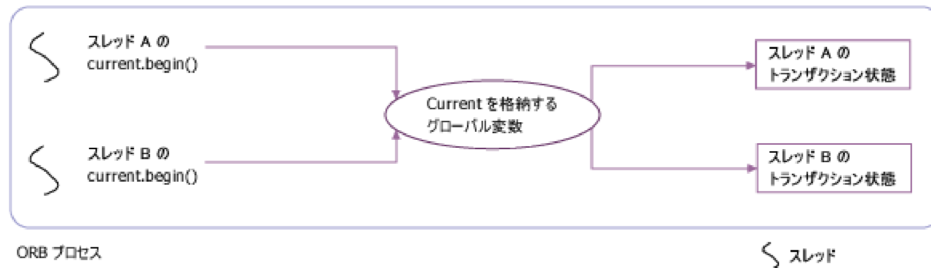
VisiTransact Transaction Service は、この 4 つすべての手順で、トランザクションオブジェクト間でトランザクションコンテキストを自動的および透過的に伝達します。上の図のステップ 1 から 2 に示すように、最初のトランザクションオブジェクトが別のオブジェクトに対する後続の要求を作成すると、トランザクションコンテキストは、この 2 番めのオブジェクトに移動します。オブジェクトがトランザクションオブジェクトでない場合は、コンテキストを受け取らないため、どのオブジェクトにもコンテキストは転送されません。

Current の機能

VisiTransact 管理のトランザクションは、Current オブジェクトを使って実行されます。Current インターフェースは、ほとんどのアプリケーションでトランザクション管理を容易にするメソッドを定義します。

Current インターフェースは、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コンテキストに依存して動作する擬似オブジェクト（その動作によってトランザクションコンテキストが変更されることもある）によってサポートされます。Current は CORBA オブジェクトではないため、リモートにアクセスすることはできません。

図 7.2　グローバルな Current オブジェクトを使ってプロセス内の複数のスレッドにまたがるトランザクションを開始する方法



begin() メソッドを使って作成された新しいトランザクションは、そのメソッドを呼び出した特定のスレッドに関連付けられます。スレッドは、一度に 1 つのトランザクションだけに関連付けることができます。スレッドが終了した場合、つまりトランザクションオリジネータのスレッドがトランザクションを完了しないで制御を戻した場合、そのスレッドに関連付けられたままのアクティブなトランザクションがあれば、タイムアウトになってロールバックされます。

メモ　Current オブジェクトを使用する場合、アプリケーションは、クリティカルセクションを実装してスレッド間の同期を保証する必要はありません。

Current オブジェクトリファレンスの取得

VisiTransact 管理のトランザクションにアクセスするには、Current へのオブジェクトリファレンスを取得する必要があります。Current オブジェクトリファレンスは、プロセスを通じて有効です。次に、現在のオブジェクトへのリファレンスを取得するための一般的な手順とサンプルコードを示します。

1 ORB resolve_initial_references() メソッドを呼び出します。このメソッドは、Current オブジェクトへのリファレンスを取得します。

2 戻されたオブジェクトを CosTransactions::Current または VISTransactions::Current object にナローイングします。

CosTransactions::Current にナローイングする場合は、CosTransactions モジュールから提供される元の一連のメソッドを使用するように指定します。VISTransactions::Current にナローイングする場合は、VisiTransact から提供される Current インターフェースの一連のメソッドと拡張機能を使用するように指定します。

VisiTransact による Current インターフェースの拡張機能の詳細については、65 ページの「Current インターフェースの拡張機能」を参照してください。

次のサンプルは、C++ の例を示します。

```
// OMG 準拠のメソッドと動作を使用します。
CORBA::Object_var
    obj = orb->resolve_initial_references("TransactionCurrent");
CosTransactions::Current_var
  current = CosTransactions::Current::_narrow(obj);
// CosTransactions メソッドの OMG 動作を使用し、
// 追加の VisiTransact メソッドも使用します。
CORBA::Object_var
obj = orb->resolve_initial_references("TransactionCurrent");
VISTransactions::Current_var
  current = VISTransactions::Current::_narrow(obj);
```

Current インターフェースとそのメソッドの使い方

Current インターフェースには、現在のスレッドまたはコンテキストのトランザクションを管理するためのメソッドが用意されています。次の表に、これらのメソッドの説明をまとめます。

VisiTransact による Current インターフェースの拡張機能の詳細については、65 ページの「Current インターフェースの拡張機能」を参照してください。

メソッド	説明
begin()	新しいトランザクションを作成します。 すでにトランザクションが実行中の場合は、SubtransactionsUnavailable 例外が生成されます。 作成されたトランザクションのタイムアウトは、 最後に呼び出された set_timeout() に基づいて設定されます。set_timeout() が発行されていない場合は、 VisiTransact Transaction Service のデフォルトのタイムアウト値が使用されます。
commit(in boolean report_heuristics)	トランザクションを完了します。 このメソッドは、 オリジネータだけが呼び出すことができます。 コミットできなかった場合、トランザクションはロールバックされます。
rollback()	トランザクションをロールバックします。 このメソッドは、オリジネータだけが呼び出すことができます。
rollback_only()	ロールバックされるようにトランザクションを変更します。このメソッドは、トランザクションがロールバックされるように、 オリジネータではなく参加者によって使用されます。

メソッド	説明
get_status()	トランザクションの状態を戻します。 実行中のトランザクションがない場合は、StatusNoTransaction 値が戻されます。
get_transaction_name()	トランザクション名を戻します。 これは、 VisiTransact Transaction Manager またはユーザーによってトランザクションに割り当てられた説明のための文字列です。 実行中のトランザクションがない場合は、空の文字列が戻されます。
set_timeout()	実行中の新しいトランザクションが完了するまでのタイムアウトを設定します。 タイムアウトを 0 に設定すると、この後で開始されるトランザクションのタイムアウトを、そのトランザクションが使用する VisiTransact Transaction Service インスタンスのデフォルトタイムアウトに設定します。タイムアウトが 0 より大きい場合は、新しいタイムアウトを指定された秒数に設定します。 seconds パラメータが、使用される VisiTransact Transaction Service インスタンスの最大タイムアウト値を超える場合、新しいタイムアウトはその最大値に設定されて、 範囲内に収められます。 この後でプロセス内の任意のスレッドで begin() を呼び出すことによって作成されたトランザクションが、確立されたタイムアウトを過ぎてもトランザクションの完了を開始できない場合、トランザクションはロールバックされます。 それ以外の場合、 タイムアウトは無視されます。このタイムアウトは、すでに実行中のトランザクションには影響を及ぼしません。
get_control()	現在、プロセスまたはスレッドに関連付けられているトランザクションコンテキストを表す Control オブジェクトを戻します。この Control オブジェクトを使用して、このトランザクションコンテキストが一時停止されている場合にトランザクションコンテキストを再開したり、 明示的な伝達を実行することができます。
suspend()	現在のトランザクションを一時停止します。 このメソッドは Control オブジェクトを戻します。これは、現在、プロセスまたはスレッドに関連付けられているトランザクションコンテキストを表します。このオブジェクトを使用して、このトランザクションコンテキストを再開できます。
resume()	一時停止されているトランザクションを再開するか、トランザクションコンテキストをプロセスまたはスレッドに関連付けます。

メモ　get_control()、suspend()、または resume() を使用する場合は、checked behavior に影響する場合があります。詳細については、76 ページの「VisiTransact Transaction Service が checked behavior を実行するしくみ」を参照してください。

次のサンプルに示すように、上の表にあるメソッドを使用して、VisiTransact 管理のトランザクションに対するアクションを実行できます。次のサンプルは、withdraw() メソッドを定義する、トランザクションオブジェクト用の **MyBank** インターフェースを示します。

```
#include <CosTransactions.idl>

interface MyBank
{
   float    balance(in long  accountNo);
   boolean     withdraw(in long accountNo, in float amount);
};
```

次のサンプルは、オリジネータがトランザクションを開始し、**MyBank** トランザクションオブジェクトの withdraw() メソッドを呼び出す例です。オリジネータは、トランザクションをコミットまたはロールバックします。

```
...
   // オブジェクトインプリメンテーションへのオブジェクトリファレンスを取得します。
   MyBank_var bank = MyBank::_bind();

   // トランザクションを開始します。
   current->begin();

   if(bank->withdraw(10, 444))
   {
    // CORBA 要求を呼び出します。
    current->commit(0);
   }
   else
   {
    current->rollback();
   }
...
```

トランザクションを開始したオリジネータは、そのトランザクションをコミットまたはロールバックする必要があります。VisiTransact Transaction Service は、トランザクションがタイムアウトになるとロールバックします。たとえば、トランザクションがコミットまたはロールバックされる前にオリジネータのスレッドが停止した場合は、トランザクションがタイムアウトになります。

同じトランザクションに関与する複数のスレッド

1 つのプロセスがあり、同じトランザクションで複数のスレッドを使用する場合は、各スレッドにトランザクションコンテキストを渡す必要があります。通常は、トランザクションコンテキストを持つスレッドから始まります。それは、このスレッドがオリジネータであり、`Current::begin()` を呼び出したからか、あるオペレーションがそのスレッドにトランザクションコンテキストを（暗黙的または明示的に）渡し、そのコンテキストをほかのスレッドに伝達する必要があるからです。それには、トランザクションの Control オブジェクトをほかのスレッドで使用できるようにします。それらのスレッドは、その Control オブジェクトを指定して `Current::resume()` を呼び出すことができます。この場合、VisiTransact は、checked behavior を提供できません。

コンテキストまたはスレッド内の複数トランザクションの使用

メモ　このリリースの VisiTransact Transaction Service では、ネストしたトランザクションはサポートされません。ただし、ここで説明する手順を使用すると、1 つのスレッドまたはコンテキストで複数のトランザクションを実行できます。

1 つのスレッド内で複数のトランザクションを管理できますが、各スレッドは、アクティブなトランザクションを一度に 1 つしか持つことができません。現在のコンテキストの関連付けを解除するには、`suspend()` メソッドを使用します。別のコンテキストを関連付けるには、`resume()` メソッドを使用します。59 ページの「Current インターフェースとそのメソッドの使い方」の表では、スレッド内で複数のトランザクションを実装するために使用されるメソッドについて説明しています。

次のサンプルは、1 つのスレッド内から複数のトランザクションを開始するオブジェクトの例です。このサンプルでは、`MyBank_impl::withdraw()` メソッドが、そのメソッドを呼び出したトランザクションを一時停止し、新しいトランザクションを開始し、最初のトランザクションを再開しています。

```
CORBA::Boolean MyBank_impl::withdraw(  CORBA::Long accountNo,
                                       CORBA::Float amount)
{
  try
  {
   // トランザクションが開始されているかどうかを確認します。
   CORBA::Object_var
```

```
    obj = orb->resolve_initial_references("TransactionCurrent");
   CosTransactions::Current_var
    current = CosTransactions::Current::_narrow(obj);
   // 現在のトランザクションを一時停止します。現在のトランザクションがない
場合、control は null になります。
   CosTransactions::Control_var   control = current->suspend();
   // 新しいトランザクションを開始します。
   try
   {
    current->begin();
    // ロジックを実行します。
    current->commit(0);
   }
   catch(...)
   {
    // 最初のトランザクションを再開します。
    current->resume(control);
    throw;
   }
  }
  catch(..) { }
}
```

VisiTransact Transaction Service のインスタンスの検索

デフォルトでは、begin() を使って初めてトランザクションを開始する際に、スマートエージェントを使って VisiTransact Transaction Service のインスタンスが検索されます。スマートエージェントの詳細については、『*VisiBroker 開発者ガイド*』を参照してください。

使用される VisiTransact Transaction Service のインスタンスは、ORB_init() に渡される引数や VISTransactions::Current インターフェースの引数を設定して制御できます。Current の引数は、ORB_init() に渡されるどの引数より優先します。この引数は、Current::begin() を使用する後続のトランザクションにだけ適用されます。

次の引数を設定できます。

- **ホスト名。**スマートエージェントは、指定されたホストで、使用できる VisiTransact Transaction Service のインスタンスを探します。
- **VisiTransact Transaction Service 名。**スマートエージェントは、ネットワーク上で、指定された VisiTransact Transaction Service のインスタンスを探します。
- **IOR。**VisiTransact は、要求されたトランザクションサービスに対して指定された IOR（CosTransactions::TransactionFactory）を使用して、ネットワーク上でトランザクションサービスインプリメンテーションのインスタンスを探します。この引数により、VisiTransact は、スマートエージェント（**osagent**）を使用しなくても操作を実行できます。

ホスト名と VisiTransact Transaction Service 名を組み合わせて指定すると、スマートエージェントは、指定されたホストで指定された VisiTransact Transaction Service のインスタンスを探します。ホスト名または VisiTransact Transaction Service 名のどちらかで IOR を指定した場合、スマートエージェントは、VisiTransact Transaction Service のインスタンスを IOR だけで探します。つまり、ホスト名と VisiTransact Transaction Service 名は無視されます。

次の表に、VisiTransact Transaction Service のインスタンスを指定するために使用できる引数を示します。

特性	ORB_init() に渡すことができる引数	VISTransactions::Current インターフェース
ホスト名	-Dvbroker.ots.currentHost	ots_host

特性	ORB_init() に渡すことができる引数	VISTransactions::Current インターフェース
VisiTransact Transaction Service 名	-Dvbroker.ots.currentName	ots_name
IOR	-Dvbroker.ots.currentFactory	ots_factory

次のサンプルは、VISTransactions::Current インターフェースの ots_name 引数を使用して、VisiTransact Transaction Service のインスタンスを名前で指定します。

```
...
CORBA::Object_var obj =
  orb->resolve_initial_references("TransactionCurrent");
CosTransactions::Current_var current =
  VISTransactions::Current::_narrow(obj);

// VisiTransact Transaction Service のインスタンスを設定します。
current->ots_name("MyTxnSvc");
...
```

VisiTransact 管理のトランザクションの伝達

暗黙的な伝達を有効にするには、参加者がトランザクションオブジェクトである必要があります。このオブジェクトは、CosTransactions::TransactionalObject から継承するか、REQUIRE 値または ADAPT 値を使って OTSPolicy オブジェクトを定義する必要があります。トランザクションに別の参加者を追加するには、ほかの参加者を追加するオブジェクトが、現在のスレッドに関連付けられているトランザクションを持つ必要があります。

トランザクションは、次の 3 つの場合に現在のスレッドに関連付けられます。

- トランザクション内の参加者が別のオブジェクトからトランザクションコンテキストを暗黙的に受け取る場合。
- Current::begin() を使って新しいトランザクションが開始される場合。
- Current::resume() を使ってスレッドにトランザクションオブジェクトのコンテキストが関連付けられている場合。

トランザクションが実行中かどうかの確認

参加者がトランザクションを必要とする場合は、新しいトランザクションを開始する前に、トランザクションが現在実行されていないことを確認する必要があります。トランザクションがすでに実行されている場合に参加者が新しいトランザクションを開始しようとすると、VisiTransact Transaction Service は CosTransactions::SubtransactionsUnavailable 例外を生成します。新しいトランザクションを開始する参加者は、制御を戻す前に、そのトランザクションをロールバックまたはコミットする必要があります。

次のサンプルは、サーバーオブジェクトがトランザクションとして作業を実行し、トランザクションがすでに実行されている場合は新しいトランザクションを開始しないようにする方法を示します。

```
CORBA::Boolean  MyBank_impl::withdraw( CORBA::Long accountNo,
                    CORBA::Float amount)
{
  // ORB インスタンスを取得します。
  CORBA::ORB_ptr orb =  CORBA::ORB_init();

  // 現在のリファレンスを取得します。
  CORBA::Object_var
    obj = orb->resolve_initial_references("TransactionCurrent");
  CosTransactions::Current_var
    current = CosTransactions::Current::_narrow(obj);
```

```
  CORBA::Boolean startFlag = 0;// トランザクションの作成の通知に使用
  CORBA::Boolean status = 0;
  try
  {
   // トランザクションが開始されているかどうかを確認します。
   if(current->get_status() == CosTransactions::StatusNoTransaction)
   {
    current->begin();
    startFlag = 1; // 現在のトランザクションを開始して所有
   }
   if(balance(accountNo) > amount)
   {
    // 引き出しロジック
    ...
    status = 1;
   }
  }
  catch(...) { }
  if(startFlag && status)
  {
   current->commit();
  }
  else if(startFlag)
  {
   current->rollback();
  }
  return status;
}
```

ロールバックするトランザクションのマーク

Current を使用する場合は、オリジネータだけが `commit()` または `rollback()` を使ってトランザクションを完了できます。このとき、参加者がトランザクションをコミットしない場合は、`Current` インターフェースの `rollback_only()` メソッドを使用できます。参加者によって `rollback_only()` メソッドが呼び出されると、ターゲットオブジェクトに関連付けられているトランザクションは、ロールバックだけを実行できるように変更されます。

実行中のトランザクションがない場合に `rollback_only()` を呼び出すと、`CosTransactions::NoTransaction` 例外が生成されます。次のサンプルは、参加者が `rollback_only()` メソッドを使用する方法を示します。

```
...
CosTransactions::Current_var current;
current->rollback_only();
...
```

トランザクション情報の取得

参加者は、`Current` インターフェースのメソッドを使用して、トランザクション名やトランザクションの状態などの現在のトランザクションに関する情報を取得できます。次の表で、このメソッドについて説明します。

メソッド	説明
`get_status()`	現在のスレッドに関連付けられているトランザクションの状態を戻します。
`get_transaction_name()`	現在のスレッドに関連付けられているトランザクションについて説明する出力可能な文字列を戻します。

get_status() メソッドは、次のいずれかの値を戻すことができます。

- StatusActive
- StatusCommitted
- StatusCommitting
- StatusMarkedRollback
- StatusNoTransaction
- StatusPrepared
- StatusPreparing
- StatusRolledBack
- StatusRollingBack
- StatusUnknown

Current インターフェースの拡張機能

VisiTransact には、VisiTransact Transaction Service のインスタンスを指定するための引数を提供する拡張インターフェースと追加メソッドが用意されています。VISTransactions::Current の引数の詳細については、62 ページの「VisiTransact Transaction Service のインスタンスの検索」を参照してください。次の表に、**VISTransactions.idl** ファイルに含まれる VisiTransact の拡張 Current インターフェースのメソッドを示します。Current インターフェースの詳細については、49 ページの「Current インターフェース」を参照してください。

メソッド	説明
begin_with_name()	ユーザーが定義した説明のためのトランザクション名を渡すことができます。たとえば、get_transaction_name() メソッドから戻される値に、このユーザー定義のトランザクション名が含まれるため、診断が容易になります。また、コンソールは、未処理のトランザクションに関する詳細情報として、この名前を使用するため、管理が容易になります。
get_txcontext()	PropagationContext を戻します。 この PropagationContext を VisiTransact Transaction Service ドメインで使用して、トランザクションを別の VisiTransact Transaction Service ドメインにエキスポートできます。
register_resource()	回復可能なオブジェクトのリソースを登録します。このメソッドは、Control オブジェクトと Coordinator オブジェクトを使用して、回復可能なオブジェクトのリソースを登録するためのショートカットです。このメソッドは、回復の調整に使用される回復コーディネータオブジェクトを戻します。通常、ほとんどのアプリケーションではこのメソッドを呼び出しません。リソースの詳細については、81 ページの「リソースオブジェクトによる トランザクションの完了の調整」を参照してください。
register_synchronization()	同期オブジェクトを登録します。このメソッドは、Control オブジェクトと Coordinator オブジェクトを使用して、Synchronization オブジェクトを登録するためのショートカットです。同期オブジェクトの詳細については、93 ページの「同期オブジェクトの実装」を参照してください。
get_otid()	Current インターフェースを介してオブジェクトトランザクション ID（otid）を提供するので便利です。これにより、Coordinator を介して PropagationContext を調べる必要がなくなります。otid を使用して、回復可能なオブジェクトに対するトランザクションを特定します。通常、ほとんどのアプリケーションではこのメソッドを呼び出しません。

第 8 章

トランザクションを作成および伝達するほかの方法

ここでは、トランザクションの管理に使用できるその他の機能について説明します。具体的には、VisiTransact Transaction Service のインターフェース（`TransactionFactory`、`Control`、`Coordinator`、`Terminator`）の使用方法について説明します。

はじめに

通常、トランザクションの管理には `Current` インターフェースを使用しますが、次の方法も使用できます。

- **明示的な伝達による間接的なコンテキスト管理** クライアントは、Current オブジェクト、Control オブジェクト、およびトランザクションの状態を記述するほかのオブジェクトを組み合わせて使用します。`Current` オブジェクトを使用するクライアントアプリケーション（自動的に暗黙的な伝達も使用する）は、`Current::getControl()` メソッドによって Control オブジェクトへのアクセスを取得することで、明示的な伝達を使用できます。VisiTransact Transaction Service オブジェクトをトランザクションオブジェクトへの明示的なパラメータとして使用できます。これは、明示的な伝達です。
- **暗黙的な伝達による直接的なコンテキスト管理** クライアントは、Current オブジェクト、Control オブジェクト、およびトランザクションの状態を記述するほかのオブジェクトを組み合わせて使用します。VisiTransact Transaction Service オブジェクトに直接アクセスするクライアントは、`Current::resume()` メソッドを使用して、そのスレッドに関連付けられている暗黙的なトランザクションコンテキストを設定できます。これにより、クライアントは、トランザクションコンテキストを暗黙的に伝達する必要があるオブジェクトのメソッドを呼び出すことができます。
- **明示的な伝達による直接的なコンテキスト管理** クライアントアプリケーションは、Control オブジェクトとトランザクションの状態を記述するほかのオブジェクトに直接アクセスします。トランザクションをオブジェクトに伝達するため、クライアントは、メソッドの明示的なパラメータとして適切な VisiTransact Transaction Service オブジェクトを含める必要があります。

上の方法でトランザクションを管理するには、次のインターフェースを使用します。

- TransactionFactory。トランザクションオリジネータがトランザクションを開始するためのメソッドを定義します。TransactionFactory インターフェースについては、68 ページの「TransactionFactory によるトランザクションの作成」を参照してください。
- Control。アプリケーションが明示的にトランザクションコンテキストを管理または伝達できるようにします。Control インターフェースについては、69 ページの「Control オブジェクトによるトランザクションの制御」を参照してください。
- Terminator。アプリケーションがトランザクションをコミットまたはロールバックできるようにします。通常、これらのメソッドは、トランザクションオリジネータによって使用されます。ただし、Control オブジェクトまたは Terminator オブジェクトを伝達することで、任意のトランザクション参加者がトランザクションをコミットまたはロールバックできます。Terminator インターフェースについては、72 ページの「Terminator によるトランザクションのコミットまたはロールバック」を参照してください。
- Coordinator。参加者がトランザクションの状態の判定、トランザクション名の検出、トランザクションコンテキストの取得を実行できるようにします。また、トランザクションがトランザクションオリジネータ以外の参加者からロールバックされることを指定できます。Coordinator インターフェースのメソッドについては、73 ページの「ロールバックするトランザクションのマーク」と 73 ページの「トランザクション情報の取得」を参照してください。

TransactionFactory によるトランザクションの作成

TransactionFactory インターフェースを使用して、トランザクションオリジネータはトランザクションを開始できます。次の例に示すように、この CosTransactions のインターフェースには、2 つのメソッド create() と recreate() が用意されています。create() メソッドは、新しいトランザクションの開始に使用されます。recreate() メソッドは、伝達コンテキストからトランザクションの Control オブジェクトを作成するために使用されます。ただし、通常のアプリケーションからは使用されないのが普通です。

```
module CosTransactions
{
 interface TransactionFactory
 {
  Control create(in unsigned long time_out);
  Control recreate(in PropagationContext ctx);
 };
};
```

VisiTransact は、TransactionFactory インターフェースの拡張機能として、特定の名前を使ってトランザクションを作成できる create_with_name() も提供します。トランザクションに名前を付けると、特定のトランザクションの進行状況の追跡や実行時のデバッグに役立ちます。

```
module VISTransactions
{
 // TransactionFactory
 //   CosTransactions::TransactionFactory を拡張します。
 //   ユーザー定義の名前を使ってトランザクションを作成し、
 //   デバッグやエラー報告などに使用できます。

 interface TransactionFactory : CosTransactions::TransactionFactory
 {
  CosTransactions::Control create_with_name(in unsigned long time_out,
                 in string userTransactionName);
 };
};
```

次の表は、TransactionFactory を使ってトランザクションを作成するためのメソッドを定義します。

メソッド	説明
create(in unsigned long time_out)	新しいトランザクションを作成し、 Control オブジェクトを戻します。この Control オブジェクトを使用して、新しいトランザクションへの参加を管理できます。time_out に 0 秒を設定すると、VisiTransact Transaction Service のインスタンスのデフォルトのタイムアウトが使用されます。
create_with_name(in unsigned long time_out, in string userTransactionName)	userTransactionName 引数で指定されたユーザー定義の名前で新しいトランザクションを作成します。
recreate(in PropagationContext ctx)	PropagationContext（トランザクションコンテキスト）の定義にしたがって既存のトランザクションの新しい表現を作成し、Control オブジェクトを戻します。 この Control オブジェクトを使用して、新しいトランザクションへの参加を管理または制御できます。

TransactionFactory インターフェースの詳細については、「TransactionFactory インターフェース」を参照してください。

次の例は、デフォルトのタイムアウト期間を使って新しいトランザクションを開始する方法を示します。

```
...
CosTransactions::TransactionFactory_var txnFactory;
CosTransactions::Control_var control;
control = txnFactory->create_with_name(0,"BankTransfer#1");
                    // デフォルトを使用します。
                    // タイムアウト値
...
```

メモ　PropagationContext は、CosTransactions:Coordinator::get_txcontext() メソッドを使って既存のトランザクションから取得できます。このメソッドについては、73 ページの「トランザクション情報の取得」を参照してください。

Control オブジェクトによるトランザクションの制御

Control インターフェースを使用して、アプリケーションは、トランザクションコンテキストを明示的に管理または伝達するための Terminator オブジェクトや Coordinator オブジェクトへのリファレンスを取得できます。Control インターフェースをサポートするオブジェクトは、固有のトランザクションに関連付けられます。

次に、Control インターフェースを示します。

```
module CosTransactions
{
 interface Control
 {
  Terminator get_terminator()
   raises(Unavailable);
  Coordinator get_coordinator()
   raises(Unavailable);
 };
};
```

次の表は、Control インターフェースのメソッドを定義します。

メソッド	説明
get_terminator()	トランザクションを終了するためのオペレーションをサポートする Terminator オブジェクトを戻します。Terminator オブジェクトを使用して、 Control オブジェクトに関連付けられたトランザクションをロールバックまたはコミットできます。Control オブジェクトが Terminator オブジェクトを提供できない場合は、CosTransactions::Unavailable 例外が生成されます。
get_coordinator()	リソースがトランザクションに参加するために必要なオペレーションをサポートする Coordinator オブジェクトを戻します。 Coordinator オブジェクトを使用して、 Control オブジェクトに関連付けられたトランザクションのリソースを登録できます。 Control オブジェクトが Coordinator オブジェクトを提供できない場合は、CosTransactions::Unavailable 例外が生成されます。

Terminator オブジェクトと Coordinator オブジェクトへのリファレンスを取得するには、オリジネータのコードに次のようなステートメントを入れます。ほとんどのメソッドはこれらのオブジェクトの一方だけを必要とするため、これらのオブジェクトは分けられています。

```
...
CosTransactions::Control_var control
CosTransactions::Terminator_var newTranTerminator;
CosTransactions::Coordinator_var newTranCoordinator;

newTranTerminator = control->get_terminator();
newTranCoordinator = control->get_coordinator();
...
```

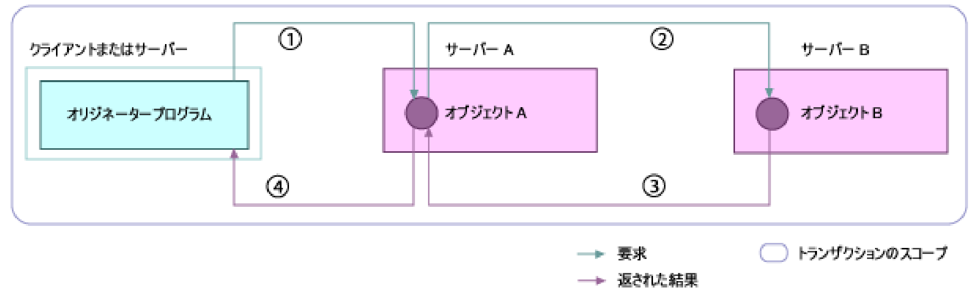
オリジネータからの明示的なトランザクションの伝達

TransactionFactory を使って開始されたトランザクションの場合、トランザクションオリジネータは、いくつかの VisiTransact Transaction Service インターフェースを使ってトランザクションを処理します。これらのインターフェースを介して、トランザクションオリジネータは、一度に複数のトランザクションを管理できます。

これらのタイプのトランザクションでは、オリジネータがすべてのオペレーションにおいて IDL シグニチャに含まれる明示的なパラメータを介して各参加者にトランザクションコンテキストを転送するため、トランザクションの参加者は同じトランザクションコンテキストを共有します。つまり、オリジネータがほかのオブジェクトにアクションの実行を要求し、結果として同じパラメータを使ってほかのオブジェクトが呼び出されても、トランザクションの状態は維持されます。次の図は、メソッドの呼び出し内からトランザクション参加者間で渡されるコンテキストを表します。

メモ TransactionFactory を使って開始されたトランザクションの場合は、暗黙的な伝達を使用できます。71 ページの「明示的な伝達から暗黙的な伝達への変更」を参照してください。

図 8.1　トランザクションコンテキストが明示的に伝達されるしくみ



1. トランザクションオリジネータは、オブジェクト A が `doWork()` メソッドを実行するように要求し、同時に Control オブジェクトまたは Coordinator オブジェクトを渡します。
2. オブジェクト A は、オブジェクト B が `doMoreWork()` メソッドを実行するように要求し、同時に Control オブジェクトまたは Coordinator オブジェクトを渡します。これで、オブジェクト B は既存のトランザクションの一部として動作できます。
3. オブジェクト B は、その結果をオブジェクト A に戻します。
4. オブジェクト A は、その結果をトランザクションオリジネータに戻します。

トランザクションをトランザクションの参加者に明示的に伝達するため、オリジネータは、トランザクションオブジェクトのリモート呼び出しの明示的なパラメータとして Control、Coordinator、または Terminator オブジェクトを設定する必要があります。

- Terminator オブジェクトを渡す場合、参加者には、トランザクションを終了するという制限された機能が与えられます。それ以外の機能は実行できません。
- Coordinator オブジェクトを渡す場合、リモートオブジェクトはトランザクションの参加者になることができますが、トランザクションを終了するための機能は与えられません。Coordinator を渡すと、リモートオブジェクトはトランザクションをロールバックできるようになります。
- Control オブジェクト を渡す場合、参加者には、Coordinator オブジェクト と Terminator オブジェクト の機能が与えられます。

次の例では、Control オブジェクト **control** がリモートトランザクションオブジェクトの `withdraw()` メソッドに明示的なパラメータとして渡されています。

```
...
CosTransactions::Control_var control;
CORBA:Boolean didSucceed;
didSucceed=bank->withdraw(10, 444, control) // CORBA 要求を呼び出します。
...
```

明示的な伝達から暗黙的な伝達への変更

トランザクションを明示的な伝達で開始した後で、暗黙的な伝達に切り替えることができます。暗黙的なトランザクションコンテキストを設定するには、Control オブジェクトを `Current::resume()` に渡します。`Current::resume()` と `Current::suspend()` の使用方法については、57 ページの「VisiTransact 管理のトランザクションの作成および伝達」と61 ページの「コンテキストまたはスレッド内の複数トランザクションの使用」を参照してください。

Current からの明示的なコンテキストの取得

トランザクションを暗黙的な伝達で開始した後でトランザクションコンテキストを明示的に取得するには、`Current::get_control()` を使用します。

Terminator によるトランザクションのコミットまたはロールバック

`Terminator` インターフェースは、トランザクションをコミットまたはロールバックするためのオペレーションをサポートします。通常、これらのオペレーションはトランザクションオリジネータによって使用されます。次に、`Terminator` インターフェースを示します。

```
module CosTransactions
{
  interface Terminator
  {
    void commit(in boolean report_heuristics)
      raises (HeuristicMixed, HeuristicHazard);
    void rollback();
  };
};
```

次の表は、`Terminator` インターフェースのメソッドを定義します。

メソッド	説明
`commit (in boolean report_heuristics)`	トランザクションに「ロールバックのみ」のマークが付けられておらず、トランザクションのすべての参加者がコミットに同意する場合は、トランザクションをコミットします。それ以外の場合は、トランザクションがロールバックされ、例外 `CORBA::TRANSACTION_ROLLEDBACK` が生成されます。トランザクションがコミットされると、このトランザクションの間に回復可能なオブジェクトに加えられた変更がすべて確定され、ほかのトランザクションやクライアントから認識できるようになります。 `report_heuristics` パラメータが true の場合、VisiTransact Transaction Service は、例外 `CosTransactions::HeuristicMixed` と `CosTransactions::HeuristicHazard` を使用して、矛盾した（またはその可能性がある）結果が生成されることを報告します。
`rollback()`	トランザクションをロールバックします。トランザクションがロールバックされると、このトランザクションの間に回復可能なオブジェクトに加えられた変更がすべてロールバックされます。

次の例は、オリジネータが動作を実行するためにアクセスするトランザクションオブジェクトの **MyBank** インターフェースを示します。

```
#include <CosTransactions.idl>

interface MyBank {
  float      balance(in long  accountNo,
          in CosTransactions::Coordinator coord);
  boolean      withdraw(in long accountNo,
          in float amount,
          in CosTransactions::Control control);
};
```

次の例は、**MyBank** トランザクションオブジェクトが関係するトランザクションをオリジネータがコミットまたはロールバックする方法を示します。この例は、withdraw() メソッド内でトランザクションを処理する場合です。balance() メソッドは、Coordinator が渡されるだけなので、トランザクションを終了できません。

```
...
CORBA::Boolean didSucceed;
...
 CosTransactions::Terminator_var
  txnTerminator=control->get_terminator();

  if(didSucceed)
  { // CORBA 要求を呼び出します。
   try
   {
    txnTerminator->commit(1);
   }
   catch(CORBA::TRANSACTION_ROLLEDBACK&)
   {
    // 失敗を戻します。
   }
  }
  else
  {
   txnTerminator->rollback();
  }
...
```

トランザクションをコミットする際のヒューリスティックな完了については、78 ページの「ヒューリスティックな完了」を参照してください。

ロールバックするトランザクションのマーク

参加者がトランザクションをコミットしない場合は、Coordinator インターフェースの rollback_only() メソッドを使用できます。参加者によって rollback_only() メソッドが呼び出されると、現在のスレッドに関連付けられているトランザクションは、ロールバックだけを実行できるように変更されます。トランザクションの準備が完了している場合は、CosTransactions::Inactive 例外が生成されます。次のサンプルは、参加者が rollback_only() メソッドを使用する方法を示します。

```
...
CosTransactions::Coordinator_var coord;
coord->rollback_only();
...
```

トランザクション情報の取得

参加者は、Coordinator インターフェースのメソッドを使用して、トランザクションの名前やトランザクションの状態などのトランザクションに関する情報を取得できます。また、トランザクションのトランザクションコンテキストも取得できます。次の表で、このメソッドについて説明します。

メソッド	説明
get_status()	現在のスレッドに関連付けられているトランザクションの状態を戻します。
get_transaction_name()	現在のスレッドに関連付けられているトランザクションについて説明する出力可能な文字列を戻します。
get_txcontext()	PropagationContext オブジェクトを戻します。

`get_status()` メソッドは、次のいずれかの値を戻すことができます。

- `StatusActive`
- `StatusCommitted`
- `StatusCommitting`
- `StatusMarkedRollback`
- `StatusNoTransaction`
- `StatusPrepared`
- `StatusPreparing`
- `StatusRolledBack`
- `StatusRollingBack`
- `StatusUnknown`

第 9 章

トランザクションの完了

ここでは、トランザクションの完了、ヒューリスティックな完了、およびマルチスレッドアプリケーションについて説明します。

トランザクションの完了

トランザクションの完了は、VisiTransact Transaction Service がトランザクション型アプリケーションの作業をコミットまたはロールバックする要求を受け取ったときに実行する一連の手順です。完了の要求は、次のさまざまな状況で開始されます。

- トランザクションオリジネータが `commit()` または `rollback()` を呼び出して完了を開始した。
- トランザクションタイムアウトが発生し、完了がトリガーされた。
- VisiTransact Transaction Service の回復時に、（ログレコード内に見つかった）未完了のトランザクションが再インスタンス化され、トランザクションの完了が再開された。

VisiTransact Transaction Service が完了を実行するしくみ

トランザクションオリジネータがトランザクションのコミットまたはロールバックを要求すると、VisiTransact Transaction Service は、そのトランザクションの完了手順を開始します。1 つのトランザクションに関係する 2 つのリソースがあるとします。VisiTransact Transaction Service は、コミットの要求を受け取ると、2 フェーズコミットの手順を開始して完了を調整します。

2 フェーズコミットの手順がエラーなく実行されてトランザクションが完了した場合、オリジネータは、結果の通知を受け取ります。2 フェーズのコミットフェーズで一方のリソースが使用できなかったなどの理由でトランザクションが完了できなかった場合、VisiTransact Transaction Service は、そのトランザクションを完了できず、後で試行するためにトランザクションを再試行キューに入れます。再試行キューに入れられたトランザクションは、完了のためにただちにディスパッチされるわけではありません。システムパフォーマンスの低下を防ぐため、各再試行の間には、遅延がプログラムされています。再試行の間隔は、最小で 15 秒、最大で 900 秒です。最初の再試行は 15 秒後に開始され、その後の再試行では、900 秒まで遅延時間が増加していきます。その後は、900 秒ごとに再試行が行われます。タイムアウトまたは回復が原因で再試行される場合は、15 秒の遅延を待つことなく最初の再試行がただちにディスパッチされます。再試行が行われている間、VisiTransact Transaction Service は、トランザクションのうち、まだ完了していない部分だけを実行します。そのトランザクションは、完了するか、VisiBroker コンソールから Stop Completion コマンドが発行されるまで、再試行キューに残ります。コンソールでトランザ

クションのリストを照会した場合、何回か再試行されたトランザクションは強調表示されます。

再試行は次の場合に行われます。

- **トランザクションタイムアウトが発生した。**トランザクションにタイムアウトが指定されており、トランザクションがその制限内に完了しなかった場合、そのトランザクションは再試行キューに入れられます。タイムアウトになったとき、すでにトランザクションが完了段階に入っていた場合、VisiTransact Transaction Service はタイムアウトを無視します。VisiTransact Transaction Service には、コマンドラインでデフォルトのタイムアウトを設定できます。
- **リソースを使用できない。**通信の障害またはリソースサーバーのダウンにより、トランザクションに必要なリソースが一時的に使用できない場合です。トランザクションは、完了するまで再試行キューに入れられます。
- **VisiTransact Transaction Service が回復し、決定レコードがトランザクションの未完了を示している。**VisiTransact Transaction Service の回復中には、VisiTransact Transaction Service がダウンしたときに完了していなかったトランザクションに関する情報がトランザクションログから収集されます。決定レコードにトランザクションがまだ完了していないことが示されている場合、そのトランザクションは、完了のために再試行キューに入れられます。

VisiTransact Transaction Service が checked behavior を実行するしくみ

VisiTransact Transaction Service は、完全な DTP（Distributed Transaction Processing）の checked behavior を実装し、トランザクションの整合性をさらに強化しています。checked behavior は、アプリケーションによって行われたすべてのトランザクション要求が完全に完了したかどうかを確認してからトランザクションをコミットすることにより、データの整合性が失われることを防ぎます。これにより、すべてのトランザクション参加者がトランザクション要求の処理を完了しない限り、コミットが成功しないことが保証されます。すべての要求が同期している場合、checked behavior はデフォルトで行われます。

checked behavior は、遅延同期要求に関連する VisiTransact 管理のトランザクションに対して適用されます。この場合は、`commit()` が発行されたときに保留中の応答があると、トランザクションがロールバックされます。トランザクションオブジェクトの要求ハンドラが遅延同期要求を行い、遅延同期要求が戻る前に応答した場合、そのトランザクションはロールバック対象としてマークされます。

VisiTransact は、一方向リクエストに対しては checked behavior を実施しません。

次の例は、遅延同期要求があり、`commit()` を呼び出した後で応答が戻される場合の checked behavior のクライアントコードです。checked behavior は成功し、トランザクションはロールバックされます。

```
...
  // Current へのリファレンスを取得します。
  ...

  // トランザクションを開始します。
  current->begin();

  // 動的な要求を作成します。
  CORBA::Request_var bankRequest = bank->_request("withdraw");
  CORBA::NVList_ptr arguments = bankRequest->arguments();

  CORBA::Any_var    amt = new CORBA::Any();
  *amt<<= ((float)1000.00);

  arguments->add_value("amount", amt, CORBA::ARG_IN);

  ...

  // 遅延同期要求を行います。
  bankRequest->send_deferred();

  // 応答を取得しませんでした
  // トランザクションをコミットします。
  try
  {
   current->commit(0);
  }
  catch(CORBA::TRANSACTION_ROLLEDBACK& e)
  {
    cerr << "SUCCESS, commit check worked()" << endl;
  }
...
```

次の例は、遅延同期要求があり、`commit()` を呼び出す前に応答が戻される場合の checked behavior のクライアントコードです。checked behavior は成功し、トランザクションはコミットされます。

```
...
  // コミットの前に要求が到着した場合
  current->begin();

  cerr << " === Invoking a dii deferred sync request" << endl;
  bankRequest->send_deferred();

  try {
   // 応答を待機します。
   bankRequest->get_response();
   current->commit(0);
  }
  catch(CORBA::TRANSACTION_ROLLEDBACK& e)
  {
   cerr << "FAILURE, TRANSACTION_ROLLEDBACK not expected" << endl;
  }

 }
```

ヒューリスティックな完了

ヒューリスティックな完了は、トランザクションが完了を試みているときに、参加しているリソースの 1 つが完了段階でヒューリスティックな決定を行った場合に発生します。ヒューリスティックな決定とは、トランザクションマネージャが決定した結果を無視して、更新をコミットまたはロールバックする決定を 1 つ以上のリソースが一方的に行うことです。

一般にヒューリスティックな決定は、通常の処理を行うことができないネットワーク障害などの異常事態や、Coordinator がタイミングよく 2 フェーズコミットプロセスを完了しない場合にだけ発生します。ヒューリスティックな決定があった場合は、その決定が VisiTransact Transaction Manager による決定とは異なり、その結果、データの整合性が失われる危険があります。

ヒューリスティックな決定によってリソースから戻される例外は次のとおりです。

- **`HeuristicRollback`** - リソースに対するコミット操作で、ヒューリスティックな決定が行われ、関連するすべての更新がロールバックされたことを報告します。
- **`HeuristicCommit`** - リソースに対するロールバックオペレーションで、ヒューリスティックな決定が行われ、関連するすべての更新がコミットされたことを報告します。
- **`HeuristicMixed`** - リソースは、関連する一部の更新をコミットし、それ以外の更新をロールバックしました。
- **`HeuristicHazard`** - リソースは、関連する少なくとも 1 つの更新の結果を把握していません（関連する更新の結果が一部不明）。既知である更新は、すべてコミットされたか、すべてロールバックされています。

リソースは、2 フェーズコミット中の任意の時点でヒューリスティックな決定を行うことができます。たとえば、Terminator が 2 フェーズコミットをタイミングよく完了しない場合、リソースはヒューリスティックな決定を行うように選択できます。ヒューリスティックな決定は、リソースオブジェクトが 2 フェーズコミットプロセスの間に行った保証（`prepare()` の間に VoteCommit を戻したこと）を取り消す方法です。

ただし、リソースが Terminator に VoteCommit の応答を戻した後で、ヒューリスティックな決定を行った場合も、リソースには、トランザクションに関する自分の動作を報告する義務があります。Terminator が最終的にリソースにロールバックまたはコミットを要求した場合は、次の状況が考えられます。

- **その結果がヒューリスティックな決定と一致している。**この場合、トランザクションは正常に完了でき、リソースはトランザクションとヒューリスティックな決定に関する情報を破棄できます。ヒューリスティックな決定がトランザクションの結果と一致しているため、Terminator がヒューリスティックな決定に関する情報を受け取る必要はありません。
- **その結果がヒューリスティックな決定とは異なる。**この場合、リソースは、事前に安定なストレージに保存されているヒューリスティックによる結果のレコードを調べ、ヒューリスティックによる結果の例外（HeuristicCommit、HeuristicRollback、HeuristicMixed、HeuristicHazard）の 1 つを戻して完了を継続します。
- リソースが Coordinator からトランザクションに関する情報を破棄するように指示されるまで、ヒューリスティックによる結果の詳細は、安定なストレージに保存しておく必要があります。

アプリケーションへのヒューリスティック情報の通知

トランザクションオリジネータは、`commit()` メソッドの `report_heuristics` パラメータを `true` に設定することで、ヒューリスティック情報通知の受け取りを要求できます。次のサンプルコードの `commit()` メソッドは、C++ 向けに `commit(1)` として記述されています。

ヒューリスティック情報の通知

```
...
if (bank->withdraw(10,444))   //withdraw メソッドを呼び出します。
{
    try
    {
      current->commit(1);  // パラメータ 1 は、ヒューリスティックによる
                      // 結果があればそれを戻すようにサーバーに要求します。
     }
     catch (const CosTransactions::NoTransaction& e)
     {
         // トランザクションがないのにコミットが発行されました。
         // 処理します。

      }
      catch (const CosTransactions::HeuristicMixed& e)
      {
         // ヒューリスティックな決定が行われました。関連する更新には、
         // コミットされた部分とロールバックされた部分があります。
         // 処理します。
  }
  catch (const CosTransaction::HeuristicHazard& e)
     {
       // ヒューリスティックな決定が行われました。関連する更新は、
       // すべてコミットされたか、すべてロールバックされています。
       // 処理します。
     }
   }
   else
   {
     current->rollback();
   }
```

リソースは、ヒューリスティック情報通知をプログラムで処理するか、システム管理者による処置を要求できます。

OTS の例外

ほかに次の OTS 例外があります。

- **SubtransactionsUnavailable** - クライアントスレッドにすでに関連するトランザクションがあり、トランザクションサービスインプリメンテーションがネストしたトランザクションをサポートしない場合に生成されます。
- **NotSubtransaction** - 現在のトランザクションがサブトランザクションでない場合に生成されます。
- **Inactive** - 発行されたコマンドに対して現在のコンテキストが正しくなく、何もアクションが実行されない場合に生成されます。
- **NotPrepared** - トランザクションが準備されない場合に生成されます（2 フェーズコミットトランザクションのみ）。
- **NoTransaction** - クライアントスレッドにトランザクションが関連付けられていない場合に生成されます。
- **Unavailable** - アプリケーションが伝達コンテキストを入手できない場合に生成されます。
- **SynchronizationUnavailable** - システムが同期をサポートしない場合に生成されます。

第 10 章

リソースオブジェクトによる トランザクションの完了の調整

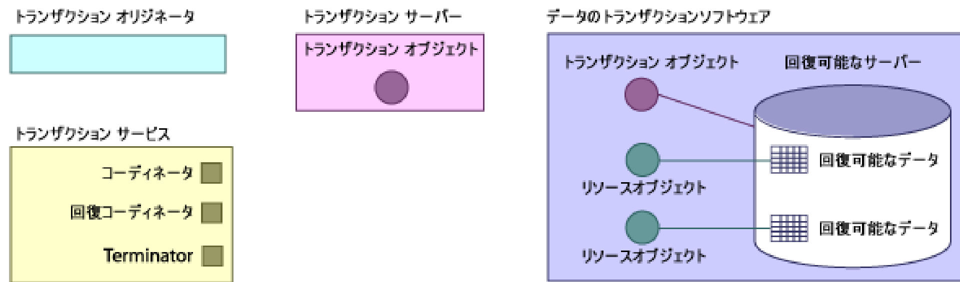
ここでは、リソースオブジェクトを使用して、1 フェーズコミットまたは 2 フェーズコミットに参加する方法について説明します。

トランザクションの完了の概要

15 ページの「基本的なトランザクションのモデル」で説明したトランザクションプロセスは、データを使用しない簡単な例でした。次の図は、データが伴うトランザクションに必要なオブジェクト（回復可能なサーバー、回復可能なオブジェクト、回復コーディネータ、リソースオブジェクト）を表すために前の例を拡張したものです。この図に示すように、実際には、これらのオブジェクトの一部がデータのトランザクションソフトウェアによってカプセル化されます。これらのオブジェクトを示すことで、背後で実行されているプロセスや IDL 内のインターフェースについて理解できます。

VisiTransact 管理のトランザクションを使用する場合、この図には、2 フェーズコミットを実行するために VisiTransact Transaction Service が使用するバックエンドオブジェクト（Coordinator、Terminator、および回復コーディネータ）も示されています。VisiTransact 管理のトランザクションを使用しない場合は、これらのオブジェクトを直接管理します。

図 10.1　2 フェーズコミットに関係するオブジェクト



次の表で、2 フェーズコミットに関係するオブジェクトについて説明します。

Object	説明
Coordinator	回復可能なオブジェクトをトランザクションに登録するために使用されます。 また、トランザクション間の調整を管理します。
Terminator	トランザクションの終了を調整します。すべての参加者がトランザクションの作業をコミットまたはロールバックするようにします。
回復可能なサーバー	1 つ以上のオブジェクトの集合。 少なくとも 1 つのオブジェクトが回復可能です。
回復可能なデータ	トランザクションの完了の影響を受ける内容を含むデータ（データベース内のテーブルなど）。回復可能なオブジェクトのすべてが CORBA インターフェースを実装しているわけではありません。
リソースオブジェクト	トランザクションの存続期間における VisiTransact Transaction Service と回復可能なオブジェクトの関係を表します。 トランザクションに参加している回復可能なオブジェクトごとに 1 つのリソースオブジェクトが必要です。
データのトランザクションソフトウェア	データベース、 ファイルシステム、 またはほかのサービス（Visibroker ネーミングサービスなど） でデータにアクセスするために使用されるオブジェクト（回復可能なサーバー、 回復可能なデータ、トランザクションオブジェクト、リソースオブジェクト） の集合。
回復コーディネータ	障害が発生した場合に、トランザクションの結果を判定したり、VisiTransact Transaction Service を使って回復プロセスを調整するために、リソースオブジェクトによって使用されます。

トランザクションの完了への参加

17 ページの「トランザクションの完了」は、2 フェーズコミットが、15 ページの「基本的なトランザクションのモデル」で扱われる簡単な例から分かれる場所です。VisiTransact Transaction Service は、2 フェーズコミットプロセスを実行する際に、トランザクション全体が原子性に基づいてロールバックされるかコミットされるようにします。2 フェーズコミットプロセスの最初のフェーズで、Terminator は、トランザクションの参加者にトランザクションをコミットする準備ができているかどうかをたずねます。すべての参加者がコミットできると応答した場合、Terminator は、すべての参加者に 2 番めのフェーズでトランザクションをコミットするように指示します。コミットの準備ができていないと応答した参加者が 1 つでもあると、Terminator は、参加者にトランザクションをロールバックするように指示します。

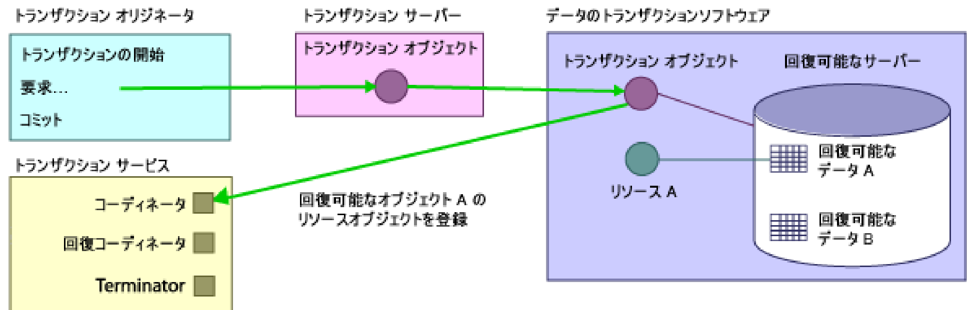
メモ　トランザクション型アプリケーションに 1 つのリソースだけが関係する場合、VisiTransact Transaction Service は、2 フェーズコミットプロセスではなく 1 フェーズコミットプロセスを開始します。

以下では、トランザクション完了の概念を細かく分けて、2 フェーズコミットプロセスについて説明します。

リソースオブジェクトがトランザクションに登録される

トランザクションに関係する回復可能なすべてのデータのためにリソースオブジェクトを登録する必要があります。トランザクションオブジェクトは、回復可能なデータのためのリソースをトランザクションの Coordinator に登録します。

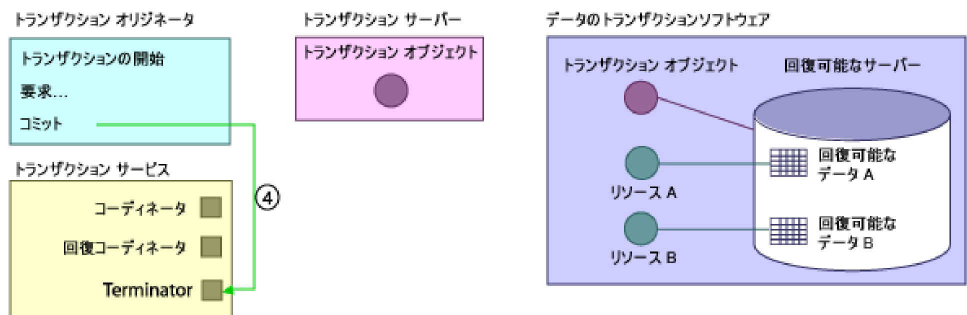
図 10.2　回復可能なデータのためのリソースオブジェクトを登録する



トランザクションオリジネータがトランザクションの完了を開始する

トランザクションオリジネータは、Terminator にトランザクションを完了することを通知します。これにより、VisiTransact Transaction Service で 2 フェーズコミットプロセスが開始されます。この手順は、17 ページの「トランザクションの完了」のステップ 4 にかわる手順です。

図 10.3　トランザクションオリジネータがトランザクションの完了を開始する

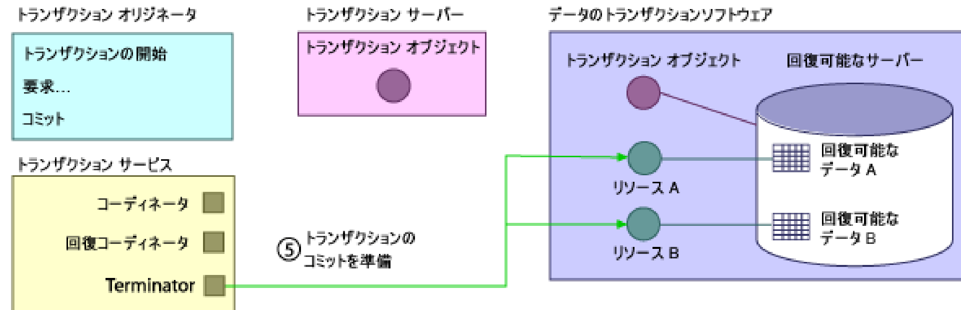


この手順では同じ動作が行われますが、実際には、背後で `commit()` の呼び出しが Terminator によって処理されます。

Terminator がリソースオブジェクトに準備を指示する

Terminator は、トランザクションオリジネータからトランザクションのコミットの要望を受け取ると、トランザクションに参加しているすべてのリソースオブジェクトに連絡して、トランザクションのコミットを準備する必要があることを通知します。そのために、Terminator は、トランザクションに登録されているすべてのリソースオブジェクトの `prepare()` メソッドを呼び出します。

図 10.4 Terminator はリソースオブジェクトにトランザクションのコミットを準備するように要請する



メモ Coordinator に登録されているリソースが 1 つだけの場合、Terminator は、最適化として 1 フェーズコミットを実行します。その場合は、Terminator は、`prepare()` を呼び出してから `commit()` を呼び出すかわりに、リソースの `commit_one_phase()` を呼び出します。

準備フェーズで例外が発生すると、トランザクションはロールバックされます。

リソースオブジェクトが Terminator に提案を戻す

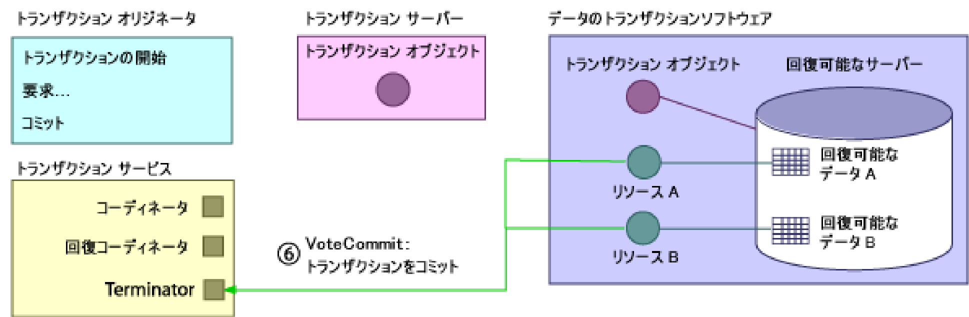
リソースオブジェクトは、準備するように通知されると、Terminator に次の提案で応答します。

- VoteCommit - リソースは、`prepare()` の後で障害が発生した場合でも、要請があればトランザクションをコミットできることを保証します。
- VoteRollback - リソースは、トランザクションのロールバックを要求し、自分自身のデータのロールバックを進めています。
- VoteReadOnly - リソースは、トランザクションに影響される永続的データを持ちません。この 2 フェーズコミットとは無関係であり、この 2 フェーズコミットはリソースの状態に影響しません。

VoteRollback または VoteReadOnly を戻した場合、リソースは、VisiTransact Transaction Service から再度連絡を受けることはなく、自分自身を安全に破棄できます。

この例では、リソース A とリソース B がともに VoteCommit を戻すとします。

図 10.5　リソースは Terminator に提案を戻す



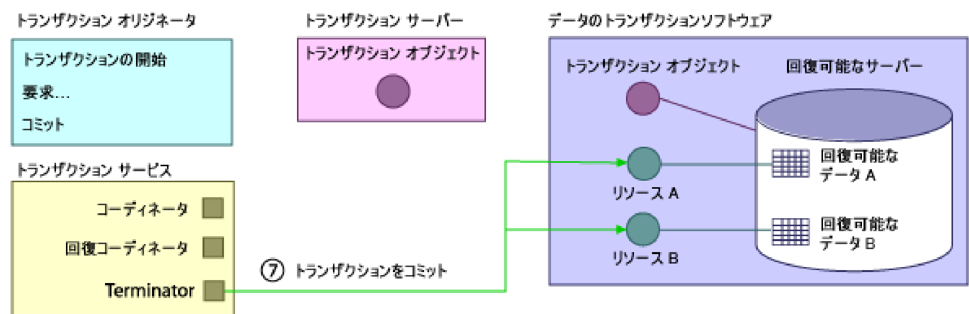
Terminator がコミットするかロールバックするかを決定する

リソースオブジェクトから受け取った提案に基づいて、Terminator は、トランザクションをコミットするかロールバックするかを決定します。この時点で、完了の決定が行われ、ログが記録されます。いずれかのリソースオブジェクトが VoteRollback を戻すか、例外を生成するか、`rollback_only()` を呼び出すと、トランザクションは Terminator によってロールバックされます。

トランザクションの決定がロールバックの場合、Terminator は、すべてのリソース（VoteRollback または VoteReadOnly を戻したリソースを除く）の `rollback()` を呼び出します。決定がコミットの場合、Terminator は、すべてのリソースの `commit()` を呼び出し、2 フェーズコミットプロセスが完了します。

この例では、両方のリソースオブジェクトが VoteCommit を戻したため、Terminator オブジェクトは、リソースオブジェクトにトランザクションをコミットするように要求します。

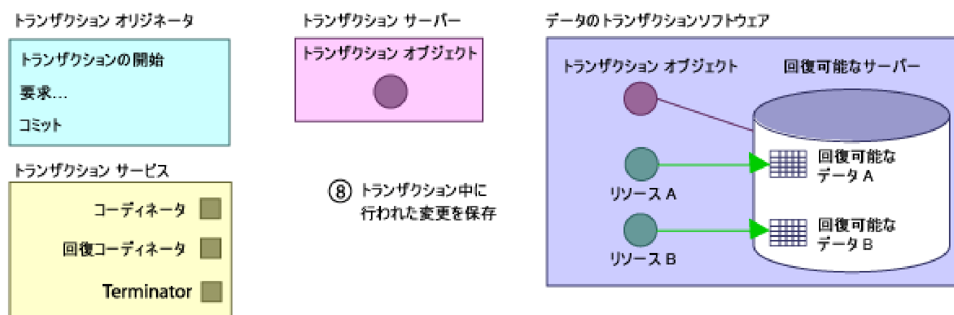
図 10.6　2 フェーズコミットの 2 番めのフェーズが開始される



リソースオブジェクトがトランザクションをコミットする

リソースオブジェクトは、トランザクションをコミットすると、トランザクションによって変更されたすべてのデータをそのデータのすべての読み取り手に公開します。回復可能なオブジェクトによって格納されるデータは、トランザクションの結果にしたがって変更されます。また、リソースオブジェクトは、障害が発生した場合の情報も格納します。トランザクションがコミットされると、最後に、そのトランザクションに関連付けられているすべてのオブジェクト（Coordinator、Terminator、回復コーディネータなど）が削除されます。

図 10.7　リソースオブジェクトはトランザクションで行われた変更をコミットする



2 フェーズコミットのまとめ

以上に示したように、2 フェーズコミットは次の手順で行われます。

1　リソースオブジェクトがトランザクションのために登録される。
2　トランザクションオリジネータがトランザクションの完了を開始する。
3　Terminator がリソースオブジェクトに準備を指示する。
4　リソースオブジェクトが Terminator に提案を戻す。
5　Terminator がコミットするかロールバックするかを決定する。
6　Terminator がリソースオブジェクトにコミットまたはロールバックを指示する。

1 フェーズコミットのまとめ

1 フェーズコミットは次の手順で行われます。

1　リソースオブジェクトがトランザクションに登録される。
2　トランザクションオリジネータがトランザクションの完了を開始する。
3　Terminator がリソースオブジェクトに 1 フェーズのコミットを指示する。
4　リソースオブジェクトが Terminator に提案を戻す。
5　Terminator がコミットするかロールバックするかを決定する。
6　Terminator がリソースオブジェクトにコミットまたはロールバックを指示する。

ロールバックのまとめ

ロールバックは次の手順で行われます。

1　リソースオブジェクトがトランザクションのために登録される。
2　トランザクションオリジネータがトランザクションの完了を開始する。
3　Terminator がリソースオブジェクトにロールバックを指示する。

障害後のトランザクション回復への関与

トランザクションをコミットする決定がログに記録された後で、VisiTransact Transaction Service（またはそのホスト）に障害が発生した場合、Terminator は、VisiTransact Transaction Service と関連のリソースオブジェクトが再び実行中になると、すべてのリソースに対して commit() の呼び出しを開始します。

トランザクションをロールバックする決定がログに記録された後で、VisiTransact Transaction Service（またはそのホスト）に障害が発生した場合、再び実行された VisiTransact Transaction Service は、そのトランザクションがロールバックされるとみなします。これは、トランザクションがロールバック対象としてマークされると、VisiTransact Transaction Service はリソースを管理しなくなり、そのためリソースにロールバックを指示できないからです。そのかわり、リソースは、回復コーディネータ（具体的には、replay_completion() メソッド）を使用して、トランザクションがロールバックされたかどうかを判定する必要があります。

リソースオブジェクトがコミットする前に VisiTransact Transaction Service に障害が発生したが、リソースオブジェクトが準備状態で、VisiTransact Transaction Service がコミットの決定をまだログに記録していない場合は、リソースが回復コーディネータに問い合わせ、トランザクションの完了を開始する責任を持ちます。

障害が発生し、登録されているリソースに Terminator が接続できない場合、Terminator は、リソースに接続できるまで、継続して接続を試みる必要があります。このようにリソースオブジェクトが再起動されるため、トランザクションの原子性が保証されます。また、VisiTransact Transaction Service は、この結果に一致するように回復可能なオブジェクトにトランザクションを完了させることができます。

VisiTransact Transaction Service に障害が発生すると、次の基本規則にしたがってトランザクションが回復されます。

- トランザクションをコミットする決定がすでにログに記録されている場合、Terminator はすべてのリソースの commit() を呼び出し、2 フェーズコミットプロセスは完了します。
- Terminator がヒューリスティック情報だけを保持している場合は、何も行われません。
- 障害が発生する前にトランザクションがロールバック対象になった場合、トランザクションは失われるため、ロールバックされます。
- 登録済みのリソースは存在するが、それに接続できない場合、Terminator は、接続できるまで、継続して接続を試みる必要があります。

第 11 章

ヒューリスティックな決定の管理

ここでは、トランザクション型アプリケーションで管理する必要があるヒューリスティックな決定について説明します。

ヒューリスティックな決定の概要

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、1 人以上のトランザクション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な決定は、通常は通信障害など通常の処理を妨げる異常事態で行われます。ヒューリスティックな決定が行われると、決定が合意結果とは異なり、データの整合性が失われるリスクがあります。

ヒューリスティックな決定によってリソースから返される例外は次のとおりです。

- `HeuristicRollback`. 参加者は、関連するすべての更新をロールバックしました。
- `HeuristicCommit`. 参加者は、関連するすべての更新をコミットしました。
- `HeuristicMixed`. 参加者は、関連する一部の更新をコミットし、それ以外の更新をロールバックしました。
- `HeuristicHazard`. 参加者は、関連する少なくとも 1 つの更新の結果を把握していません。

ヒューリスティックな決定と例外については、75 ページの「トランザクションの完了」を参照してください。

heuristic.log ファイルの概要

VisiBroker VisiTransact は、VisiTransact Transaction Service の各インスタンスに対して 1 つのヒューリスティックログを作成します。ログファイルのデフォルトの場所は **<VBROKER_ADM>/its/transaction_service/<transaction_service_name>/heuristic.log** です。このログファイルはテキスト形式で保存され、表示はできますが編集はできません。ヒューリスティックログには、VisiTransact Transaction Service インスタンスに関連付けられるヒューリスティックによって完了したすべてのトランザクションが記録されます。

ヒューリスティックログレコードには、トランザクションに対してグローバルな情報が記録されます。

- **例外。**必要に応じてトランザクションオリジネータに報告される例外です。CosTransactions::HeuristicHazard Exception のように、ログレコードの Transaction Info 部分の前に記述されます。
- **トランザクション名。**トランザクションの名前（ユーザー定義または VisiTransact Transaction Service が割り当てる）。Update_Inventory_Database のように、ログレコードの Transaction Info 部分の name フィールドに出力されます。
- **トランザクション ID。**トランザクション ID の ASCII バージョンです（`otid`）。ログレコードの Transaction Info 部分の id フィールドに出力されます。
- **トランザクションオリジネータのホスト。**トランザクションオリジネータが配置されているホストマシンの IP アドレスです。ログレコードの Originator Info 部分の host フィールドに出力されます。

ヒューリスティックログには、ログレコードの Participant Info セクションの各リソースに次の情報が記録されます。

- **リソース名。**ヒューリスティックな決定を行った VisiTransact Transaction Service インスタンスに登録された Resource オブジェクトの名前です。name フィールドに出力されます。
- **リソースホスト。**リソースが配置されているホストの IP アドレスです。host フィールドに出力されます。
- **リソース IOR。**リソースのインターオペラブルオブジェクトリファレンスです（IOR）。ior フィールドに出力されます。
- **リソースの提案。**コミットの準備を要求されたときにリソースが送信した提案です。voteForPrepare フィールドに出力されます。
- **リソースの決定。**リソースに行われたヒューリスティックな決定です（OutcomeHeuristicHazard、OutcomeHeuristicMixed など）。outcome フィールドに出力されます。

適切なファイルアクセス許可がある場合は、別の場所にヒューリスティックログファイルを移動してアーカイブできます。その場合、次にヒューリスティックが発生すると VisiTransact Transaction Service インスタンスはヒューリスティックログファイルを再作成します。ログファイルを別の場所にコピーすれば、ログのバックアップコピーを作成できます。これは、ヒューリスティックログを毎日バックアップする場合に便利です。

注意　**heuristic.log** ファイルは編集しないでください。

ヒューリスティックログの解釈

Update_Inventory_Database というトランザクションが開始したとします。このトランザクションには、**inventory** と **customer** という 2 つのリソースが登録されています。トランザクション完了処理の一環として、この 2 つのリソースはトランザクションのコミットを準備するように要請され、両方のリソースは VoteCommit の提案を返します。その後に、VisiTransact Transaction Service はリソースにトランザクションをコミットするように要求します。**customer** リソースのコミットは成功して戻りましたが、**inventory** リソースはヒューリスティックな決定を実行し、HeuristicHazard 例外とともに戻りました。

このトランザクションのヒューリスティックログは、次のようになります。トランザクションオリジネータに戻される例外は、`CosTransactions::HeuristicHazard` です。太字は、90 ページの「heuristic.log ファイルの概要」で説明されている情報の位置を示します。ヒューリスティックログの例には、見やすくするために空白を追加しています。

```
06/02/98, 14:43:43.587, gemini, /net/gemini/vsi2/its/dev/jmitra/vbroker/adm/../bin/
ots,
>None, 0, 0, Error, TransactionService, 4004,
at 0X000001, 0X04110FA4, 896823823, >587
CosTransactions::HeuristicHazard Exception:

Transaction Info:
name = Update_Inventory_Database
Id =
_56495349_01000000_ce400ff2_0000cac1_67656d69_6e695f6f_74730000_0000000
0_000
00000_00000000_00000000_00000000_3574720f_0000e845_00000000_00000000

Originator Info:
host = 206.64.15.75

Participant Info:
name = inventory
host = 206.64.15.75
ior =
IOR:002020200000002549444c3a73797374656d5f746573742f44756d6d7950617274
696369
70656e743a312e300020202000000001000000000000006200010000000000d323036
2e3634
2e31352e373500000f730000004600504d430000000000000002549444c3a7379737465
6d5f74
6573742f44756d6d7950617274696369706e743a312e3000000000000000000e746573
745265
736f757263653100
voteForPrepare = VoteCommit
outcome = OutcomeHeuristicHazard

Participant Info:
name = customer
host = 206.64.15.75
ior =
IOR:002020200000002549444c3a73797374656d5f746573742f44756d6d7950617274
696369
70656e743a312e300020202000000001000000000000006200010000000000d323036
2e3634
2e31352e373500000f730000004600504d430000000000000002549444c3a7379737465
6d5f74
6573742f44756d6d7950617274696369706e743a312e3000000000000000000e746573
745265
736f757263653200
voteForPrepare = VoteCommit
outcome = OutcomeNone
```

問題を特定した後の処理

ヒューリスティックログを確認して問題の本質を特定したら、問題を解決するためにできることがいくつかあります。

最初に、ヒューリスティックログのトランザクション名とトランザクション ID をリソース側のログ（データベースログ）のトランザクション ID と照合します。問題が特定できたら、リソース側で手動で訂正できます。たとえば、91 ページの「ヒューリスティックログの解釈」で説明されているように、リソースログの Update_Inventory_Database を探し、**inventory** リソースに変更内容を手動でコミットします。

第 12 章

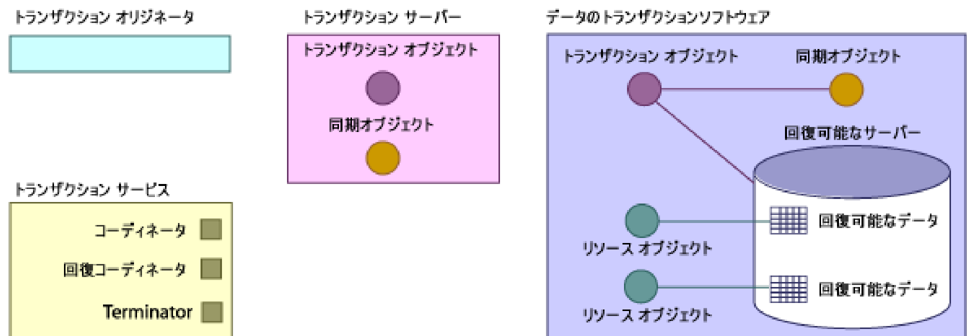
同期オブジェクトの実装

ここでは、同期オブジェクトの実装方法について説明します。

同期オブジェクトについて

同期オブジェクトは、トランザクションの完了の開始前と終了後に、トランザクションの完了をオブジェクトに通知できるようにします。次の図は、標準的な同期オブジェクトがトランザクション型アプリケーションのアーキテクチャ内で位置する場所を示します。

図 12.1 トランザクション型アプリケーションにおける標準的な同期オブジェクトの位置



アプリケーションが commit() を呼び出すと、VisiTransact Transaction Service がトランザクションの完了を開始する前に、`before_completion()` メソッドが呼び出されます。ロールバック要求の場合、`before_completion()` メソッドは呼び出されません。`after_completion()` メソッドは、通常の処理中に常に呼び出されます。

Synchronization オブジェクトを回復することはできません。VisiTransact Transaction Service のインスタンスに障害が発生しても、同期オブジェクトには通知されません。

コミットプロトコルの前に同期オブジェクトを使用する

before_completion() メソッドを使用して、同期オブジェクトは、トランザクションの作業が実行された後でコミットプロトコルが開始される前（prepare() や commit_one_phase の前）に作業を実行できます。たとえば、次の作業を実行できます。

- **パフォーマンスを向上させる。**トランザクションオブジェクトとのやり取り中に変更をキャッシュした後で、同期オブジェクトを使用して、その変更をディスクにフラッシュしたり、リソースを登録することもできます。これの利点は、必要になるまでリソースオブジェクトや開いた状態のデータベース接続を保持しなくてよいことです。
- **その他の作業をトリガーする。**たとえば、監査データベースにレコードを書き込み、そのデータベースをリソースとして同期オブジェクトから登録できます。
- **トランザクションの整合性をチェックする。**必要なすべてのオペレーションが実行されたかどうかを確認できます。たとえば、口座の残高が更新されたかどうかや、残高の変更が履歴テーブルに記録されたかどうかを確認します。

ロールバックやコミットの後で同期オブジェクトを使用する

after_completion() メソッドを使用して、同期オブジェクトは、トランザクションが完了した後（つまり、Terminator がリソースに commit()、rollback()、または commit_one_phase() を指示した後）に作業を実行できます。同期オブジェクトを使用して、次の作業を実行できます。

- **クリーンアップを実行する。**たとえば、メモリオブジェクトを解放できます。
- **トランザクションの完了をほかのプロセスに通知する。**たとえば、同期オブジェクトは、トランザクションの結果をイベントとしてイベントチャネルに送信したり、トランザクションの結果に依存して処理が行われる別のオブジェクトに結果を通知することができます。状態は、StatusCommitted または StatusRolledBack のいずれかです。

同期オブジェクトの登録

次のメソッドのいずれかを使用して、CosTransactions::Coordinator に同期オブジェクトを登録できます。

- CosTransactions::Coordinator::register_synchronization()
- VISTransactions::Current::register_synchronization()

トランザクション型アプリケーションが VisiTransact 管理のトランザクションを使用するか、明示的に伝達されるトランザクションを使用するかに関係なく、VisiTransact Transaction Service は、暗黙的な伝達を使ってトランザクション情報を同期オブジェクトに渡します。

同期オブジェクトが登録されており、トランザクションをコミットする要求が行われると、Terminator は、実際に完了を実行する前に自動的にすべての同期オブジェクトの before_completion() を呼び出します。同期オブジェクト内から before_completion() 呼び出しの間の動作を判定します。登録されているすべての同期オブジェクトが完了すると、Terminator はトランザクションの完了を実行します。ロールバックは、before_completion() メソッドから（VISTransactions::Current または CosTransactions::Coordinator の）rollback_only() を呼び出すことで、実行が保証されます。また、before_completion() メソッドによって例外（CORBA::TRANSACTION_ROLLEDBACK など）が生成された場合も、トランザクションがロールバックされます。

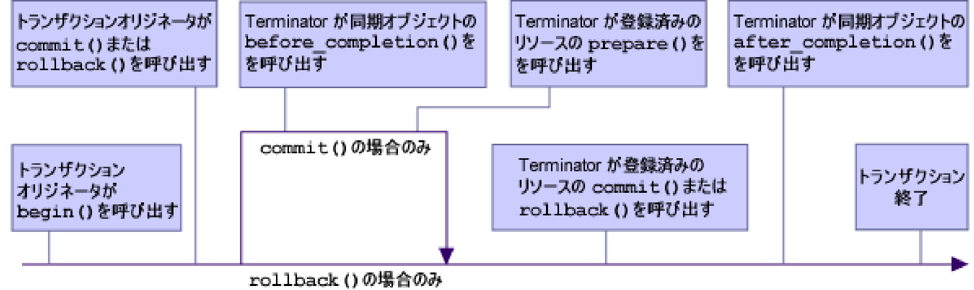
いずれかの同期オブジェクトがトランザクションをロールバック対象としてマークした場合、Terminator は、残りの同期オブジェクトの before_completion() の呼び出しを停止します。任意の同期オブジェクトが rollback_only() を呼び出すことができるため、commit() を呼び出しても、トランザクションのコミットは保証されません。

Terminator が同期オブジェクトと次にやり取りするのは、トランザクションが完了した後です。つまり、リソースオブジェクトから commit()、commit_one_phase()、または rollback() のすべての応答を受け取った後です。このとき、Terminator は、登録されて

いるすべての同期オブジェクトの after_completion() を自動的に呼び出して、トランザクションの結果を状態として渡します。同期オブジェクト内から after_completion() 呼び出しの間の動作を判定します。

次の図は、同期オブジェクトが関係する場合に 2 フェーズコミットプロセスで行われるさまざまな呼び出しを時系列で示します。

図 12.2 同期オブジェクトを含む 2 フェーズコミットの時系列



障害が同期オブジェクトに及ぼす影響

Terminator が before_completion() メソッドを呼び出そうとしたときに、その同期オブジェクトを利用できない場合、トランザクションはロールバックされます。連絡を受けなかった同期オブジェクトは、before_completion() が呼び出されません。VisiTransact Transaction Service が after_completion() を呼び出そうとしたときに利用できない同期オブジェクトがあった場合、それは無視されます。

VisiTransact Transaction Service インスタンスが修復されても、Synchronization オブジェクトは失われたままです。完了は再開されますが、Synchronization オブジェクトは元に戻りません。

トランザクションオブジェクトにおける同期オブジェクトの役割

トランザクションオブジェクトにトランザクションの結果を通知する場合は、トランザクションオブジェクトが Synchronization インターフェースを提供する必要があります。VisiTransact Transaction Service は、after_completion() メソッドを呼び出すときに、同期オブジェクトにトランザクションの完了状態を通知します。

第 13 章

下位互換性と移行

下位互換性

OTS1.1 クライアント対 OTS1.2 サーバー

OTS1.1 クライアントは、OTS1.2 サーバーにあるオブジェクト（IOR に ADAPTS OTS ポリシーの値がある場合）に対して安全にメソッドを呼び出すことができます。

OTS1.2 サーバーから取得したオブジェクトの IOR に REQUIRES OTS ポリシーの値がある場合、これらのオブジェクトへのすべての呼び出しは、アクティブなトランザクションのスコープ内で発生する必要があります。また、OTS1.2 サーバーから取得したオブジェクトの IOR に FORBIDS OTS ポリシーの値がある場合、これらのオブジェクトへのすべての呼び出しは、アクティブなトランザクションのスコープ外で発生する必要があります。

OTS1.1 サーバー対 OTS1.2 クライアント

OTS1.2 クライアントは、OTS1.1 クライアントと同様に OTS1.1 と正常に動作します。ただし、OTS1.1 クライアントとは異なり、OTS1.2 クライアントは無条件にトランザクションコンテキストを伝達することがありません。

コールバックを使用する場合、クライアントがトランザクションコンテキストを OTS1.1 サーバーからのコールバックによって伝達するには、OTS1.1 サーバーに OTS1.2 クライアントが渡すコールバックオブジェクトのタイプを `TransactionalObject` にする必要があります。

移行

ここでは、トランザクションオブジェクトの従来の定義からポリシーを使用する定義への移行について説明します。

VBE 5.1 以前のバージョンで作成されたトランザクションオブジェクトは、CosTransactions::TransactionalObject インターフェースを継承する定義を使用します。トランザクションの動作を制御する機能を利用するには、VisiTransact ポリシーを定義する方法に移行する必要があります。

手順は次のとおりです。

1. すべての IDL ファイルから TransactionalObject インターフェースを削除します。すべてのターゲットオブジェクトのトランザクション要件を制御するための適切な OTSPolicy 値を使用します。
   このリリースでは、非共有トランザクションモデルはサポートされていません。このため、Invocation Policy では、SHARED および EITHER だけが有効です。このポリシーには明示的に値を設定しなくてもかまいません。その場合、VisiTransact は、各ターゲットオブジェクトの Invocation Policy の値を EITHER に設定します。
   ただし、ターゲットオブジェクトの Invocation Policy を設定した場合、VisiTransact は、OTSPolicy 値に照らしてその有効性を確認します。
2. クライアント側で、非トランザクションオブジェクトに対する呼び出しを制御するための適切な `NonTxTargetPolicy` を使用します。
3. `CORBA::ORB::create_policy()` メソッドを使用して、対応するポリシーを作成します。
4. 新しい VisiTransact ライブラリとともにコードをコンパイルします。

第 14 章

セッションマネージャの概要

この章では、データベースを VisiTransact ベースのアプリケーションと統合するための一般的な手順について説明します。また、セッションマネージャと XA リソースディレクタについて詳しく説明します。

メモ　セッションマネージャは Solaris プラットフォームでのみサポートされています。

データベースの VisiTransact アプリケーションへの統合方法

VisiBroker VisiTransact では、DBMS を VisiTransact Transaction Service、アプリケーション、トランザクションオブジェクトなどと容易に統合できます。セッションマネージャとそれに関連付けられているリソースは、DBMS への完全なトランザクションアクセスを提供します。セッションマネージャの XA インプリメンテーションと、そのリソースインプリメンテーション（XA リソースディレクタ）により、完全な 2 フェーズコミット機能がサポートされます。一方、セッションマネージャの DirectConnect バージョンは、統合化リソースを使用して、単一のデータベースへの最適化されたトランザクションアクセスを提供します。ただし、プログラミングモデルの制約は大きくなります。セッションマネージャは、アプリケーションプログラムに埋め込むことにより、いつでもデプロイメントできます。

セッションマネージャは擬似 IDL インターフェースのインプリメンテーションであり、C++ アプリケーションが設定済みのデータベース接続を取得することを可能にします。セッションマネージャは、アプリケーションをデータベース固有の要件（接続処理、スレッド管理、トランザクションの関連付け、リソース登録など）から分離します。セッションマネージャを使って接続が取得されると、VisiTransact トランザクションサービスによってトランザクションが自動的に調整されます。アプリケーション開発者は、トランザクションにデータベースの関与を組み込むコードを記述する必要はありません。アプリケーションコードでは、データベース内の必要なデータにアクセスするための操作を処理するだけで済みます。

メモ　VisiBroker VisiTransact の DBMS 統合ストラテジは、これより大きい統合ストラテジの一部です。VisiTransact は、メインフレームを含む多くのプラットフォームで、一般的なトランザクションプロセスモニタ（Tuxedo、CICS、IMS）やメッセージングソフトウェア（MQSeries）を使用するシステムとも統合できます。

現在、セッションマネージャは、Oracle9i との接続を提供しており、プラグイン可能リソースインターフェースを使用すると、選択したデータベースのセッションを管理できます。Pluggable Resource Interface の詳細については、131 ページの「VisiTransact 向けプラグイン可能データベースリソースモジュール」を参照してください。

セッションマネージャの概要

セッションマネージャは、トランザクション型データベース接続を管理するためのインターフェースを提供する多機能コンポーネントです。

メモ VisiTransact Transaction Service を使用する際に、セッションマネージャの使用は必須ではありません。セッションマネージャと同じ機能を実行するコンポーネントがほかのベンダーから提供されています。VisiTransact Transaction Service は、OMG トランザクションサービス仕様に準拠した同等のコンポーネントであれば、任意のコンポーネントとともに動作します。セッションマネージャの XA インプリメンテーションを使用する場合は、XA リソースディレクタも使用する必要があります。これらは相互依存です。現在、セッションマネージャと XA リソースディレクタは、VisiBroker VisiTransact Transaction Service でのみ動作します。

セッションマネージャは、次の機能を提供します。

- 特定の種類のデータベースへの接続を開く、または接続プールから開いている接続を取得する。
- 接続を現在のトランザクションコンテキストに関連付ける。
- 適切なリソースを Coordinator に登録する。XA インプリメンテーション用に XA リソースディレクタを登録する。セッションマネージャの DirectConnect インプリメンテーション用にローカルの統合化リソースを登録する。
- 再利用できるように接続をプールする。
- 接続スレッドの要件を管理する。

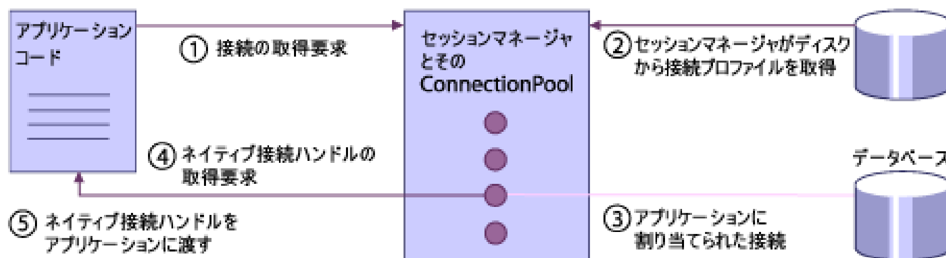
セッションマネージャは、それを使用するアプリケーションにリンクされます。セッションマネージャを使用する場合、リンクされたアプリケーションプロセスを起動する際に、特定のコマンドラインパラメータを使用する必要がある場合があります。コマンドラインパラメータについては、181 ページの「コマンド、ユーティリティ、引数、および環境変数」を参照してください。

データベースへの接続を開く

アプリケーションは、セッションマネージャを使用することにより、特定の種類のデータベースへの VisiTransact 対応の接続を取得できます。この接続は、あらかじめ設定した接続プロファイルを使って作成されます。設定については、以下の節を参照してください。接続が作成されると、VisiTransact Transaction Service は、セッションマネージャとともにトランザクションの調整を行います。

アプリケーションとセッションマネージャは次のように協調して機能します。アプリケーションは、`resolve_initial_references()` を使ってセッションマネージャの ConnectionPool オブジェクトへのリファレンスを取得します。アプリケーションは、ConnectionPool に適切な設定プロファイル名を提供し、ConnectionPool は、そのプロファイルの設定を使ってデータベースへの接続を取得します。ConnectionPool は、次に、このデータベース接続を表す Connection オブジェクトをアプリケーションに戻します。

図 14.1 セッションマネージャは接続ハンドルを取得し、ハンドルをアプリケーションコードに渡す。アプリケーションコードは、この接続ハンドルを介してデータベースと直接対話する。



次に、アプリケーションはデータベース作業を実行するために、Connection オブジェクトにネイティブのデータベースハンドルを要求します。次に、アプリケーションコードは、このネイティブの接続ハンドルを介してデータベースと直接対話します。たとえば、Oracle を使用している場合、アプリケーションコードは OCI を直接呼び出します。

セッションマネージャ API を使ってデータベースへの接続を取得する方法については、119 ページの「セッションマネージャを使用したデータアクセス」を参照してください。

プログラミングは、アプリケーションが使用しているセッションマネージャインプリメンテーションによって、さまざまな制限を受けます。データベースのネイティブ API を使ってトランザクションの処理（SQL 文など）を実行する場合のプログラミング要件と制限については、167 ページの「XA Session Manager for Oracle OCI、version 9i Client」と 175 ページの「DirectConnect Session Manager for Oracle OCI、version 9i Client」を参照してください。

接続プロファイル

接続に必要なすべての情報は、接続プロファイルに保持されます。各プロファイルは一意の名前を持ち、その内容はデータベースログイン ID などの属性です。属性セットは、セッションマネージャインプリメンテーションによって異なります。詳細については、167 ページの「XA Session Manager for Oracle OCI、version 9i Client」と 175 ページの「DirectConnect Session Manager for Oracle OCI、version 9i Client」を参照してください。

接続の設定

VisiBroker コンソールを使って接続プロファイルを作成および設定します。接続プロファイルは、データベースに接続するために必要なすべての属性で構成されます。VisiBroker コンソールについては、141 ページの「VisiBroker コンソールの使い方」を参照してください。

接続とトランザクションの関連付け

セッションマネージャは、データベース接続で実行するデータベース作業にトランザクションを関連付けます。アプリケーションがこの機能を提供する必要はありません。この関連付けは、アプリケーションがセッションマネージャの接続を解放するまで保持されます。

リソースの登録

セッションマネージャは、適切なリソースを Coordinator に自動的に登録します。アプリケーション開発者は、リソース登録のためのコードを追加する必要はありません。DirectConnect インプリメンテーションには、不可視でセッションマネージャに埋め込まれたリソースオブジェクトが含まれていますが、XA インプリメンテーションは XA リソースディレクタという外部プロセスを使用します。セッションマネージャの XA インプリメンテーションを使用するには、XA リソースディレクタを使用できる必要があります。XA リソースディレクタの起動については、107 ページの「セッションマネージャを使用した VisiTransact とデータベースの 統合」を参照してください。

接続の解放

アプリケーションが接続に対する 1 つの作業単位を完了すると、セッションマネージャは Connection オブジェクトの解放を要求します。暗黙的なトランザクションコンテキストの場合は、トランザクションがそのスレッドとの関連付けを解除される前に、接続が解放される必要があります。この関連付けの解除は、次の場合に発生します。

- トランザクションオブジェクトの呼び出しで、呼び出しがクライアントに戻されたとき。
- トランザクションが完了したとき（`commit()` または `rollback()`）。
- トランザクションが中断されたとき。

アプリケーションが接続を解放すると、セッションマネージャは、ほかのトランザクションが使用できるように、データベース接続を解放します。

メモ 接続が解放されると、アプリケーションは、その特定の Connection オブジェクトまたはそれに関連付けられたネイティブの接続ハンドルの使用を続けることはできません。そのトランザクションまたはほかのトランザクションでさらに作業を実行するには、アプリケーションは新しい Connection オブジェクトを取得する必要があります。

接続のプール

セッションマネージャは接続を自動的にプールします。アプリケーションにコードを追加する必要はありません。アプリケーションが接続を解放したとき、セッションマネージャはその接続を自動的に閉じません。接続プールに接続を保持します。接続を求める別の要求があると、プールは接続を再利用しようとします。使用可能な互換性のある接続がない場合にだけ、新しい接続を開きます。

同じトランザクション内では、作業を完了するために必要なだけ何度でも、Connection オブジェクトを取得および解放できます。セッションマネージャの ConnectionPool は、作業単位の完了後に Connection オブジェクトを解放する方が効率がよいため、トランザクション全体の完了を待たずに Connection オブジェクトを解放してください。

メモ セッションマネージャの DirectConnect インプリメンテーションは、1 つの接続を使ってトランザクションの作業を実行するため、トランザクションが完了しない限り ConnectionPool は接続を再利用できません。接続は、トランザクションがコミットまたはロールバックされた後、再利用のためにプールに戻されます。

スレッド要件の管理

セッションマネージャは、データベースが要求するすべての接続スレッド要件を管理します。XA では、特定のスレッドとの接続の維持に関する詳細を完全には指定していないため、DBMS 開発企業は、スレッドの動作に関する XA の要件をさまざまに解釈しています。たとえば、Oracle を使用している場合、XA を使って開いた接続では、その接続が存続する間のすべての呼び出しは、同じスレッドに存在する必要があります。このため、独自のポリシーにしたがってスレッドを管理するほかのソフトウェアと統合することは困難です。

セッションマネージャにより、アプリケーションは現在のスレッドで動作する接続ハンドルを常に取得できます。

すべてのデータベース接続にスレッドの制限があるわけではありません。制限が存在しない場合、セッションマネージャは高い効率で接続をプールできます。詳細については、「セッションマネージャを使用したデータアクセス」を参照してください。

XA プロトコルを使ったグローバルトランザクション

メモ セッションマネージャと XA リソースディレクタの動作対象は、DBMS と RDBMS に限定されません。これらは、XA プロトコルをサポートする任意のリソースマネージャで機能します。リソースマネージャは、通常はデータベースですが、2 フェーズコミットを実行できる XA 準拠の任意のリソースも含まれます。リソースマネージャのもう 1 つの例は、メッセージキューです。

XA は、トランザクションマネージャがグローバル（2 フェーズコミット）トランザクションを調整できるようにするために、X/Open によって規定された業界標準のプロトコルです。ほとんどの RDBMS ベンダーは、外部のトランザクションコーディネータ（VisiTransact Transaction Service、TP モニタ など）がトランザクションの完了を制御する方法として、XA をサポートしています。

セッションマネージャと XA リソースディレクタはどちらも、XA に対応しています。通常、これらは XA の異なる部分を受け持ちます。XA のうち、作業をトランザクションに関連付ける部分は、セッションマネージャが処理します。トランザクションの完了と回復を行う部分は、XA リソースディレクタによって実行されます。

セッションマネージャは、VisiTransact トランザクションサービスとの組み合わせで、XA インターフェース呼び出しを実行して、データベースに対するアプリケーションの作業をトランザクション内に組み込みます。

XA リソースディレクタは、トランザクションの Terminator の指示にしたがってデータベースの 2 フェーズコミットを実行したり、VisiTransact Transaction Service と XA 準拠データベースを仲介することによって回復に関与します。XA リソースディレクタはスタンドアロンプロセスとしてデプロイメントされます。

各データベースに、XA リソースディレクタのインスタンス化（またはプロセス）を 1 つデプロイメントする必要があります。

XA リソースディレクタの概要

メモ XA リソースディレクタは、セッションマネージャの XA インプリメンテーションとともに使用します。XA リソースディレクタは、セッションマネージャの DirectConnect インプリメンテーションでは使用しません。

トランザクションの完了と回復において、XA リソースディレクタは、VisiTransact と X/Open トランザクション環境の間を仲介します。これにより、リソースオブジェクトと XA 準拠のデータベース間の相互運用性が実現します。セッションマネージャは、アプリケーションの作業をトランザクションに関連付けるための仲介を行います。リソースディレクタは、ネットワーク上の特定のデータベースを使用するすべてのトランザクションについて、コミット、ロールバック、および回復時に仲介役として動作する永続的オブジェクトです。

各データベースサーバーごとに XA リソースディレクタが 1 つずつ関連付けられます。セッションマネージャは、トランザクションの完了を調整するリソースディレクタを VisiTransact Transaction Service に通知します。すべての作業が完了すると、VisiTransact Transaction Service はそのリソースディレクタと通信して、リソースディレクタにトランザクションをコミットするかロールバックするかを指示します。

メモ XA リソースディレクタを実装する必要も、VisiTransact Transaction Service に登録する必要もありません。処理は自動的に行われます。ただし、システム管理者は、データベースの実行中は必ず XA リソースディレクタを使用可能にしておく必要があります。

分散トランザクションの回復

XA リソースディレクタは、関連付けられているデータベース内のすべてのトランザクション（VisiTransact Transaction Service によって開始されたトランザクション）をコミットまたはロールバックによって確実に完了します。トランザクションは、VisiTransact Transaction Service、XA リソースディレクタ、またはリソースマネージャに障害が発生した場合でも完了します。障害が発生した時点で完了していなかったトランザクションは、この 3 つのコンポーネントが回復して実行中になると解決されます。

トランザクション完了時に使用される規則または 2 フェーズコミットの詳細については、75 ページの「トランザクションの完了」と 81 ページの「リソースオブジェクトによるトランザクションの完了の調整」を参照してください。

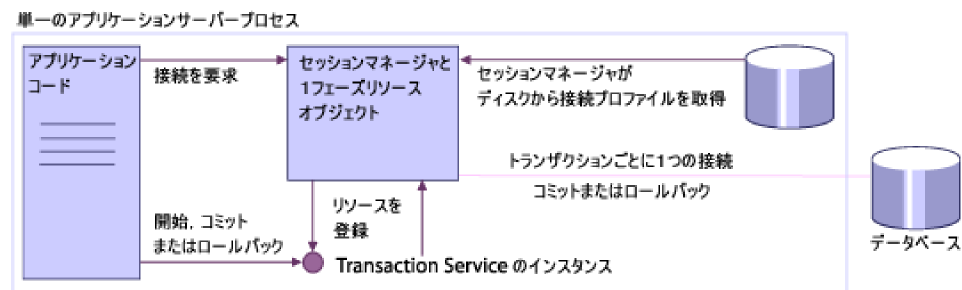
DirectConnect セッションマネージャ

1 つのデータベースと対話する 1 つのアプリケーションサーバーだけがトランザクションに関係している場合、セッションマネージャの XA インプリメンテーションが提供するグローバルな（2 フェーズコミット）トランザクションを使用するかわりに、セッションマネージャの DirectConnect インプリメンテーションを使用できます。これは、リソースが埋め込まれたセッションマネージャを含む単一のプロセスで構成されます。最適なパフォーマンスを得るために、VisiTransact Transaction Service インスタンスはアプリケーションコードにリンクされますが、これは必須ではありません。セッションマネージャの DirectConnect インプリメンテーションを使用するトランザクションは、トランザクションのすべてのコンポーネントが 1 つのプロセスでローカルに置かれるため、*ローカル*トランザクションとみなされます。DirectConnect アクセストランザクションでは、1 つのプロセスが 1 つのデータベースと対話します。特定のトランザクションの作業はすべて、1 つの物理データベース接続上で行われます。同じトランザクションに関するデータベースアクセスでは、接続に必ず同じ接続プロファイルが使用されます。このアーキテクチャの利点は、1 フェーズコミットだけを実行することによるパフォーマンスの向上です。また、2 フェーズコミットをサポートしていないデータベースで実行される処理に対して、トランザクションの考え方を適用できます。

メモ　セッションマネージャの DirectConnect インプリメンテーションを使用する場合、XA リソースディレクタのサービスは必要ありません。セッションマネージャは、内部の透過的なリソースインプリメンテーションを使用します。

単一のアプリケーションサーバープロセスは、特定のトランザクションで使用される可能性があるすべてのメソッドを含むマルチスレッドプロセスです。このプロセスは、1 つのデータベースと対話します。追加のリソース（データベースや、メッセージキューのようなほかの種類のリソース）は許可されません。たとえば、入出金のトランザクションを実行している場合、入金と出金のアプリケーションサーバープロセスは、異なるコンピュータに置くのではなく、1 つのコンピュータ上の 1 つのプロセスに置きます。入金／出金プロセスは、ネットワークを介して 1 つのデータベースと対話できます。ただし、トランザクションは 1 つのデータベースとだけ対話します。そのデータベースとの対話はすべて、その 1 つのプロセスで発生します。

図 14.2　トランザクションのすべてのコンポーネントが 1 つのプロセス内にある DirectConnect アクセストランザクション



セッションマネージャの DirectConnect インプリメンテーションでは、いくつかのパフォーマンス最適化機能を利用できます。トランザクションに関する唯一のプロセスとして、データベースに対して単一のアプリケーションサーバープロセスを使用する場合、トランザクションは（VisiTransact を介して）1 フェーズコミットを実行します。VisiTransact Transaction Service が埋め込まれている場合、セッションマネージャは、すべてのリソース

登録とトランザクションのすべての作業をそのプロセスのローカルで実行します。セッションマネージャ、VisiTransact Transaction Service インスタンス、および入金／出金プロセスは、すべて同じアプリケーションサーバープロセス内にあるため、ネットワークを介したり、プロセスの境界をまたいで互いに対話する必要がありません。また、1 フェーズコミットなので、VisiTransact Transaction Service は、ディスクにログを作成する必要がありません。結果として、パフォーマンスが向上します。

DirectConnect アクセストランザクションを使用することによるその他の利点は、簡単に実行できることです。XA クライアントライブラリや、グローバルトランザクションを有効にするコンポーネントをインストールする必要はありません。たとえば、Oracle で XA アクセストランザクションを使用する場合は、Oracle の分散オプションをインストールする必要がありますが、DirectConnect アクセストランザクションでは、これは不要です。

メモ XA リソースディレクタは、セッションマネージャの DirectConnect インプリメンテーションでは使用しません。使用されるリソースオブジェクトは、セッションマネージャの DirectConnect インプリメンテーションに組み込まれています。

アプリケーションがセッションマネージャを介したデータベースアクセスを要求すると、ConnectionPool オブジェクトは、そのトランザクションにデータベース接続を割り当てます。DirectConnect 接続は、XA 接続とは異なり、トランザクション全体が完了するまで、そのトランザクションに割り当てられたままになります。アプリケーションは、分散トランザクションの場合と同じ手順にしたがって、接続の取得と解放を行います。そのため、VisiTransact Transaction Service は、特定の接続にどのトランザクションとデータベースが関連付けられているかを認識できます。コミット要求を受け取ると、セッションマネージャに埋め込まれたリソースオブジェクトが同じ物理接続を取得し、トランザクションの 1 フェーズコミットまたはロールバックを実行します。

つまり、セッションマネージャが接続を管理するため、アプリケーションは同じトランザクションに対して `getConnection()` を呼び出すたびに、同じ接続を取得します。アプリケーションコードが接続状態を保持しなくても、サーバーが同じトランザクションに対して何度も呼び出しを行うたびに、すべての作業は同じトランザクション内で行われます。

リソースの登録

DirectConnect リソースが登録された後でリソースを登録しようとすると（`register_resource()` 呼び出し）、`CORBA::BAD_PARAM` 例外が生成されます。つまり、VisiTransact Transaction Service は、DirectConnect 接続が使用された後は `register_resource()` 呼び出しを受け付けません。

`getConnection()` が DirectConnect 接続を取得しようとしたときに、Coordinator にリソースがすでに登録されている場合、その要求は失敗し、`VISSessionManager::Error` 例外が生成されます。

デプロイメントに関する問題

VisiTransact Transaction Service は、スタンドアロンのインスタンスまたは埋め込まれたインスタンスのどちらかの設定を選択できます。VisiTransact Transaction Service を単一のアプリケーションサーバーに埋め込んだ場合、DirectConnect アクセストランザクションの処理時にパフォーマンスが向上します。

DirectConnect アクセストランザクションの制約

DirectConnect アクセストランザクションを使ってパフォーマンスを向上させる場合、次に示すようないくつかの制約を受けます。

- トランザクションには 1 つのアプリケーションサーバーだけが関与できます。
- トランザクションには 1 つのリソース（セッションマネージャなど）だけが関与できます。
- 特定のトランザクションの接続は、一度に 1 つのスレッドだけが取得できます。`getConnection()` が呼び出された後は、接続が解放されるまで、ほかのスレッドはそのトランザクションの接続を取得できません。
- 単一接続なので、その接続のプロパティを変更するアプリケーションのすべての動作は（一部のデータベース呼び出しが行う動作など）、その接続が終わるまで維持されます。アプリケーションが後で別のスレッドを使用したり、別の作業単位を実行する場合、接続のプロパティは最初に設定されたままです。接続は再利用されるため、これが後続のトランザクションでの作業に影響を及ぼすことがあります。

DirectConnect と XA アクセストランザクションの共存

セッションマネージャは、同じプロセス内に DirectConnect アクセストランザクションと XA アクセストランザクションが共存できるように設計されています。一部のデータベースインプリメンテーションでは、同じプロセスからの DirectConnect アクセストランザクションと XA アクセストランザクションの混在は許可されません。たとえば、Oracle9i の DirectConnect インプリメンテーションと XA インプリメンテーションは、同じプロセス内では互換性がありません。そのため、この 2 つの実装を混在させることは VisiTransact によって禁止されます。セッションマネージャの特定のインプリメンテーションの詳細については、167 ページの「XA Session Manager for Oracle OCI、version 9i Client」と 175 ページの「DirectConnect Session Manager for Oracle OCI、version 9i Client」を参照してください。

第 15 章

セッションマネージャを使用した VisiTransact とデータベースの統合

この章では、データベースと統合する VisiTransact トランザクション型アプリケーションの管理に必要な知識について説明します。

VisiTransact をデータベースと統合するためには、データベース管理者は次のタスクを実行します。

1. VisiTransact をデータベースと統合することによる影響を評価します。
2. データベースが VisiTransact と統合する準備ができていることを確認します。
3. セッションマネージャ設定サーバーを設定します。
4. 接続プロファイルを設定します。
5. XA リソースディレクタをデプロイメントして設定します。それには、XA リソースディレクタを起動し、必要に応じて VisiBroker オブジェクトアクティベーションデーモン（OAD）に登録します。この手順は、セッションマネージャの XA インプリメンテーションを使用するときにだけ必要です。
6. セッションマネージャを埋め込んだアプリケーションオブジェクトを起動します。

これらのタスクの詳細については、以下の節で説明します。XA と DirectConnect については、別の節でも説明されています。データベース管理者は、前にリストされたタスクのほかにも実行するタスクがあります。追加のタスクを次に示します。

- ヒューリスティックの処理
- パフォーマンス向上のためのチューニング
- 接続プロファイルの永続的ストアファイルの管理

これらの追加タスクは、この章の後ろで詳細に説明しています。

XA を使って VisiTransact をデータベースと統合することによる影響を評価する

管理者にとって最も重要なタスクの 1 つは、特定のサイト環境で分散トランザクション処理を行った場合の影響を評価することです。分散トランザクションの処理には、固有の環境が必要です。使用しているデータベースには分散トランザクションによる処理が適していない場合もあります。評価の際に検討する事項を次に示します。

- XA プロトコルの使用はオーバーヘッドを増やす。
- データベースは 2 フェーズコミットの際に高度な可用性を必要とする。
- データが今までよりも長時間ロックされたり利用できなくなり、処理の並行性が低下する場合がある。
- データベースは高度なトランザクションを実行する。場合によっては、ほかのアプリケーションコンポーネントと協調して動作する必要がある。

これらの項目については、次の節で説明します。

XA の使用によるオーバーヘッドの増加

一般に、XA プロトコルと XA インターフェース呼び出しを使ってデータベースと通信すると、余分なオーバーヘッドが生じます。XA に発生するオーバーヘッドを次に示します。

- トランザクションとの関連付けを行うために発生するデータベースとの間の往復。
- トランザクションに対するデータベースの関与を登録するために発生する VisiTransact Transaction Service との間の往復。
- 処理の準備とコミットを実行するために発生する VisiTransact Transaction Service から XA リソースディレクタへの 1 回または 2 回の往復。

VisiTransact では、関連付けとその解除のための呼び出しは、アプリケーションが使用する際に具体的に要求したデータベース接続にだけ発生します。オーバーヘッドは、使用しないリソースマネージャには発生しません。

高可用性の必要

VisiTransact Transaction Service がデータベースセットに対する 2 フェーズコミットを呼び出す場合、準備フェーズの間にどちらも実行できなければ、トランザクションはロールバックします。そのトランザクションで実行された作業結果は失われます。

ロック中または利用できないデータ

2 フェーズコミットを実行すると、並行処理にボトルネックが発生することがあります。データがロックされてコミットされるまでの間は、そのトランザクションによってロックされたデータの読み取りや変更は、データベースによって防止されます。たとえば、行データを更新すると行データがロックされ、トランザクションをコミットするまでほかのユーザーはそれを変更できません。これは、処理の並行性を低下させます。

メモ データベースがデータをロックする動作はさまざまなので、データベースとアプリケーションによって 1 行のデータだけがロックされることも、複数行のデータがロックされることもあります。

部分的な制御の取得

分散トランザクション処理の利点と欠点を評価するときは、管理作業のスコープが増えることを考慮してください。これは別の利点と欠点をもたらします。準備フェーズが開始すると終了できなくなるため、制御が部分的に失われます。これは、スコープが広くなり、考慮する必要があるコンポーネントが増えるためです。2 フェーズコミットが中断されると、ヒューリスティック出力が発生します。データベースユーティリティを使用すると、強制的にヒューリスティック出力を得ることができます。

VisiTransact Transaction Service がヒューリスティックを処理する方法の詳細については、75 ページの「トランザクションの完了」を参照してください。

DirectConnect を使って VisiTransact をデータベースと統合することによる影響を評価する

DirectConnect トランザクションに関して必要な管理作業は、XA トランザクションの場合より少量です。

DirectConnect を使用する場合の制限を次に示します。

- トランザクションに関与できるリソース（DirectConnect リソース）は 1 つだけです。
- 1 つのデータベースに対するトランザクション作業は、1 プロセスに制限されます。

DirectConnect を使用する利点を次に示します。

- デプロイメントのシナリオの単純さ
- XA 調整のために実行されるデータベースへの RPC の少なさ

データベースの準備

セッションマネージャの機能を使用するためには、まず、分散トランザクションアクセスに必要なソフトウェアのサブセットがデータベースにあることをデータベース管理者に確認します。データベース管理者は、必要に応じて、追加ライブラリをロードしたり、データベース内で SQL スクリプトを実行したり、データベースサーバーの設定パラメータを変更したり、クライアント側ライブラリをインストールすることにより、データベースのインストールを変更します。詳細については、167 ページの「XA Session Manager for Oracle OCI、version 9i Client」と 175 ページの「DirectConnect Session Manager for Oracle OCI、version 9i Client」を参照してください。

一般に、DirectConnect の場合は接続が標準のユーザー接続なので、データベースを準備するための追加手順はありません。

接続プロファイルセット

セッションマネージャがデータベースに接続するためには、その接続を行う方法についての情報が必要です。この情報は、接続プロファイルと呼ばれる属性セットとしてパッケージされています。接続プロファイルは VisiBroker コンソールまたは smconfigsetup ユーティリティを使って作成され、ディスクに保存されるので、永続的に保存でき、アプリケーションサーバーで実行されるセッションマネージャはいつでもプロファイルを取得できます。接続プロファイルがディスク上に存在するため、設定サーバーを常に実行しておく必要はありません。プロファイルセット（接続プロファイルの論理グループ）は、同じ設定サーバーと関連付けられます。各設定サーバーは、一意の名前によって識別されます。ここでは、接続プロファイルの永続的ストアファイルの管理について説明します。

メモ　アプリケーションのセッションマネージャを使ってプロファイル属性に加える変更と、XA リソースディレクタが使用するプロファイル属性とを分離したい場合があります。そのような場合は、セッションマネージャと XA リソースディレクタにそれぞれ別の接続プロファイルを作成します。

プロファイルセットは永続的ストレージファイルに保存されます。永続的ストレージファイルをデフォルトの場所に置くか、別の場所に置いてから引数 -Dvbroker.sm.pstorePath を使ってその場所を指すことができます。

セッションマネージャクライアントが使用する接続プロファイルの変更

セッションマネージャクライアントが使用する接続プロファイルを変更するには、次の手順にしたがいます。

1. VisiBroker コンソールを使って接続プロファイルを変更します。
2. この接続プロファイルを使用するアプリケーションプロセスをシャットダウンします。
3. アプリケーションプロセスを再起動します。

XA リソースディレクタが使用する接続プロファイルの変更

XA リソースディレクタが使用する接続プロファイルを変更するには、次の手順にしたがいます。

1. VisiBroker コンソールを使って接続プロファイルを変更します。
2. 影響を受けるリソースディレクタを使用するアプリケーションプロセスをシャットダウンします。
3. XA リソースディレクタをシャットダウンしてから再起動します。

メモ　アプリケーションプロセスを実行したままにすることもできますが、XA リソースディレクタのシャットダウン時に実行中のトランザクションがあると、ロールバックされることがあります。

XA リソースディレクタの使用

XA リソースディレクタは、トランザクションの実行とリカバリ処理のために、セッションマネージャの XA インプリメンテーションと組み合わせて使用されます。XA リソースディレクタはスタンドアロンプログラムとしてデプロイメントされます。

セッションマネージャの DirectConnect インプリメンテーションを使用している場合、XA リソースディレクタは不要です。

XA リソースディレクタのデプロイメント

VisiBroker VisiTransact からアクセス可能な各データベースサーバーに、XA リソースディレクタのインスタンスをデプロイメントします。XA インプリメンテーションを使用している場合、XA リソースディレクタはデータベースサーバーの実行中は常に実行されている必要があります。これにより、完了プロトコルとリカバリプロトコルでリソースディレクタを使用できます。

同じ `OSAGENT_PORT` の各データベースには、1 つの XA リソースディレクタだけをデプロイメントすることをお勧めします。同じデータベースの同じ `OSAGENT_PORT` に複数の XA リソースディレクタがあるのは非効率的です。通常の操作ではトランザクションを正常にコミット、ロールバックできますが、VisiTransact Transaction Service がダウンして回復するときにリカバリ操作が重複します。その結果、内部リカバリサイクルを完了した VisiTransact Transaction Service に対して、再開要求によるオーバーロードが発生します。

XA リソースディレクタの起動

次のコマンドを使用して、XA リソースディレクタを起動します。

```
prompt>xa_resdir -Dvbroker.sm.profileName=<profile>
  [-Dvbroker.sm.pstorePath=<path>] [-Dvbroker.sm.configName=<name>]
```

次の表は、XA リソースディレクタの起動パラメータを示します。

パラメータ	説明
`-Dvbroker.sm.profileName=<profile>`	データベースとの接続を確立するために使用するセッションマネージャ接続プロファイルの名前。 必須です。
`-Dvbroker.sm.pstorePath=<path>`	永続的ストアファイルが存在するディレクトリのパス。 デフォルトでは、 永続的ストアファイルは `<VBROKER_ADM>/its/session_manager/` にあります。
`-Dvbroker.sm.configName=<name>`	使用するセッションマネージャ設定サーバーの名前。デフォルトでは、セッションマネージャ設定サーバーに割り当てられた名前は **<host>_smcs** です。 ここで **host** は、セッションマネージャ接続プロファイルを作成したサーバーの名前です。これは、プロファイルセット名とも考えられます。

`-Dvbroker.sm.pstorePath` パラメータを指定しなかった場合に使用されるデフォルトについては、113 ページの「永続的ストアファイルのデフォルトパスの確認」を参照してください。セッションマネージャ接続プロファイルの設定方法については、147 ページの「[Session Manager Profile Sets] セクションの使用」を参照してください。

XA リソースディレクタによる接続プロファイルの使用方法

セッションマネージャの接続プロファイルを作成するほかに、XA リソースディレクタの接続プロファイルも作成する必要があります。リソースディレクタは、接続の設定に必要な属性に基づいて、一部のセッションマネージャと同一のプロファイルを使用する場合があります。セッションマネージャが使用しないプロファイルを使用する場合もあります。セッションマネージャはデータベースに問い合わせるために複数のプロファイルを使用することがありますが、データベースの XA リソースディレクタは 1 つのプロファイルだけを使用します。

クライアント側ライブラリのデプロイメント

セッションマネージャと XA リソースディレクタは、クライアント側データベースのライブラリにアクセスできる必要があります。このライブラリには、セッションマネージャと XA リソースディレクタオブジェクトがアクセスするデータベースの XA クライアント側ライブラリも含まれます。

XA リソースディレクタをリモートからシャットダウンする

XA リソースディレクタをリモートからシャットダウンするには、次のコマンドを使用します。

```
prompt> vshutdown -type rd [-name <ITS_XA_Resource_Director_name>]
```

XA リソースディレクタのタイプ（rd）は必須の引数です。

リソースディレクタの名前は、**osfind** コマンドを使用するか接続プロファイルを調べることによって見つけることができます。混乱を回避するために、リソースディレクタには接続プロファイルやデータベースと同じ名前を付けることをお勧めします。**vshutdown** コマンドの詳細については、184 ページの「vshutdown」を参照してください。

XA リソースディレクタを OAD に登録する

オペレータの介入なしで XA リソースディレクタを起動するには、VisiBroker OAD（オブジェクトアクティベーションデーモン）に登録します。XA リソースディレクタのインプリメンテーションは、`oadutil reg` コマンドラインインターフェースを使って登録できます。

XA リソースディレクタを OAD に登録する構文を次に示します。

```
oadutil reg -i visigenic.com/VISSessionManagerSupport/ImplicitResource
-o <resource_director_name> -cpp <installation_dir_path>/bin/xa_resdir -a
-Dvbroker.sm.profileName=<profile> -a -Dvbroker.sm.pstorePath=<path> -a -
Dvbroker.sm.configName=lt;name>
```

次の表は、XA リソースディレクタを OAD に登録するパラメータを示します。

パラメータ	説明
`resource_director_name`	OAD に登録する XA リソースディレクタの名前であり、アクティブ化されるオブジェクト名です。リソースディレクタを起動するために使用されるプロファイルには、プロファイル内のリソースディレクタ名と同じ名前を付けて、データベース名を反映することをお勧めします。複数のデータベースをインストールする場合は、これによってリソースディレクタをプロファイル名に関連付けることが容易になります。

メモ `profile`、`path`、`name` の各パラメータについては 111 ページの「XA リソースディレクタの起動」を参照してください。

XA リソースディレクタを起動するために使用される接続プロファイルには、プロファイル内の XA リソースディレクタ名と同じ名前を付ける必要があります。また、データベース名とも同じである必要があります。複数のデータベースをインストールする場合は、これによって XA リソースディレクタをプロファイル名に関連付けることが容易になります。

セッションマネージャ接続プロファイルの設定方法については、147 ページの「[Session Manager Profile Sets] セクションの使用」を参照してください。

メモ OAD コマンドを使用するには、あらかじめ OAD が実行されている必要があります。OAD の起動については、『VisiBroker for C++ *開発者ガイド*』の「OAD の起動」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。

セッションマネージャベースのアプリケーションプロセスの起動

管理者が明示的にセッションマネージャを起動する必要はありません。セッションマネージャは、プログラムの中で使用されていると、その中で自動的に起動して初期化されます。ORB の初期化では、接続プロファイル属性や、セッションマネージャに関するほかのオプションが含まれているコマンドライン引数が参照されます。

デフォルト以外のパス、またはデフォルト以外のプロファイルセット名を使用する場合は、アプリケーションを起動するときに次の引数を指定して、接続プロファイル属性の永続的ストアが使用されるようにします。

-Dvbroker.sm.pstorePath=<path> -Dvbroker.sm.configName=<name>

パス引数は必須ではありません。パス引数を指定しない場合に、セッションマネージャとセッションマネージャ設定サーバーがデフォルトのパスとプロファイルセット名を確認する方法については、113 ページの「永続的ストアファイルのデフォルトパスの確認」を参照してください。

メモ セッションマネージャのアプリケーションに対するコマンドライン引数については、111 ページの「XA リソースディレクタの起動」を参照してください。

永続的ストアファイルのデフォルトパスの確認

セッションマネージャを使用するときに、`-Dvbroker.sm.pstorePath` 引数は必須ではありません。パス引数を指定しない場合、セッションマネージャとセッションマネージャ設定サーバーは次の順序で設定を確認します。

1. `-Dvbroker.sm.pstorePath` のコマンドライン引数の設定内容。コマンドラインでパスを指定しない場合は、次を確認します。
2. `VBROKER_ADM` 環境変数で設定された内容。これは、インストール時にすべてのデフォルトを受け入れた場合のデフォルトです。VisiTransact は、永続的ストアファイルを `VBROKER_ADM` の下のサブディレクトリ **its/session_manager** に置きます。

ヒューリスティックの適用

データベースユーティリティを使用して、準備フェーズに達した後のトランザクションを監視することができます。場合によっては、トランザクションを解決するために介入する必要があります。たとえば、VisiTransact Transaction Service またはその関連アプリケーションの 1 つの障害が長時間続く場合や、ネットワーク接続に障害がある場合です。データベース管理者が介入して、準備されたトランザクションを VisiTransact Transaction Service を使用しないでコミットするかロールバックする場合、結果の状態はヒューリスティックと呼ばれます。つまり、データベースは VisiTransact Transaction Service とは異なる方法でトランザクションを完了することができます。2 フェーズコミットをサポートするほとんどのデータベースには、ヒューリスティックを適用するためのインターフェースがあります。

VisiTransact Transaction Service がヒューリスティックを処理する方法の詳細については、75 ページの「トランザクションの完了」を参照してください。

パフォーマンスのチューニング

VisiTransact Transaction Service をアプリケーションサーバーに埋め込んだ場合、パフォーマンスの潜在的な向上を現実のものとするために、クライアントはサービスが VisiTransact Transaction Service の正しいインスタンスにバインドされていることを確認する必要があります。

VisiTransact Transaction Service をアプリケーションサーバーに埋め込むことの詳細については、54 ページの「アプリケーションへの VisiTransact Transaction Service インスタンスの埋め込み」を参照してください。

XA のチューニング

分散トランザクションに対して VisiTransact Transaction Service を使用している場合、ネットワークトラフィックを減らすとパフォーマンスが向上します。ネットワークトラフィックを減らすには、VisiTransact Transaction Service のインスタンスと同じノード、またはデータベースと同じノードに一部のコンポーネントを配置します。通信は、セッションマネージャと VisiTransact Transaction Service の間、セッションマネージャとデータベースの間、VisiTransact Transaction Service と XA リソースディレクタの間、および XA リソースディレクタとデータベースの間で発生します。これらのコンポーネントを同じノード上に配置すると、ネットワークトラフィックは減少します。セッションマネージャを使用するトランザクションオブジェクトを VisiTransact Transaction Service またはデータベースと同じノード上に配置することや、XA リソースディレクタをデータベースまたは VisiTransact Transaction Service と同じノードに配置することを検討してください。

セッションマネージャ設定サーバー

セッションマネージャ設定サーバーは、接続プロファイルとサーバーを VisiBroker コンソールのエージェントとして 1 つのセットにしたものです。その目的は、VisiBroker コンソールが接続プロファイルにネットワークアクセスできるようにすることです。

永続的ストアファイルのディレクトリ構造

デフォルトでは、永続的ストアファイルはディスク上の接続プロファイルセットのサブディレクトリに存在します。デフォルトのディレクトリを使用しない場合は、別のディレクトリを指定します。デフォルトのパスは、インストール時に変更するか、コマンドラインフラグ `-Dvbroker.sm.pstorePath` を使って変更するか、またはセッションマネージャを使用するプロセス（XA リソースディレクタなど）によって変更できます。

セッションマネージャとセッションマネージャ設定サーバーが永続的ストアファイルのデフォルトパスを確認する方法については、113 ページの「永続的ストアファイルのデフォルトパスの確認」を参照してください。

注意 **session_manager** ディレクトリには、**install** というディレクトリがあります。**install** ディレクトリの内容を変更したり、手動でファイルを追加しないでください。このディレクトリは、VisiTransact のインストール時に自動的に作成されます。

VisiTransact コンソールを使って接続プロファイルを作成すると、対応するファイルが **session_manager/config** ディレクトリ内のサブディレクトリに作成されます。サブディレクトリファイルの名前はセッションマネージャ設定サーバーの名前に対応しており、プロファイルセット名として使用できます。デフォルトでは、設定サーバーの名前は **<host>_smcs** です。ここで **host** は、設定サーバーが常駐するマシンの名前です。たとえば、マシン名が **athena** の場合、設定サーバーには **athena_smcs** という名前が付けられます。接続プロファイルは設定サーバーのサブディレクトリに保存され、1 つのプロファイルが 1 つのファイルに対応します。接続プロファイルには、**test_oracle_xa** のように意味のある名前を付けることができます。VisiBroker コンソールを使って接続プロファイルに付けた名前は、対応する永続的ストアファイルに自動的に割り当てられます。永続的ストアファイルを手動で作成する必要はありません。VisiBroker コンソールを使用すると、ファイルはセッションマネージャ設定サーバーによって作成されます。VisiTransact は、永続的ストアファイルに次の例のように拡張子 **.cfg** を追加します。

test_oracle_xa.cfg

メモ VisiBroker コンソールに基づいて接続プロファイルの名前を作成するとき、名前の大文字と小文字を区別する規則は、その名前が保存されるファイルシステムで使用される規則と同じです。たとえば、UNIX で大文字と小文字の両方を使って接続プロファイル名を割り当てた場合、後でファイルを検索するときは、そのとおりに指定する必要があります。

永続的ストアファイルはバイナリなので手動で編集できませんが、別の場所にバックアップとしてコピーできます。または、**<configuration_server>** サブディレクトリ全体を別の場所にコピーして、別のプロファイルセット名に変更できます。

接続プロファイル設定を分割して、同じノードに複数の接続プロファイルセットを作成することができます。これは、強い理由がない限りほとんど利点はありません。ノード上に複数のプロファイルセットを置くと、各プロファイルセットに対して 1 つずつ、別のノード上であっても **session_manager/config** ディレクトリにサブディレクトリが作成されます。同じ名前で複数のプロファイルセットを作成すると VisiBroker コンソールで作成したプロファイルセットと識別できなくなるため、避けてください。

メモ セッションマネージャを使用するプロセスは、デフォルトの場所に移動するか、コマンドライン引数を使って指定した任意の場所に移動することにより、1 つの接続プロファイルセットにだけアクセスできます。デフォルトの場所の名前空間か、コマンドライン引数によって指定された場所に制限されます。指定されていない場所にはアクセスできません。たとえば、特定のアプリケーションプロセス内のセッションマネージャのインスタンスは、マーケティングの接続プロファイルセットにだけアクセスできます。給与振込の接続プロファイルセットにはアクセスできません。

永続的ストアファイルのデプロイメント

セッションマネージャを使用するアプリケーションを実行しているすべてのノードは、ディスクから永続的ストアファイルを読み取れる必要があります。したがって、永続的ストアファイルをデプロイメントするときは、いくつかのオプションがあります。

- **オプション 1 :** 接続されたノードグループ内のノードの 1 つに、1 つの設定サーバーが存在します。永続的ストアファイルのディスク上のセットは、共有ファイルシステムを通して共有されます。
- **オプション 2 :** 各ノード上に、一意の名前の設定サーバーと永続的ストアファイルのディスク上のセットが存在します。
- **オプション 3 :** 接続されたノードグループによって、1 つの設定サーバーが共有されます。共有ファイルシステムはありません。永続的ストアファイルのディスク上のセットを変更する場合は、マスターロケーションからほかのノードに永続的ストアファイルのセットを手動でコピーする必要があります。

オプション 1：共有ファイルシステム上の永続的ストアファイル

望ましいデプロイメント方法は、永続的ストアファイルのセットを共有ファイルシステムに配置することです。VisiTransact のインストールでは、セッションマネージャ設定サーバーおよび関連する永続的ストアファイルのディスク上のセットは、1 つのノードにだけデプロイメントされます。セッションマネージャ設定サーバーを実行しているノードに VisiTransact をインストールする場合は、永続的ストアファイルのディレクトリ構造を指定して、この共有ディスクに作成することができます。

インストール後、VisiBroker コンソールを使用して、ネットワーク上で使用されているすべてのセッションマネージャ接続プロファイルを更新できます。すべてのプロファイルが 1 組の接続プロファイルセットとして表示されます。ほかのノード上のセッションマネージャを使用するアプリケーションサーバーは、起動時に永続的ストアを含む共有ディスクを検索するように（`-Dvbroker.sm.pstorePath=<path>` を使用）設定する必要があります。VisiBroker コンソールを使って接続プロファイルを更新すると、変更されたプロファイルはアプリケーションサーバーの起動時に認識されます。

メモ 接続プロファイルはこのオプションと共有されるため、このグループノード内で使用されるセッションマネージャ設定サーバーのインスタンスは 1 つだけであることが重要です。

オプション 2：各ノード上の永続的ストアファイル

セッションマネージャが実行されている各ノードには、独自の設定サーバーと永続的ストアファイルのセットがあります。ネットワーク上の 1 つの VisiBroker コンソールは、それぞれのディスク上の永続的ストアファイルを変更できます。1 つのノードで接続プロファイルを更新する場合は、VisiBroker コンソールを通してほかのノードでも更新して、その変更をほかのノードに反映する必要があります。

このオプションにはディスク共有が不要だという利点がありますが、多数のノードで接続プロファイルを同期させるという複雑さがあります。VisiBroker コンソールもセッションマネージャ設定サーバープロセスのどちらも、異なるディスク上の接続プロファイルのセットを同期できません。

オプション 3：各ノードにコピーされた永続的ストアファイルのセット

ネットワークにディスク上のキャッシュをデプロイメントするために、永続的ストアファイルのマスターセットを作成して手動で各ノードにコピーできます。前のオプションと同様に、このオプションには多数のノードを同期させなければならないという短所があります。ただし、オペレーティングシステムやネットワークのコピーコマンドを使ってプロファイルファイルをコピーすれば、VisiBroker コンソールを使ってそれぞれの永続的ストアファイルを更新するよりも簡単な場合があります。

メモ 一度に設定サーバーのサブディレクトリ内のすべての永続的ストアファイルをコピーすることも、1 つの永続的ストアファイルだけをコピーすることもできます。目的の場所にインストールサブディレクトリが存在しない場合は、サブディレクトリ全体をコピーする必要があります。

手動によるセッションマネージャ設定サーバーの起動

セッションマネージャ設定サーバーが OAD に登録されると、OAD はそれを自動的に起動できます。ただし、手動で設定サーバーを起動するには、次のコマンドを使用します。

prompt>smconfig_server [-Dvbroker.sm.pstorePath=<path>] [-Dvbroker.sm.configName=<name>]

次の表は、セッションマネージャ設定サーバーの起動時のパラメータを示します。

パラメータ	説明
`-Dvbroker.sm.pstorePath`	永続的ストアファイルが存在するディレクトリのパスを指定します。 デフォルトでは、 永続的ストアファイルは `<VBROKER_ADM>/its/session_manager` にあります。
`-Dvbroker.sm.configName`	使用するセッションマネージャ設定サービスの名前を指定します。 デフォルトでは、 セッションマネージャ設定サーバーに割り当てられた名前は **<host>_smcs** です。 ここで **host** は、 セッションマネージャプロファイルを作成したサーバーの名前です。

設定サーバーのシャットダウン

次の理由で、セッションマネージャ設定サーバーをシャットダウンする場合があります。

- メンテナンスを実行する必要がある。
- エラーが発生した場合。
- サーバーが稼動しているマシンを再起動する必要がある。

セッションマネージャ設定サーバーを手動でシャットダウンする場合は、このコマンドを使用します。

prompt>vshutdown -type smcs [-name <smcs_name>]

セッションマネージャ設定サーバー名は、設定サーバーを起動したときにコマンドライン引数 `-Dvbroker.ots.configName` で使用した名前か、デフォルト名 **<host>_smcs** である必要があります。

メモ　シャットダウンしようとしているセッションマネージャ設定サーバーに関連する接続プロファイルをほかのユーザーが VisiBroker コンソールを使って変更、作成していると、セッションマネージャ設定サーバーはその作業を終了してからシャットダウンします。

vshutdown コマンドの詳細については、184 ページの「vshutdown」を参照してください。

セキュリティ

データベースのパスワードは、セキュリティのために VisiBroker コンソールには表示されません。セッションマネージャとリンクするアプリケーションは、データベースパスワードをクリアテキストとして取得できます。これらのアプリケーションでは設定ファイルに対する読み取りアクセスが必要なため、接続プロファイルの永続的ストアファイルへのアクセスを制限することによってデータベースパスワードへのアクセスを制御できます。正しいファイルアクセス許可を提供し、権限がないユーザーにパスワード情報を公開しないアプリケーションを開発するのは、アプリケーション開発者とシステム管理者の役割です。

第 16 章

セッションマネージャを使用したデータアクセス

この章では、分散環境（DirectConnect アクセスと XA アクセスを含む）で、セッションマネージャを使ってトランザクションオブジェクトとデータベース間の接続を管理する方法について説明します。CORBA トランザクションサービス仕様とデータベースの概念を十分に理解していることが前提です。

メモ　セッションマネージャは Solaris プラットフォームでのみサポートされています。

セッションマネージャには、次のインターフェースが含まれます。

- `Connection`：トランザクション対応のデータベース接続を表します。
- `ConnectionPool`：プールから接続を割り当てます。

セッションマネージャと XA リソースディレクタの概要については、99 ページの「セッションマネージャの概要」を参照してください。

統合の準備

セッションマネージャの機能を使用するには、あらかじめ次の操作を実行する必要があります。

- データベースをインストールします。XA アクセストランザクションを処理するかどうかによって、または環境内のほかのコンポーネントによっては、特別な設定が必要な場合があります。詳細については、167 ページの「XA Session Manager for Oracle OCI、version 9i Client」と175 ページの「DirectConnect Session Manager for Oracle OCI、version 9i Client」を参照してください。
- VisiTransact システム管理者がセッションマネージャ用の接続プロファイルを作成してあることを確認します。XA アクセストランザクションを処理する場合、VisiTransact システム管理者は、XA リソースディレクタ用の接続プロファイルも作成する必要があります。
- アプリケーションがセッションマネージャ（ `ConnectionPool` インターフェースと `Connection` インターフェース）を使って接続ハンドルを取得することを確認します。接続を取得するには、名前（ VisiBroker コンソールを使って接続プロファイルに指定した名前）を指定して接続プロファイルを使用します。
- セッションマネージャの XA インプリメンテーションの場合は、VisiBroker VisiTransact からアクセスできる各データベースに対して、XA リソースディレクタのインスタンスがデプロイメントされ実行されていることを VisiTransact システム管理者に確認します。詳細については、Chapter 15, “ セッションマネージャを使用した VisiTransact とデータベースの 統合 .” を参照してください。

- アプリケーションが正しくトランザクションを処理していることを確認します。現在のスレッドにアクティブなトランザクション（暗黙的または明示的なコンテキスト）が必要です。これにより、リソースが確実に VisiBroker VisiTransact のトランザクションに含まれます。VisiTransact 管理のトランザクションについては 57 ページの「VisiTransact 管理のトランザクションの作成および伝達」を、トランザクションのほかの管理方法については 67 ページの「トランザクションを作成および伝達するほかの方法」を参照してください。

セッションマネージャの使い方：手順のまとめ

次の手順は、セッションマネージャの使い方の概要を示します。

1. ConnectionPool オブジェクトのリファレンスを取得します。
2. アクティブなトランザクションがあることを確認します。
3. ConnectionPool から適切な接続プロファイルの Connection オブジェクトを取得します。
4. `getNativeConnectionHandle()` を使用して、Connection オブジェクトからネイティブの接続ハンドルを取得します。
5. ネイティブの接続ハンドルを使ってデータにアクセスします。
6. セッションマネージャの Connection オブジェクトを解放し、ネイティブ接続ハンドルのすべてのコピーを消去します。
7. Connection オブジェクトの割り当てを解除します。

メモ　1 つのトランザクションで多くの作業を実行できます。接続はプールされるため、Connection オブジェクトを保持する時間を短くして、保持し続けないようにする必要があります。1 つのトランザクション内では、必要なだけ何度でも Connection オブジェクトを取得できます。

次の節では、各手順を詳しく説明します。

ConnectionPool オブジェクトリファレンスの取得

次に、ConnectionPool オブジェクトへのリファレンスを取得するための一般的な手順とサンプルコードを示します。

1. ORB `resolve_initial_references()` メソッドを呼び出し、`VISSessionManager::ConnectionPool` オブジェクト型を渡します。
2. 戻されたオブジェクトを `VISSessionManager::ConnectionPool` にナローイングします。次のサンプルコードは、ConnectionPool オブジェクトリファレンスを取得する C++ の例です。

```
{
CORBA::ORB_var orb = CORBA::ORB_init();
CORBA::Object_var initRef =
    orb->resolve_initial_references("VISSessionManager::ConnectionPool");
VISSessionManager::ConnectionPool_var pool =
    VISSessionManager::ConnectionPool::_narrow(initRef);
    ...
}
```

ConnectionPool オブジェクトリファレンスの使用

ConnectionPool オブジェクトリファレンスは、それを作成したプロセス全体で有効です。つまり、任意のスレッドで使用できます。ConnectionPool オブジェクトへのリファレンスを取得するために何度も呼び出しを実行できます。または、プロセス全体でリファレンスを 1 つだけ使用し、重複した `resolve_initial_references()` 呼び出しを削減することもできます。

Connection プールからの Connection オブジェクトの取得

アプリケーションは、ConnectionPool オブジェクトへのリファレンスを取得したら、`getConnection()` 呼び出しを使用して、アプリケーションに対してこのデータベース接続を表す Connection オブジェクトを取得します。この時点で、セッションマネージャがデータベース接続を Connection オブジェクトにバインドします。

`getConnection()` 呼び出しは、アクティブで暗黙的なトランザクションコンテキストを要求します。`getConnectionWithCoordinator()` 呼び出しを使用すると、Coordinator を使用して、トランザクションを明示的に指定できます。`getConnectionWithCoordinator()` の詳細については、121 ページの「明示的なトランザクションコンテキストの使用」を参照してください。

`getConnection()` メソッドは次の処理を行います。

1 データベース接続の取得。

 同じ接続プロファイルを持つ使用されていない接続がプール内にある場合、プールはその接続を戻します。一致する接続プロファイルを持つ使用可能な接続がない場合、セッションマネージャは新しい接続を作成します。接続は、特定のセッションマネージャインプリメンテーションに適したメソッドを使って作成されます。

メモ　接続属性をプログラムでオーバーライドすることはできません。

2 この接続で実行される作業をトランザクションに関連付けます。

3 適切なリソースオブジェクトを VisiTransact Transaction Service に登録します。

次のサンプルコードは、`Connection` オブジェクトを取得して接続を表す方法を示します。

```
...
VISSessionManager::ConnectionPool_var pool;
// プールにデータベース接続を要求します。
VISSessionManager::Connection_var conn = pool->getConnection("quickstart");
...
```

手順のいずれかの段階で発生したすべてのエラーは、`getConnection()` または `getConnectionWithCoordinator()` に対する例外として戻されます。この手順のいずれかが失敗した場合、セッションマネージャは Connection オブジェクトを戻すのではなく、例外を生成します。

明示的なトランザクションコンテキストの使用

明示的なトランザクションコンテキストの接続を取得するには、`getConnectionWithCoordinator()` を使用します。`getConnectionWithCoordinator()` メソッドは、次の目的で使用されます。

- アクティブな暗黙的トランザクションコンテキストがないときに Connection オブジェクトを取得するため
- Connection オブジェクトを取得して、現在アクティブな暗黙的トランザクションコンテキスト以外のトランザクションで使用するため

`getConnectionWithCoordinator()` を呼び出して Coordinator リファレンスを渡すと、セッションマネージャは Coordinator を使用して、通常は暗黙的なコンテキストで実行するすべてのタスクを実行します。セッションマネージャは、暗黙的な VisiTransact 管理トランザクションを使用するかわりに、明示的に記述されたトランザクションコーディネータ

を使用します。接続は、解放されない限り、このトランザクションコーディネータを使って設定されます。

次のサンプルコードは、`getConnectionWithCoordinator()` が Coordinator を介してトランザクションコンテキストを渡す例を示します。

```
...
VISSessionManager::ConnectionPool_var pool;
// 「quickstart」プロファイルを使用したデータベース接続をプールに要求します。
VISSessionManager::Connection_var conn =
    pool->getConnectionWithCoordinator("quickstart", coordinator);
...
```

トランザクションコンテキストの明示的な伝達については、67 ページの「トランザクションを作成および伝達するほかの方法」を参照してください。

接続プールの最適化

セッションマネージャは、接続プールを自動的に保持しており、属性セットに基づいて接続をアプリケーションに戻します。接続プールの効率を向上させるために、アプリケーションは 1 つのデータソースへのすべての接続について、同じ接続プロファイルと属性を使用する必要があります。

ネイティブ接続ハンドルの取得

Connection オブジェクトによって表される接続をアプリケーションが使用するためには、`getNativeConnectionHandle()` メソッドを使って Connection オブジェクトからネイティブの接続ハンドルを取得する必要があります。次に、アプリケーションコードは、このネイティブの接続ハンドルを介してデータベースと直接対話します。このネイティブ接続ハンドルは、その後で通常のデータアクセスを実行するために使用できます。つまり、慣れているデータベース API を使って作業できます。

次のサンプルコードは、OCI インターフェースで使用するために、Oracle データベースへの接続ハンドルを取得する方法を示します。

```
...
VISSessionManager::Connection_var conn;
  // Oracle OCI 接続ハンドルを取得します。
 handles = (SampleOraHandle *) smconn->getNativeConnectionHandle();
...
```

ネイティブ接続ハンドルの使用

セッションマネージャを介して取得した接続ハンドルを使用するには、任意のネイティブ接続ハンドルを使用する場合と同様に、ネイティブのデータベース API を使用します。ただし、セッションマネージャを使用する場合は、一部のアクションを実行できません。どのインプリメンテーションでも、トランザクションの状態に影響を与えるデータベースへの呼び出しは、トランザクションを開始、コミット、ロールバックする呼び出しも含めて、すべて禁止されています。このような呼び出しは、作業におけるトランザクションの整合性に影響します。トランザクションの影響は表に現れない場合があります。たとえば、Oracle の DDL ステートメント（テーブル作成など）は、暗黙的なコミット呼び出しを実行します。禁止されているアクションの詳細については、167 ページの「XA Session Manager for Oracle OCI、version 9i Client」と 175 ページの「DirectConnect Session Manager for Oracle OCI、version 9i Client」を参照してください。

スレッドの要件

セッションマネージャは、データベース接続のスレッド要件があれば、それを管理します。`thread_portable` 引数を指定して `isSupported()` メソッドを使用すると、接続をほかのスレッドで使用できるかどうかを判別できます。通常、`getConnection()` メソッドによって戻される接続は、その接続を取得するのに使用されたスレッドでのみ有効です。アプリケーションは、接続ハンドルをほかのスレッドで使用できません。インプリメンテーションによっては、制限がゆるい場合もあります。特定のデータベースのスレッド要件の詳細については、167 ページの「XA Session Manager for Oracle OCI、version 9i

Client」と 175 ページの「DirectConnect Session Manager for Oracle OCI、version 9i Client」を参照してください。

接続の解放

Connection オブジェクトで `release()` を呼び出すと、トランザクションと特定の接続との関連付けが終了します。Connection オブジェクトを解放しても、基底のデータベース接続を閉じることにはなりません。接続は、再利用のためにプールに戻されます。Connection オブジェクトの解放後は、ネイティブの接続ハンドルや Connection オブジェクトをアプリケーションが再度使用することはできません。このトランザクションでさらに作業を実行する場合は、別の Connection オブジェクトを取得します。

接続を解放するために使用できるメソッドには、`release()` と `releaseAndDisconnect()` の 2 つがあります。使用方法については、以下の表を参照してください。

通常、ネイティブの接続ハンドルはポインタです。したがって、接続を解放する際は、そのポインタのすべてのコピーを null に設定する必要があります。

IDL インターフェース呼び出しを実装している場合は、その呼び出しから戻る前に、接続に対して `release()` または `hold()` を実行する必要があります。実行しないと、セッションマネージャは、クライアントが信頼できないために接続が失われるのを避けることができず、また、一部のデータベースインプリメンテーションで必要なスレッド要件を適用できません。

データベース API を使って接続の解除を試みることは禁止されています。`release()` または `releaseAndDisconnect()` を使用してください。

注意　release メソッドが呼び出された後でアプリケーションが接続ハンドルを使用すると、予期しない結果が生じます。

次の表で、`release()` メソッドと `releaseAndDisconnect()` メソッドの動作について説明します。

メソッド	動作
`release(in ReleaseType type)`	`release()` を呼び出す場合、アプリケーションは 2 つの列挙値のうち 1 つを使用する必要があります。`MarkSuccess` を指定して `release()` を使用すると、接続とトランザクションとの関連付けが正常に解除され、接続はプールに戻ります。 トランザクションをロールバックのみとマークする場合は、`MarkForRollback` を指定して `release()` を使用します。 セッションマネージャは、トランザクションがロールバックされることをデータベースと VisiTransact Transaction Service に通知します。アプリケーションは、トランザクションの整合性が失われる状態を検出すると、 この動作を実行します。
`releaseAndDisconnect()`	このメソッドは、 アプリケーションが接続で何らかの問題を検出したときに使用され、 接続を強制的に閉じ、トランザクションにロールバックのみのマークが付けられたことを VisiTransact Transaction Service に通知します。

次のサンプルコードは、トランザクションを正常に解放するコードの例です。

```
...
VISSessionManager::Connection_var conn;
conn->release(VISSessionManager::Connection::MarkSuccess);
...
```

Connection インスタンスの割り当て解除

release() を呼び出しても、CORBA _release メソッドのように Connection オブジェクトが解放されるわけではありません。このメソッドは、アプリケーションにとって接続が必要なくなったことを ConnectionPool に示します。アプリケーションは、さらに Connection オブジェクトを解放する必要があります。これらは CORBA オブジェクトなので、delete() を呼び出すことはできません。Connection オブジェクトを管理する最も安全な方法は、オブジェクトを Connection_var に保持することです。Connection_var が破棄されると、すべてがクリーンアップされます。ConnectionPool オブジェクトの場合は、オブジェクトを ConnectionPool_var に保持します。

メモ Connection オブジェクトで release() が呼び出されない場合は、デフォルトのデストラクタが接続を解放し、トランザクションを「ロールバックのみ」とマークします。これにより、このメソッドから異常終了した場合に、トランザクションが処理を続行することを確実に回避できます。アプリケーションがロールバックせずにトランザクションの整合性を維持できる場合は、独自の例外処理が接続を明示的に解放し、必要に応じて成功とマークします。

例外の表示

セッションマネージャオブジェクトが例外を生成する場合があります。例外は IDL で定義されています。したがって、例外は標準 CORBA の方式で処理されます。ORB は情報を呼び出し元に転送する役目を果たします。

例外	動作
VISSessionManager::Error	この例外は、VISSessionManager モジュールで定義され、一連の ErrorInfo 構造体を含みます。ErrorInfo は、reason、subsystem、error code からなる構造体です。
VISSessionManager::ConnectionPool::ProfileError	この例外は、ConnectionPool インターフェースで定義され、理由とコードで構成されます。

アプリケーションが受け取る ProfileError 例外には、理由とエラーコードの 2 つのフィールドがあります。これらのフィールドを調べることで、エラーに関する情報を確認できます。

アプリケーションが Error 例外を受け取った場合、ErrorInfos シーケンスで exception.info.length() メソッドを呼び出すことにより、ErrorInfos のシーケンスの長さを確認する必要があります。長さがわかったら、シーケンス内の各 ErrorInfo を調べることができます。

次のサンプルコードは、接続呼び出しで例外を補足するコードの例です。

```
...
  try
  {
    CORBA::ORB_var orb = CORBA::ORB_init();
    CORBA::Object_var object =
      orb->resolve_initial_references("VISSessionManager::ConnectionPool");
    pool = VISSessionManager::ConnectionPool::_narrow(object);

    conn = pool->getConnection("quickstart");

  lda_ptr = (Lda_Def*) conn->getNativeConnectionHandle();
  }
  catch (VISSessionManager::ConnectionPool::ProfileError &ex)
  {
    cerr << "Profile error: " << ex.code << ex.reason << endl;

    conn->releaseAndDisconnect();

    throw ApplicationException(); // このエラーは、アプリケーション要素によって定義さ
れます。
  }
  catch (VISSessionManager::Connection::Error &ex)
  {
    cerr << "Session Manager error: " << endl;

    int len = ex.info.length();
    for (CORBA::ULong i=0; i<len; i++)
    {
      cerr << ex.info[i].subsystem << "-" << ex.info[i].code
        << ": " << ex.info[i].reason << endl;
    }
    conn->releaseAndDisconnect();

    throw ApplicationException(); // このエラーは、アプリケーション要素によって定義さ
れます。
  }
...
```

属性の表示

接続プロファイル属性を表示するには、次の 2 つの方法があります。2 つのメソッドの使用目的は異なります。1 つは現在接続が割り当てられているときに使用され、もう 1 つは割り当てられている接続がないときに使用されます。

メソッド	動作
`VISSessionManager::Connection::getAttributes()`	このメソッドは、現在割り当てられている接続の設定プロファイルの属性の値を戻します。このメソッドは、Connection インターフェースにあります。
`VISSessionManager::ConnectionPool::getProfileAttributes()`	このメソッドを使用すると、接続を割り当てずに、接続プロファイルの属性を照会できます。このメソッドは、使用する接続プロファイルを評価するために使用できます。このメソッドは、ConnectionPool インターフェースにあります。

`getAttribute()` と `getProfileAttributes()` の詳細、およびサポートされるすべてのデータベースに共通の接続属性のリストについては、VisiBroker for C++ API リファレンスの「VISSessionManager モジュール」を参照してください。

セッションマネージャ情報の取得

セッションマネージャのバージョン、`hold()` メソッドがサポートされているかどうか、データベースのスレッドポリシーなどの情報を取得するには、次のメソッドを使用します。

```
string* getInfo(in string info_type)
boolean isSupported(in string support_type)
```

次の種類の情報は、すべてのセッションマネージャに共通です。

- **`"version"`** -- 汎用セッションマネージャのバージョン番号を戻します。バージョン番号は、VisiBroker ユーティリティ `vbver` の標準の 5 フィールド文字列として戻されます。この `info_type` は、対話中のコンポーネントを示す具体的な情報は戻しません。これは、情報提供の目的で使用されます。
- **`"version_rm"`** -- セッションマネージャのリソースマネージャ専用コンポーネントのバージョン番号を戻します。これは、情報提供の目的で使用されます。

次のサポートタイプは、すべてのタイプのセッションマネージャで使用できます。

- **`"hold"`** -- `hold()` メソッドがサポートされている場合は true、そうでない場合は false を戻します。
- **`"thread_portable"`** -- 接続がそれを作成したスレッドだけに制限されている場合は true を、そうでない場合は false を戻します。

次のサンプルコードは、getInfo() の使用例です。

```
...
VISSessionManager::Connection_var conn;
CORBA::String_var info = conn->getInfo("version");
...
```

hold() と resume() の使用

これらのメソッドは、制御用スレッドがクライアントに戻るときに、セッションマネージャの Connection の所有権を保持するために使用されます。

メソッド	動作
`VISSessionManager::Connection::hold()`	このメソッドは、 制御スレッドが現在のプロセスから離れて戻ろうとしていることをセッションマネージャに通知します。
`VISSessionManager::Connection::resume()`	このメソッドは、`hold()` の後で使用され、この Connection の制御スレッドがプロセスに戻ったことをセッションマネージャに示します。

hold() の使い方

セッションマネージャは、この接続に関してアクティブなスレッドが現在のプロセス内にない場合に通知を受けることを要求します。この制約の主な理由は、要求元がエラーなどの理由でこのプロセスに戻ってリソースを解放できない場合に、この接続に使用されているすべてのリソースをセッションマネージャがクリーンアップする必要があるということです。セッションマネージャは、アプリケーションがまだアクティブに接続を使用しているかどうかがわからない限り、トランザクションの関連付けを解除してクリーンアップを続行することができません。

`hold()` は、ほかにも小さな目的のために使用されます。データベース接続の中には、特定のスレッドに使用が制限されているものがあります。次に、そのような場合の例を示します。

- あるクライアントがサーバーに対してインターフェース呼び出しを行いました。
- このクライアントは、セッションマネージャを介して Connection オブジェクトを取得しました。
- その後、このクライアントは、そのトランザクションでさらに作業を実行しようとして、同じサーバーに対して別のインターフェース呼び出しを行いました。サーバープロセスで使用される BOA の動作によっては、この呼び出しは異なるスレッドで発生する可能性があります。

接続が別のスレッドで使用されることを許可しない一部のセッションマネージャインプリメンテーションでは、同じサーバーに対する 2 回めのインターフェース呼び出しはサポートされない場合があります。`hold()` を使用すると、このことをセッションマネージャがアプリケーションに通知できます。

メモ `hold()` の使用は、接続を独占するためパフォーマンスに影響を及ぼします。`hold()` は、必要な場合にだけ使用してください。

`timeout` パラメータは、接続がタイムアウトになり、リソースをクリーンアップするまでセッションマネージャが待機する時間を秒単位で示します。クリーンアッププロセスの一環として、接続は ConnectionPool に戻され、トランザクションにはロールバックのマークが付けられます。

アプリケーションは、`resume()` 呼び出しを入れずに、複数の `hold()` 要求を送信できます。`hold()` が 2 度呼ばれると、呼び出しごとにタイマーが新しい値にリセットされます。たとえば、8:42:30 に `hold(60)` を呼び出すと、8:43:30 にタイムアウトになります。ただし、続けて 8:42:50 に `hold(45)` を呼び出すと、2 度めの `hold()` 呼び出しによってタイマーがリセットされ、8:43:35 にタイムアウトになります。

メモ このメソッドをサポートしていないデータベースセッションマネージャのインプリメンテーションもあります。アプリケーションは、`isSupported()` を使用して、セッションマネージャが `hold()` メソッドをサポートしているかどうかを照会できます。また、詳細については、167 ページの「XA Session Manager for Oracle OCI、version 9i Client」と 175 ページの「DirectConnect Session Manager for Oracle OCI、version 9i Client」を参照してください。

Connection オブジェクト、または対応するデータベース接続ハンドルをもう一度使用するには、事前に Connection オブジェクトの `resume()` を呼び出す必要があります。

resume() の使い方

`resume()` は、`hold()` に関連付けられたタイマーをキャンセルし、セッションマネージャが基底の接続を変更してアクティブなアプリケーションと競合しないようにします。Connection が保留状態にないときに `resume()` を呼び出すと、`VISSessionManager::Error` 例外になりますが、トランザクションや接続の状態は変更されません。

メモ `hold()` の呼び出しと `resume()` の呼び出しの間に、アプリケーションが Connection オブジェクトまたは関連するネイティブデータベースハンドルに基づいてほかの呼び出しを行うことはできません。この間に `hold()` 呼び出しがタイムアウトになった場合は、セッションマネージャが接続を解放し、トランザクションにロールバックのマークを付ける権利を持ちます。これにより、クライアントが停止したり、再度呼び出しを行わなくなっても、そのトランザクションによってアプリケーションサーバーに保持されているリソースが永久に残ることはなくなります。

簡単な統合の例

次のサンプルコードは、セッションマネージャを使って DBMS を統合するサンプルプログラムです。

```
...
void applicationWork(CosTransactions::Coordinator *coordinator)
{
  VISSessionManager::ConnectionPool_var pool;
  //ConnectionPool リファレンスを取得します。
  try
  {
    CORBA::ORB_var orb = CORBA::ORB_init();
    CORBA::Object_var initRef =
      orb->resolve_initial_references("VISSessionManager::ConnectionPool");
    pool = VISSessionManager::ConnectionPool::_narrow(initRef);
  }
  catch (CORBA::Exception &ex)
  {
    cout << "Corba exception obtaining reference to ConnectionPool"
      << endl;
    cout << ex << endl;
    throw ApplicationException();
  }
  // 確実に破棄されるように、スタックで Connection_var を宣言します。
  VISSessionManager::Connection_var conn;
  Lda_Def *lda_ptr = 0;
  try
  {
    // プールにデータベース接続を要求します。
    // データベースプロファイル "quickstart" を使用します。
    conn = pool->getConnection("quickstart");

    //  ネイティブの Oracle OCI 呼び出しに使用する接続ハンドルを取得します。
    lda_ptr = (Lda_Def*) conn->getNativeConnectionHandle();
  }
  catch(const VISSessionManager::ConnectionPool::ProfileError& ex)
  {
    // このプロファイルに関するエラーを受け取りました。
    cerr << "Profile error:¥n";
      << "  " << ex.code
      << ": " << ex.reason
      << endl;
    throw ApplicationException();
    // これは、アプリケーションで定義します。
  }
}
  catch(const VISSessionManager::Error& ex)
  {
    cerr << "Session Manager error:¥n";
    // すべてのエラーメッセージを出力します。
    for(CORBA::ULong i = 0; i < ex.info.length(); i++)
    {
      cerr << "  " << ex.info[i].subsystem
        << "-"  << ex.info[i].code
        << ": " << ex.info[i].reason
        << endl;
    }
    throw ApplicationException();
    // これは、アプリケーションで定義します。
  }

  //lda を使って Oracle データにアクセスします。
  ...
```

```
    // ここに達した場合、未処理の例外は発生していません。
    // 接続を正常に解放します。
    conn->release(VISSessionManager::Connection::MarkSuccess);
}
```

XA インプリメンテーションに関する問題

XA インプリメンテーションは、VisiTransact トランザクションへの全面的な関与をサポートしています。セッションマネージャの XA インプリメンテーションを使用している場合は、DirectConnect インプリメンテーションを使用している場合と動作の一部が異なります。ここでは、XA インプリメンテーションに関する問題について説明します。

トランザクションの完了または回復

セッションマネージャは、`getConnection()` の呼び出し時に、XA リソースディレクタを VisiTransact Transaction Service に自動的に登録します。リソースディレクタは準備を完了し、トランザクションの完了（コミットまたはロールバック）を待機します。トランザクションのすべての作業が完了し、アプリケーションが `commit()` または `rollback()` を呼び出すと、VisiTransact Transaction Service はリソースディレクタを呼び出して、トランザクションをコミットまたはロールバックします。場合によっては、リソースディレクタが回復を調整します。リソースディレクタは、管理者の介入なしで、XA リソース（データベース）と VisiTransact Transaction Service 間のすべての回復を処理します。

トランザクションの完了と 2 フェーズコミットの詳細については、75 ページの「トランザクションの完了」と 81 ページの「リソースオブジェクトによるトランザクションの完了の調整」を参照してください。リソースディレクタの詳細については、「セッションマネージャの概要」を参照してください。

DirectConnect インプリメンテーションに関する問題

セッションマネージャの DirectConnect インプリメンテーションを使用している場合は、XA インプリメンテーションを使用している場合と動作の一部が異なります。

DirectConnect トランザクションでは、接続は次の状態のいずれかになります。

- 使用可能で、関連付けられていない。この接続は、任意のトランザクションで使用できます。
- 使用可能だが、特定のトランザクションに関連付けられている。この接続は、ほかのトランザクションには使用できませんが、同じトランザクションでは取得して使用できます。
- 使用中。この接続は、ほかのスレッドやトランザクションには使用できません。

後の 2 つの状態は、クライアントへのアクセスをシリアライズするための動作を提供します。2 つの異なるスレッドが同時に同じ接続を使用することはできないため、アクセスはシリアライズする必要があります。したがって、同じトランザクションの DirectConnect 接続の 1 つで、2 つの異なるスレッドが同時に作業することはできません。

トランザクションの状態またはリソースの状態は 1 つのプロセスに保持されているため、何らかの要素が失敗した場合、トランザクションはロールバックされます。コミットフェーズで障害が発生した場合、トランザクションがコミットされたかどうかは確認できません。何が発生したかを VisiTransact Transaction Service が認識しているかどうかに基づいて、`commit()` は `CosTransactions::HeuristicHazard` または `CORBA::TRANSACTION_ROLLEDBACK` を受け取ります。

トランザクションの完了または回復

DirectConnect トランザクションの場合、コミットプロセスは 1 フェーズコミットです。セッションマネージャに埋め込まれた 1 フェーズリソースが使用されます。コミットの直前まで作業に使用されていた接続は、プロセスが継続している限り、そのトランザクションの作業をコミットするプロセスでも利用できます。VisiTransact Transaction Service は、コミットの指示を受けると、リソースに 1 フェーズコミットの実行を指示します。コミットが終了すると接続は解放され、別のトランザクションの作業のためにプールに戻されます。

DirectConnect セッションマネージャを保持するアプリケーションサーバーがダウン状態になった場合、1 フェーズリソースは失われ、トランザクションはロールバックされます。

メモ アプリケーションは、接続を解放するために、コミット前に `release()` または `releaseAndDisconnect()` を呼び出しておく必要があります。

DirectConnect から XA への変更

DirectConnect 環境向けに開発したアプリケーションを後から XA 環境で使用する場合、コードを変更する必要はありません。守らなければならない基本的な規則は 1 つだけです。つまり、DirectConnect と XA の両方のプログラミングの制限事項を準拠することです。DirectConnect から XA への変換で必要な変更は、セッションマネージャの XA インプリメンテーション用に設定された接続プロファイルを使用することだけです。また、XA リソースディレクタがデータベースにデプロイメントされていない場合は、それをデプロイメントする必要があります。

第 17 章

VisiTransact 向け プラグイン可能データベース リソースモジュール

プラグイン可能データベースリソースモジュール（または、プラグイン可能リソースインターフェース）は、Borland VisiTransact によって管理されるトランザクション内でトランザクション型アプリケーションがデータベースを永続的ストレージとして使用できるようにする一連の事前定義インターフェースを実装するコンポーネントです。

このマニュアルのサンプルは Oracle9i データベース向けですが、プラグイン可能リソースインターフェースを使用して、選択したデータベースを使用するトランザクションを管理できます。

概念

プラグイン可能データベースリソースモジュールの概要

プラグイン可能データベースリソースモジュールは、Borland VisiTransact によって管理されるトランザクション内でトランザクション型アプリケーションがデータベースを永続的ストレージとして使用できるようにする一連の事前定義インターフェースを実装するコンポーネントです。

開発者は、使用するデータベースの種類に応じてモジュールを実装する必要があります。モジュールが適切に実装およびコンパイルされると、セッションマネージャの接続マネージャがモジュールをアプリケーションにロードします。

プラグイン可能モジュールは共有ライブラリの形式で提供されます。セッションマネージャの接続マネージャは、モジュールをロードしてから、定義済みインターフェースによってモジュールとやり取りします。この定義済みインターフェースにより、セッションマネージャの接続マネージャはトランザクションに必要なタスクを実行します。具体的には、トランザクション型アプリケーションによるデータ処理のために、サポートするデータベースから物理接続を取得すること、データベースに対してトランザクションの関連付け、関連付けの解除、および決定（コミットまたはロールバック）を通知すること、データベースが不要になったときに切断することなどです。

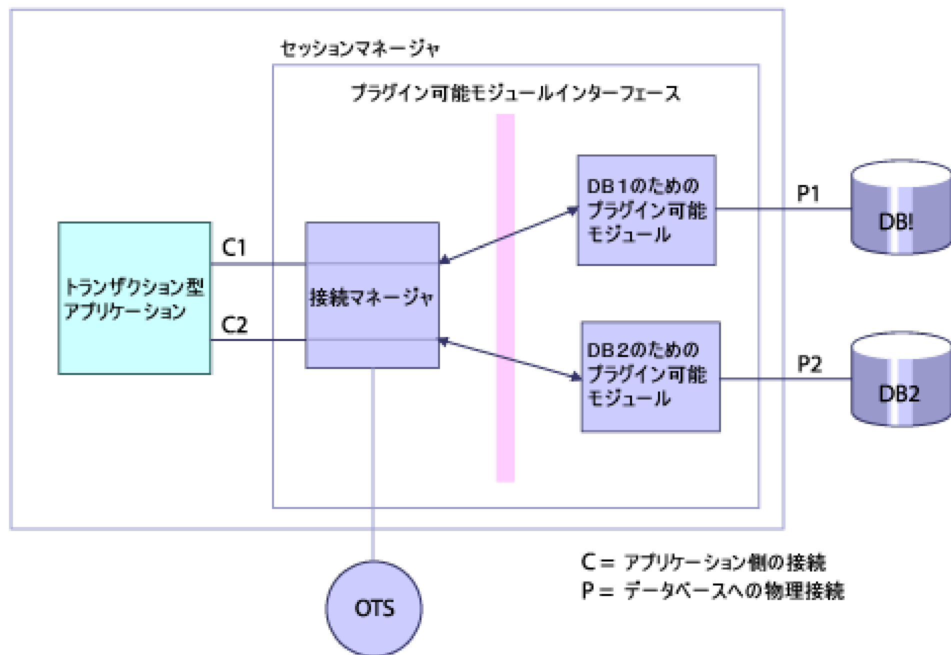
定義済みインターフェースは単純で標準に準拠しており、2 種類のトランザクション、つまりローカルトランザクション（直接接続、DC）とグローバルトランザクション（XA 接続）をサポートします。直接接続は、トランザクションのリソースが 1 つだけの場合に使用されます。その場合は、複数のリソースのコミットを調整する必要がないため、VisiTransact はコミットまたはロールバックを直接データベースに送ります。XA 接続は、1 つのグローバルトランザクションで複数のデータベースを扱う場合に使用されます。VisiTransact は X-Open 仕様で定義された XA インターフェースを使用して、グローバルトランザクションを完了するようにデータベースを調整します。

Borland VisiTransact のプラグイン可能モジュール技術を使用すると、VisiTransact が管理するトランザクション型アプリケーションにさまざまなデータベースを容易に統合できます。

構造の説明

次の図は、プラグイン可能モジュールの基本モデルです。

図 17.1 プラグイン可能データベースのリソースサポートモジュール



トランザクション型アプリケーションは、安全なビジネスタスクを実行するために、通常はトランザクションを初期化します。セッションマネージャの接続マネージャは、トランザクション型アプリケーションとプラグイン可能モジュールの間に存在し、トランザクションにおけるデータベースへのアクセスに関する接続と関連付けを処理します。さまざまなプラグイン可能モジュールが必要に応じてアプリケーションによってロードされ、そのモジュールを通してすべての固有のデータベースに接続できます。

また、セッションマネージャの接続マネージャは、ライブ接続をキャッシュして再利用することで、パフォーマンスを向上させます。さらに、VisiTransact と通信してリソースオブジェクトを登録します。

XA 接続の場合は、独立したコンポーネントであるリソースディレクタが使用されます。

接続管理

トランザクション型アプリケーションが初めてトランザクションを開始し、指定されたプロファイル名の接続を取得すると、セッションマネージャの接続マネージャがプロファイルにしたがって正しいプラグイン可能モジュールをプロセスにロードします。セッションマネージャの接続マネージャは、モジュールを通してデータベースへの接続を作成し、物理リンクを標準の接続オブジェクトにラップしてから、それをトランザクションに関連付けます。その後、接続オブジェクトをアプリケーションに返します。

アプリケーションは接続オブジェクトを取得すると、それを使ってデータベースのトランザクションデータを安全に更新します。一定範囲の作業が終了すると、アプリケーションは接続を解放します。これにより、セッションマネージャの接続マネージャは、接続オブジェクトに割り当てられていたリソースを回収するか、現在の接続をほかのタスクが使用できる状態にすることができます。

接続オブジェクトは、作成されるたびに固有の設定プロファイルに関連付けられます。このプロファイルには、セッションマネージャの接続マネージャがデータベースへの物理接続を作成するために必要な情報が含まれています。また、セッションマネージャの接続マネージャは、トランザクションコンテキスト、内部接続状態、タイムアウトなどの属性を接続オブジェクトに関連付けます。

セッションマネージャの接続マネージャから取得された接続は、関連付けられたトランザクションが完了して解放、切断、または終了するまで有効です。

プラグイン可能接続オブジェクトは、解放されると再利用のためにプールされます。ただし、プールと再利用のメカニズムは DC 接続と XA 接続では異なります。

次の表は、DC 接続における接続の使用方法、プール方法、再利用方法を示します。

インターフェース呼び出し（クライアント）	セッションマネージャの接続マネージャ	プラグインモジュール	データベース
`getConnection()` `getConnectionWithCoordinator()`	プールから使用できる接続を検索し、見つかった場合はトランザクションに関連付けてから返します。 使用できる接続がプール内にない場合は、プラグインモジュールをロードし、モジュールを通して接続を取得してから、新しい接続オブジェクトを作成し、それをクライアントに返します。	この接続オブジェクトがセッションマネージャにロードされ、新しい接続で使用されます。セッションマネージャは、接続を再利用する場合、プラグインモジュールを呼び出しません。	プラグインモジュールから接続要求を受け取ります。クライアントはその接続を使ってテーブルのデータを更新できます。
`release()`	接続をトランザクションから切断し、その接続を接続プールに入れます。	-	-
`releaseAndDisconnect()`	まず、接続とトランザクションの関連付けを解除し、現在のトランザクションをロールバックしてから、プラグインモジュールを通してデータベースとの接続を解除します。	トランザクションをロールバックし、データベースから接続を解除するために呼び出されます。ロールバックの呼び出しは VisiTransact から行われます。	プラグインモジュールからロールバック要求と切断要求を受け取り、接続を解放します。

インターフェース呼び出し（クライアント）	セッションマネージャの接続マネージャ	プラグインモジュール	データベース
`hold()`	接続を保留状態に設定します。これ以降、呼び出し（resume を除く）を行うと例外が生成されます。一定の時間が経過すると、セッションマネージャが接続を復元します。	-	保留状態の間、データベースは要求を受け取りません。
`resume()`	接続が再開され、クライアントは再び接続を使用できます。	-	データベースはクライアントからネイティブな呼び出しを受け取り、テーブルを更新できます。
トランザクションのコミット（VisiTransact から）	-	`commit()` インターフェースが呼び出されます。	データベースは、このトランザクションのすべての変更をコミットします。
トランザクションのロールバック（VisiTransact から）	トランザクションがロールバックされ、接続は接続プールに送られます（再利用のため）。	rollback() インターフェースが呼び出されます。	データベースは、このトランザクションで行われたすべての変更をロールバックします。

次の表では、XA 接続におけるメカニズムについて説明します。

インターフェース呼び出し（クライアント）	セッションマネージャの接続マネージャ	プラグインモジュール	データベース
`getConnection()` `getConnectionWithCoordinator()`	ローカルスレッドプールから、再利用可能な接続を検索します。見つからない場合は、プール内に新しい接続を作成し、その接続（およびスレッド）をトランザクションに関連付けます。	XA 接続を確立するために、xa_switch() インターフェースが呼び出されて xa スイッチ構造へのポインタを取得します。次に xa_open_string() が呼び出されて、データベースのリソースマネージャを開くための文字列を取得します。	xa 接続が開き、現在の呼び出しスレッドに関連付けられます。
`release()`	呼び出しスレッドとトランザクションの関連付けが解除されます。接続オブジェクトがプールで再利用できるようになります。	具体的なインターフェースは呼び出されません。xa スイッチを通して必要な呼び出しが行われます。	開いているリソースが現在のトランザクションから一時的に切断されます。

インターフェース呼び出し（クライアント）	セッションマネージャの接続マネージャ	プラグインモジュール	データベース
`releaseAndDisconnect()`	呼び出しスレッドとトランザクションの関連付けが解除され、接続が閉じます。	`xa_close_string()` インターフェースが呼び出されて、接続を閉じるために使用される文字列が取得されます。xa スイッチを通してほかの必要な呼び出しが行われます。	データベース内のリソースに対する xa 接続が閉じます。
トランザクションのコミット（VisiTransact から）	リソースディレクタがトランザクションのために接続を取得し、xa インターフェースを使って 2 フェーズコミットを完了します。接続は後からプールに回収され、再利用できるように準備されます。	具体的なインターフェースは呼び出されません。xa スイッチを通して必要な呼び出しが行われます。	データベースはリソースディレクタから XA 呼び出しを受け取り、データに加えられた変更をコミットします。
トランザクションのロールバック（VisiTransact から）	リソースディレクタがトランザクションのために接続を取得し、xa インターフェースを使ってトランザクションをロールバックします。接続は後からプールに回収され、再利用できるように準備されます。	具体的なインターフェースは呼び出されません。xa スイッチを通して必要な呼び出しが行われます。	データベースはリソースディレクタから xa 呼び出しを受け取り、データに加えられた変更をロールバックします。

XA 接続と DC 接続の大きな違いはスレッドモデルです。DC 接続の場合、アプリケーションがスレッドから接続を取得すると、そのデータベースで可能であれば、接続オブジェクトをプロセスの任意のスレッドに渡すことができます。XA 接続の場合、セッションマネージャの接続マネージャがさまざまなスレッドの接続を取得し、各接続オブジェクト（および接続が必要なスレッド）を VisiTransact が管理するグローバルトランザクションに関連付けます。XA 接続をスレッド間で渡すと、予期しない結果が生じることがあるため、お勧めしません。

プラグイン可能モジュールの作成

接続プロファイル

プラグイン可能モジュールが提供する各接続は、設定プロファイルに関連付けられます。この設定プロファイルには、セッションマネージャの接続マネージャが接続を取得するために必要な情報が含まれます。次の表は、その情報を示します。

名前	値	意味
プロファイル名	最大 63 文字の文字列（ASCII 文字）	プロファイル名は、この設定情報を保存するファイルの名前です。 また、アプリケーション内で接続の種類を一意に表します。
データベースタイプ	最大 63 文字の文字列（ASCII 文字）	これは、プラグイン可能モジュールがサポートするデータベースを表す情報文字列です。
バージョン情報	最大 63 文字の文字列（ASCII 文字）	データベースのバージョン情報を示す文字列です。
プラグイン可能モジュール名	最大 63 文字の文字列（ASCII 文字）	プラグイン可能モジュールの名前。 セッションマネージャは、必要に応じて、この名前のモジュールをプロセスにロードします。
接続パラメータ	最大 256 文字の文字列（ASCII 文字）	セッションマネージャが新しい接続を取得するためにプラグイン可能モジュールの `getITSDataConnection()` 呼び出しに渡す文字列パラメータ。これにより、モジュールは生成できるさまざまな種類の接続をカスタマイズできます。 詳細については、製品に付属するサンプルを参照してください。

セッションマネージャの接続マネージャが正しいモジュールをロードし、接続を作成するために、アプリケーションは `getConnection()` インターフェース（VISSessionManager.idl で定義）を呼び出すときに設定プロファイル名を提供する必要があります。そのため、アプリケーションを開始する前に、対応する設定プロファイルが作成されている必要があります。

プロファイルを作成するには、製品に付属する smconfigsetup ツールを使用します。

smconfigsetup ツールを起動するには、次の手順にしたがいます。

1. `osagent` を起動します。
2. `smconfig_server` を起動します。
3. `smconfigsetup` を起動します。

smconfigsetup ツールを起動すると、プロファイルを管理するために使用できるオプションリストが表示されます。オプション 7 を選択して、プラグイン可能モジュールの設定プロファイルを作成します。メッセージにしたがって、上の表で定義されているすべての情報を順に設定できます。ツールは、セッションマネージャが使用できるように、指定された場所にプロファイルを保存します。

インターフェースの定義

プラグイン可能モジュールが実装する必要があるインターフェースは、単一のヘッダーファイルで定義されます。

このヘッダーファイルには、セッションマネージャの接続マネージャが接続オブジェクトを取得するために使用する機能と接続クラスが定義されています。プラグイン可能モジュールは、すべてのインターフェースを実装する必要はありません。モジュールがサポートする必要がある接続の種類に基づいて、必須のインターフェースとオプションのインターフェースがあります。

唯一の関数

すべてのプラグイン可能モジュールは、この関数を実装する必要があります。

`GetITSDataConnection()` 関数は、次のように定義されます。

```
extern "C"
ITSDataConnection* GetITSDataConnection(const char* param);
```

この関数は、セッションマネージャの接続マネージャによって呼び出された場合、新しい接続を表す新しいオブジェクトを返す必要があります。既存の接続が再利用される場合、セッションマネージャの接続マネージャは再度この関数を呼び出すことはありません。この関数は C 言語の呼び出し規則を使用します。

また、唯一のパラメータとして文字列を使用します。プロファイルに対応する接続がある場合は、設定ファイル内でこのパラメータを指定することにより、それを制御できます。セッションマネージャの接続マネージャは、プロファイルからこのパラメータを取得し、引数としてこの関数に渡します。

戻り値は `ITSDataConnection` 型のオブジェクトへのポインタです。プラグイン可能モジュールは、このオブジェクトに含まれるインターフェースを実装する必要があります。

この関数は、プラグイン可能サポートモジュールのエントリポイント（モジュールローダー、つまりセッションマネージャの接続マネージャが最初に呼び出す関数）として扱うことができます。セッションマネージャの接続マネージャは、この関数を使ってオブジェクトを取得し、接続を管理します。

ITSDataConnection クラス

このクラスは、次のように定義されます。

```
class ITSDataConnection
{
  public:
  virtual void connect() = 0;
  virtual void disconnect() = 0;
  virtual void rollback() = 0;
  virtual void commit() = 0;
  virtual xa_switch_t* xa_switch() { return 0; }
  virtual const char* xa_open_string() { return 0; }
  virtual const char* xa_close_string() { return 0; }
  virtual void* native_handle() { return 0; }
};
```

ITSDataConnection クラスのメソッドは、次の 3 つのグループに分けられます。

1. ネイティブハンドル取得インターフェース
2. ローカルトランザクション接続／完了インターフェース
3. グローバルトランザクション接続／完了インターフェース

ネイティブハンドル取得インターフェース

```
void* native_handle();
```

この関数は、モジュールによってサポートされる、データベースのネイティブ API にアクセスするために使用されます。戻り値は void ポインタです。インプリメンテーションでは、データベース内のデータの操作に必要な任意の型を戻すことができます。トランザクション型アプリケーションは、getNativeConnectionHandle() を介してこのポインタを取得します。このメソッド内で、セッションマネージャの接続マネージャは、native_handle() を呼び出し、そのポインタをアプリケーションに戻します。

すべてのプラグイン可能モジュールは、この関数を実装する必要があります。

ローカルトランザクション接続／完了インターフェース

ローカルトランザクションをサポートするプラグイン可能モジュールは、これらの関数を実装する必要があります。

次の 4 つのメソッドは、セッションマネージャの接続マネージャによって使用され、ローカルトランザクションの開始と完了をデータベースに通知します。

void connect();

データベースとの接続を確立し、ローカルトランザクションが開始されたことをデータベースに通知します。

void disconnect();

接続が確立されている場合は、それが不要になったことを示します。したがって、接続を閉じることができます。

void rollback();

トランザクションをコミットするように、データベースに通知します。

void commit();

トランザクションをロールバックするように、データベースに通知します。

グローバルトランザクション接続／完了インターフェース

グローバルトランザクションをサポートするプラグイン可能モジュールは、これらの関数を実装する必要があります。

セッションマネージャは、X-Open の XA インターフェースを使用して、XA 準拠データベースとやり取りします。

xa_switch_t* xa_switch();

セッションマネージャの接続マネージャがプラグイン可能モジュールに要求するものは、xa.h で定義されているすべての XA API を含む xa_switch_t データ構造体へのポインタだけです。xa_switch() 関数は、この目的で使用されます。この関数は、呼び出されるたびに、このデータへの有効なポインタを戻す必要があります。

通常、各データベースは xa_switch_t を実装し、それをクライアントに公開します。このデータ構造体の名前は、データベースによって異なります。たとえば、Oracle9i は、xa_switch_t を xaosw という名前のグローバル変数として実装しています。

この関数は、セッションマネージャの接続マネージャによって、接続のタイプを識別するためにも使用されます。この関数が 0 を戻した場合、セッションマネージャは、接続を DC タイプとして処理します。そうでない場合は、XA タイプとして処理します。

グローバルトランザクションをサポートするプラグイン可能モジュールは、この関数を実装し、0 以外を戻す必要があります。

const char* xa_open_string();

この関数は、`xa_open()` 呼び出しの引数として使用される文字列を戻します。

const char* xa_close_string();

この関数は、xa_close() 呼び出しの引数として使用される文字列を戻します。

この 2 つのメソッドは、データベースへの XA 接続を開くまたは閉じるためのデータベース固有のパラメータを取得するために、セッションマネージャによって呼び出されます。xa_open_string() から戻された文字列は xa_open() で使用され、xa_close_string() から戻された文字列は xa_close() で使用されます。

これらの関数が 1 つの XA 接続に対して呼び出されると、セッションマネージャは、それらの戻り値を後で使用できるように保存します。戻されたポインタの有効性をインプリメンテーションが維持し続ける必要はありません。

ビルドと実行

1 必要なヘッダーファイルをインクルードします。

   プラグイン可能モジュールのコンパイルには、VisiTransact 固有のライブラリは必要ありません。ただし、ソースファイルで itsdataconnection.h をインクルードする必要があります。xa.h は、インクルードパスに含める必要がある XA 標準ヘッダーです。通常、XA インターフェースをサポートするデータベースでは、インストールディレクトリに xa.h が用意されています。

2 必要なライブラリが使用可能であることを確認します。

   コンパイル時に VisiTransact 固有のライブラリは必要ありません。ただし、データベース固有のライブラリが必要なことがあります。たとえば、プラグイン可能モジュールで oracle9i データベースをサポートし、OCI を使用する場合は、ライブラリパスに oracle9i クライアントライブラリを含め、コードでそれをリンクする必要があります。

3 プラグイン可能モジュールをビルドします。

   プラグイン可能モジュールは共有ライブラリの形式である必要があります。コンパイラごとに、ビルド対象のタイプを制御するフラグは異なります。共有ライブラリのビルドに必要なフラグについては、サンプルを参照してください。

プラグイン可能モジュールを使用したアプリケーションの実行

トランザクション型アプリケーションの側では、プラグイン可能モジュールに関する情報は不要です。ただし、データベースのデータにアクセスするために、ネイティブハンドルのインターフェースを認識しておかなければならないことがあります。アプリケーションは、プラグイン可能モジュールにリンクされている必要はありません。モジュールは、実行時に必要に応じて、動的にプロセスにロードされます。したがって、アプリケーションを開始する前に、プラグイン可能モジュールがライブラリパスに含まれており、セッションマネージャの接続マネージャを正常にロードできることを確認します。

プログラミングの制限事項

DirectConnect プロファイルを使用する場合は、アプリケーションで「接続操作」、「トランザクション操作」、および「暗黙的な操作」を呼び出さないでください。同様に、XA プロファイルを使用する場合にも別のプログラミングの制限事項があります。詳細については、178 ページの「プログラミングの制限事項」とを参照してください。

既知の制限

プラグインセッションマネージャでは、`VISSessionManager::Connnection::isSupported()` API の戻り値は静的です。DC 接続の場合、`isSupported("hold")` は true を戻し、`isSupported("thread_portable")` は true を戻します。XA 接続の場合、`isSupported("hold")` は false を戻し、`isSupported("thread_portable")` は false を戻します。

現在、プラグインセッションマネージャでは、`VISSessionManager::Connection::getInfo("version_rm")` は NULL を戻します。これは、プラグインセッションマネージャでは情報が適用されないためです。

第 18 章

VisiBroker コンソールの使い方

ここでは、VisiTransact Transaction Service によるトランザクションの管理、ヒューリスティックな完了の追跡、エラーメッセージの表示、セッションマネージャ設定サーバーによるデータアクセスのためのデータベース接続の設定などを含む VisiBroker コンソールの概要について説明します。

VisiBroker コンソールの概要

VisiBroker コンソールを使用すれば、グラフィックによるトランザクション状態の監視、ヒューリスティックログの表示、メッセージログの表示、データベースアクセスの設定が簡単にできます。VisiBroker コンソールの VisiTransact 機能は、次の 3 つのセクションに分割されます。

- 141 ページの「［Transaction Services］セクション」
- 142 ページの「［Session Manager Profile Sets］セクション」
- 146 ページの「メッセージログの表示」

［Transaction Services］セクション

［Transaction Services］セクションでは、VisiTransact Transaction Service のインスタンスおよびそのネットワーク経由のトランザクションを管理できます。また、監視するように選択した VisiTransact Transaction Service インスタンスの元で実行されているトランザクションの状態を監視し、トランザクションの完了を制御できます。

［Transaction Services］セクションを選択すると、特定の OSAGENT_PORT で実行されている VisiTransact Transaction Service のすべてのインスタンスが表示されます。

VisiTransact Transaction Service のインスタンスを選択すると、右側のパネルに 3 つのタブが表示されます。タブ間を移動すれば、次のような操作ができます。

- **トランザクションの表示。**［Transactions］タブでは、この VisiTransact Transaction Service のインスタンスのトランザクションに関する情報が表示されます。トランザクションの詳細を表示し、トランザクションのロールバックまたはコミットを実行し、または VisiTransact Transaction Service のシャットダウンを実行できます。
- **ヒューリスティックの監視。**［Heuristics］タブには、VisiTransact トランザクションのヒューリスティック出力に関する情報が表示されます。ヒューリスティック出力があるトランザクションの各参加者の詳細が表示されます。ヒューリスティックの詳細については「89 ページの「ヒューリスティックな決定の管理」を参照してください。

- ログメッセージの表示。［Message Logs］タブには、VisiTransact Transaction Service のインスタンスと同じノードにあるすべての VisiTransact コンポーネントのメッセージログのエラー、警告、情報メッセージが表示されます。

［Session Manager Profile Sets］セクション

［Session Manager Profile Sets］セクションでは、セッションマネージャのプロファイルを作成および設定できます。セッションマネージャは、アプリケーションと XA リソースディレクタに定義済みのデータベース接続を提供します。

VisiBroker コンソールの起動

VisiBroker コンソールを起動する前に、VisiTransact Transaction Service のインスタンスまたはセッションマネージャ設定サーバーのインスタンスが起動していることを確認します。詳細については、142 ページの「VisiTransact Transaction Service の起動」と 142 ページの「セッションマネージャ設定サーバーの起動」を参照してください。

VisiTransact Transaction Service の起動

VisiTransact Transaction Service は、OAD で起動できます。ただし、次のコマンドを使用すれば、トランザクションサービスを手動で起動できます。

```
prompt>ots
```

OTS コマンドの完全なリストについては、182 ページの「ots」を参照してください。

メモ　管理する VisiTransact Transaction Service が実行されていないか、またはネットワーク上にない場合は、VisiBroker コンソールで管理できる VisiTransact サービスのリストに表示されません。**osfind** ユーティリティを使用すれば、VisiTransact Transaction Service のインスタンスがネットワークで実行されているかどうかを確認できます。

セッションマネージャ設定サーバーの起動

セッションマネージャ設定サーバーは、OAD で起動できます。ただし、次のコマンドを使用すれば、セッションマネージャ設定サーバーを手動で起動できます。

```
prompt>smconfig_server
```

smconfig_server の完全なリストについては、183 ページの「smconfig_server」を参照してください。

VisiBroker コンソールの起動

Windows では、［Borland Deployment Platform］プログラムグループの［Borland Management Console］アイコンをクリックし、コンソールの左側のナビゲーションバーの［VisiBroker］アイコンを選択できます。

または、Windows または UNIX のコマンドプロンプトで、次のコマンドを入力します。

```
vbconsole
```

VisiBroker コンソール画面が表示されます。

［Transaction Services］セクションの使用

［Transaction Services］セクションの機能を使用すれば、選択した VisiTransact Transaction Service のトランザクション情報を監視および管理し、トランザクションのコミットまたはロールバックを実行してトランザクションの状態を解決し、トランザクションサービスのインスタンスをシャットダウンし、ヒューリスティックを監視し、メッセージを表示できます。

［Transaction Services］セクションの使用の詳細については、以下の節を参照してください。

- 143 ページの「トランザクションサービスのインスタンスの検索」
- 143 ページの「トランザクションの監視」
- 144 ページの「トランザクションリストの再表示」
- 144 ページの「特定のトランザクションの詳細の表示」
- 144 ページの「特定のトランザクションの制御」
- 145 ページの「トランザクションリストのフィルタリング」
- 145 ページの「ヒューリスティックによるトランザクションの表示」
- 146 ページの「ヒューリスティックの詳細の表示」
- 146 ページの「メッセージログの表示」
- 146 ページの「メッセージログのフィルタリング」

トランザクションサービスのインスタンスの検索

特定の VisiTransact Transaction Service のトランザクションのリストを表示するには、ネットワークで実行されている VisiTransact Transaction Service インスタンスから選択する必要があります。ほかのインスタンスに切り替える場合は、トランザクションサービスのリストから VisiTransact Transaction Service を選択します。ただし、［Transactions］タブで一度に表示できるのは、VisiTransact Transaction Service の 1 つのインスタンスのトランザクションだけです。

トランザクションの監視

［Transactions］タブには、選択した VisiTransact Transaction Service のトランザクションのリストが表示されます。これにより、完了していないトランザクションを追跡できます。このトランザクションのリストから、現在の状態を表示でき、定期的に再表示して最新の情報を追跡できます。トランザクションを監視するには、次の手順にしたがいます。

1. トランザクションサービスのリストから監視する VisiTransact Transaction Service のインスタンスを選択します。選択したインスタンスのトランザクションの表が表示されます。

   このリストには、各トランザクションに対して次の情報が表示されます。

   - トランザクション名
   - 状況
   - トランザクションオリジネータのホスト
   - 経過時間（秒）
   - トランザクションの完了の試行回数

2. トランザクションのリストを昇順で並べ替えるには、並べ替える列のヘッダーをクリックします。

   画面下部のステータスバーには、トランザクションの数、フィルタリングの状態、現在のシステムアクティビティに関する追加情報が表示されます。

トランザクションリストの再表示

メインウィンドウのツールバーの［Refresh］ボタンをクリックすると、トランザクションのリストを再表示できます。完了したトランザクションは表に表示されません。

特定のトランザクションの詳細の表示

［Transactions］タブを使用すれば、特定のトランザクションの詳細を表示できます。この詳細情報は、トランザクションの状態を解決するために使用できます。

特定のトランザクションを表示するには、すべてのトランザクションの行から選択します。

［Transaction］ビュータブの下部の表には、選択したトランザクションの各参加者に関する次の情報が表示されます。

- 参加者の IOR
- 参加者のホスト
- 提案の準備
- 出力

［PrepareVote］列には、参加者の提案が入ります。指定できる値は、次のとおりです。

- Unknown
- ReadOnly
- Commit
- Rollback

下の 3 つの値は、トランザクションが準備されている場合にだけ表示されます。

［Outcome］列には、参加者のコミットフェーズの結果が表示されます。指定できる値は、次のとおりです。

- None
- Commit
- Rollback
- HeuristicCommit
- HeuristicRollback
- HeuristicMixed
- HeuristicHazard

メモ　ヒューリスティックとヒューリスティックログの詳細については、89 ページの「ヒューリスティックな決定の管理」を参照してください。

特定のトランザクションの制御

［Transactions］タブでは、［Force Rollback］または［Stop Completion］機能を使用して、完了していないトランザクションの状態を解決できます。

- ［**Force Rollback**］は、完了の準備段階を終了していないすべてのトランザクションに使用できます。
- ［**Stop Completion**］は、VisiTransact Transaction Service にトランザクションの完了を停止するように指示します。トランザクションの完了を停止するには、［Stop Completion］をクリックします。

ハングまたは未確定トランザクションの解決

トランザクションが長い間完了しない場合、または結果が未確定な場合は、トランザクションを停止またはロールバックして解決できます。トランザクションを解決するには、次の手順を実行します。

1 問題のトランザクションの詳細を表示するには、144 ページの「特定のトランザクションの詳細の表示」の手順を使用します。

2 コミットのロールバックまたは停止を選択してトランザクションを解決します。

   - トランザクションをロールバックするには、［Force Rollback］をクリックします。

     トランザクションが完了の準備フェーズより進んでいるが存在している場合は、トランザクションをロールバックできないことを示す［Transaction Is In Second-Phase］ダイアログボックスが表示されます。

     トランザクションが存在しない場合、完了したトランザクションはロールバックできないことを示す［Transaction Not Found］ダイアログボックスが表示されます。

   - トランザクションの完了を停止するには、［Stop Completion］をクリックします。

     VisiTransact Transaction Service がこのトランザクションの情報を持たない場合は（完了済みなどの原因が考えられます）、トランザクションを解決できないことを示す［Unknown Transaction］ダイアログボックスが表示されます。

トランザクションリストのフィルタリング

［Filter Transactions］をクリックすると、［Transactions］タブに表示されているトランザクションをフィルタリングできます。

［Filter］ダイアログボックスでは、分と秒でフィルタ値のオプションを設定できます。期間フィルタ値より古いトランザクションだけが表示されます。

フィルタオプションをキャンセルするには、［Transactions］タブの［Filter Off］をクリックします。

ヒューリスティックによるトランザクションの表示

［Heuristics］タブでは、ヒューリスティック出力がありヒューリスティックログに置かれているトランザクションが表示されます。この情報を表示するには、次の手順にしたがいます。

1 ［Heuristics］タブをクリックします。

   VisiTransact は、トランザクションサービスの各インスタンスに対して **heuristic.log** というヒューリスティックログファイルを作成します。デフォルトでは、このディレクトリのパスは <VBROKER_ADM>/its/transaction_service/<transaction_service name>/heuristic.log です。［Heuristics］タブには、リストの各項目に対して次の情報が表示されます。

   - トランザクションの名前
   - オリジネータ
   - ヒューリスティック出力の時間

2 タブへのトランザクションリストのロードをキャンセルするには、［Cancel Refresh］をクリックします。

ヒューリスティックの詳細の表示

［Heuristics］タブを使用すれば、特定のヒューリスティックな完了の詳細を表示できます。この詳細には、トランザクションの解決に役立つ情報が含まれます。

特定のヒューリスティックな決定を表示するには、すべてのヒューリスティックな決定の行から選択します。

［Heuristics］ビュータブの下部の表には、選択したトランザクションに関する次の情報が表示されます。

- 参加者
- ホスト
- PrepareVote
- 出力
- 最終例外（ヒューリスティックが発生する前に発生した最終例外）

メモ　ヘッダーに表示される番号は、ヒューリスティックのリスト内の位置に対応します。リストの最初のヒューリスティックの番号は 1 です。

［PrepareVote］列には、参加者の提案が入ります。可能な値は、Commit または Rollback です。

［Outcome］列には、参加者のコミットフェーズの結果が表示されます。可能な値は、HeuristicCommit、HeuristicRollback、HeuristicHazard、HeuristicMixed、または参加者のヒューリスティックがない None です。

メモ　ヒューリスティックとヒューリスティックログの詳細については、89 ページの「ヒューリスティックな決定の管理」を参照してください。

メッセージログの表示

メッセージログは、VisiTransact Transaction Service を実行する各物理マシンに対して作成されます。メッセージログは `<VBROKER_ADM>/its/message.log` にあります。

選択したトランザクションサービスが置かれているノードのメッセージのリストを表示するには、［Message Log］タブを使用する必要があります。［Message Log］には、次のメッセージが表示されます。

- エラー
- 情報
- 警告

メッセージログのフィルタリング

［Filter Messages］をクリックすると、メッセージログのメッセージをフィルタリングできます。

［Filter］ダイアログボックスが表示されます。

メッセージは、次のようにしてフィルタリングできます。

- 時間ウィンドウを指定する
- 表示するメッセージのタイプを指定する
- ［Source］、［Category］、［Code］、または［Host］フィールドに上記以外の条件を入力する（フィールドに入力される値は、大文字と小文字が区別されます）

フィルタリングを停止するには、［Filter Off］をクリックします。

メッセージログの調整

メッセージログからエントリを完全に削除できます。選択した日付と時刻より古いメッセージは、メッセージログから完全に削除されます。メッセージログを調整するには、次の手順にしたがいます。

1 ［Trim Message Log］をクリックして［Trim］ダイアログボックスを表示します。

　［Trim］ダイアログボックスが表示されます。

2 日付と時刻を設定し、［OK］をクリックします。日付は、月、日、年の形式で読み込まれます。時刻は時、分、秒の形式で読み込まれます。

［Session Manager Profile Sets］セクションの使用

［Session Manager Profile Sets］セクションでは、セッションマネージャ設定サーバーにアクセスできます。これとセッションマネージャを混同しないでください。

セッションマネージャ設定サーバーは、永続的ストレージファイルに対して接続プロファイルを読み書きします。セッションマネージャ設定サーバーにアクセスすると、新しい接続プロファイルの作成と設定、または既存の接続プロファイルの編集ができます。セッションマネージャ設定サーバーの詳細については、107 ページの「セッションマネージャを使用した VisiTransact とデータベースの 統合」を参照してください。VisiBroker コンソールを使って接続プロファイルを設定する方法の詳細については、次の節を参照してください。

- 148 ページの「セッションマネージャ設定サーバーへのアクセスの取得」
- 148 ページの「新しい接続プロファイルの作成と設定」
- 149 ページの「既存の接続プロファイルの編集」

接続プロファイルとは

接続プロファイルは、特定のデータベースに接続するために必要なすべての接続属性で構成されます。詳細については、99 ページの「セッションマネージャの概要」を参照してください。

セッションマネージャ設定プロファイルの作成に必要な属性は、接続するデータベースのタイプによって異なります。たとえば、VisiBroker VisiTransact を Oracle DBMS と統合するには、Session Manager for Oracle OCI を使用する必要があります。その後に、Oracle 専用の属性とすべてのデータベースに共通の属性の組み合わせを使って接続プロファイルを作成できます。

現在、セッションマネージャは Oracle9i データベースとの接続性を提供します。詳細については、167 ページの「XA Session Manager for Oracle OCI、version 9i Client」と 175 ページの「DirectConnect Session Manager for Oracle OCI、version 9i Client」を参照してください。

Pluggable Resource Interface では、選択したデータベースを統合するセッションマネージャを作成できます。詳細については、131 ページの「VisiTransact 向けプラグイン可能データベースリソースモジュール」を参照してください。

セッションマネージャ設定サーバーへのアクセスの取得

142 ページの「セッションマネージャ設定サーバーの起動」の説明にしたがってセッションマネージャ設定サーバーを起動していることを確認します。

セッションマネージャ設定サーバーにアクセスするには、［Session Manager Profile Sets］で［Session Manager Configuration Server］を選択します。

選択したセッションマネージャ設定サーバーで管理されている接続プロファイルのプロファイル名と詳細が右側のパネルの表に表示されます。

新しい接続プロファイルの作成と設定

新しい接続プロファイルを作成する前に、148 ページの「セッションマネージャ設定サーバーへのアクセスの取得」の説明にしたがってセッションマネージャ設定サーバーにアクセスする必要があります。

メモ　smconfigsetup ユーティリティを使って接続プロファイルを作成することもできます。詳細については、186 ページの「smconfigsetup」を参照してください。

新しい接続プロファイルを作成して設定するには、次の手順を実行します。

1 新しいプロファイルを作成するには、右側のパネルの［New］をクリックします。

　［:New Profile］ダイアログボックスが表示されます。

メモ　既存のプロファイルに基づいて新しいプロファイルを作成するには、コピーするプロファイルを選択し、［Duplicate］をクリックします。同じ属性の設定を使用するか、または変更します。ただし、新しいプロファイルに一意の名前を付ける必要があります。

2 ［New Profile Name］フィールドに一意のプロファイル名を入力します。

3 ［DB Type］ドロップダウンリストボックスからデータベースタイプを選択します。

　データベースタイプは、接続プロファイルのデータベースの種類とトランザクションアクセスのタイプを示します。各データベースタイプには、特定の属性が関連付けられています。データベースタイプを入力すると、［Connection Profile Editor］画面に表示される属性が決まります。

4 ［OK］をクリックします。

　［Connection Profile Editor］ダイアログボックスが表示されます。

5 ［Database Name］フィールドに値を入力します。

　このフィールドには、それぞれのデータベースに基づく値を入力する必要があります。

6 ［User Name and Password］フィールドにデータベースのユーザー名とパスワードを入力します。

7 ［Save］をクリックします。

　保存した値は永続的ストレージファイルに書き込まれ、このファイルにアクセスできるすべてのセッションマネージャまたは XA リソースディレクタが値を読み込むことができます。

既存の接続プロファイルの編集

既存の接続プロファイルを編集する前に、セッションマネージャ設定サーバーにアクセスする必要があります。アクセスの方法については、148 ページの「セッションマネージャ設定サーバーへのアクセスの取得」を参照してください。

既存の接続プロファイルを編集するには、次の手順を実行します。

1 既存のプロファイルを編集するには、リストから編集するプロファイルを選択します。

2 ［Open］をクリックします。

  ［Connection Profile Editor］ダイアログボックスが表示されます。

3 ［Connection Profile Editor］画面で変更します。

4 ［Save］をクリックします。

新しい値が適用される時期の詳細については、110 ページの「XA リソースディレクタが使用する接続プロファイルの変更」を参照してください。

接続プロファイルのフィルタリング

初めてセッションマネージャ設定サーバーの接続プロファイルにアクセスするときは、すべてのデータベースタイプに共通するプロファイルが表示されます。

1 プロファイルをデータベースでフィルタリングするには、［DB Type］ドロップダウンリストボックスからデータベースタイプを選択します。

  選択したデータベースタイプのプロファイルだけがタブに表示されます。

  各データベースタイプには、特定の属性が関連付けられています。たとえば、データベースタイプに［Oracle］を選択すると、Oracle データベースに関連付けられたプロファイル（属性を含む）だけが表示されます。Oracle データベース固有の属性がある場合は、共通の属性の列の右に列が追加されます。

2 すべてのデータベースのプロファイルを表示するには、［DB Type］ドロップダウンボックスの［All］を選択します。

接続プロファイルの削除

接続プロファイルを削除するには、接続プロファイルを選択し、［Delete］をクリックします。削除するかどうかを確認するダイアログボックスが表示されます。

接続プロファイルのリストの更新

選択したセッションマネージャ設定サーバーの接続プロファイルのリストを更新するには、［Refresh］をクリックします。

第 19 章

サーバーアプリケーションモデル

ここでは、VisiTransact がサポートするサーバーアプリケーションモデルと XA 設定について説明します。このモデルでは、トランザクションロジックがアプリケーションビジネスロジックから透過的になります。

サーバーアプリケーショントランザクションとデータベース管理

OMG OTS（Object Transaction Service）仕様（バージョン 1.4 以下）では、分散トランザクションアプリケーションの以下の部分が標準化されます。

- CORBA 分散トランザクションアプリケーション管理モデル。これは、OMG IDL の形式で表される OMG バージョンの X/Open DTP モデルです。
- この DTP モデルでの、アプリケーションクライアント、トランザクションコーディネータ、および関連トランザクション対応リソースマネージャの間の、IDL のインターフェースとサービスコンテキストに関連するトランザクション対応相互運用可能プロトコル。このプロトコルは、さまざまな OTS インプリメンテーションと JTS インプリメンテーションによってサポートされています。
- IDL インターフェースとローカルオブジェクト（CosTransactions::Current）によって示される暗黙的および明示的なトランザクションアプリケーションプログラミングモデル。

OTS は、アプリケーションサーバー側のデータベース統合と暗黙的なトランザクションプログラミングモデルを提供しません。OMG OTS 仕様によると、サーバー側では、アプリケーションが XA インターフェースを使用して、明示的にデータベース接続とトランザクション制御を行う必要があります。

ITS サーバーアプリケーションモデルを使用すると、トランザクション制御（およびデータベース接続）は、サーバントビジネスロジックインプリメンテーションによって処理する必要がなくなるかわりに、POA 作成時にポリシーとして指定される属性設定になります。

このセクションの理解に必要な知識

このセクションは、読者に以下の知識があることを前提としています。

- **データベースと埋め込み SQL** データベーステーブルを作成、表示、および操作するためのデータベースツール（Oracle sqlplus など）の使用方法を理解している必要があります。埋め込み SQL でプログラムを作成できること、またデータベース付属の埋め込み SQL ／ C++ 変換プリコンパイラ（Oracle proc など）によってアプリケーションを構築できる必要があります。
- **XML と DTD** XML を使用して XA 設定を記述できること、またデータ型定義（DTD）を理解している必要があります。
- **OMG と X/Open の DTP（Distributed Transaction Process）** DTP アーキテクチャの概念と用語、またクライアント側の暗黙的なトランザクションプログラミングモデル（トランザクション `Current` インターフェースを使用するトランザクションの起動と終了）について理解している必要があります。OMG OTS 仕様全体を読んで理解する必要はありません。
- **XA とデータベース接続の設定** XA 設定とデータベース設定に小さな変更を加えることができる必要があります。

コンテナ管理トランザクション（CMT）の用語、EJB と CCM の概念、EJB または CCM によるインプリメンテーション、デプロイメント、アプリケーションアセンブリの概念を理解すると、このセクションの情報を使用する際に役立ちます。

概念と用語

このセクションでは以下の用語を使用します。

- **クライアント** CORBA アプリケーション。詳細は、「クライアント初期化トランザクション」を参照してください。
- **AP** トランザクションを開始できる CORBA クライアントアプリケーション。
- **サーバー** ビジネスロジックを実装する CORBA サーバーアプリケーション。詳細は、「RM」と「サーバー初期化トランザクション」を参照してください。
- **TM** グローバルトランザクションを調整するトランザクションマネージャ。これは、通常、スタンドアロンのサーバープロセスです（VisiTransact OTS サーバーなど）。VisiTransact（ots.dll/so）ではインプロセス TM もサポートされますが、推奨できません。
- **RM** リソースマネージャ。OMG OTS では、通常、RM はデータベースサーバーを指します。RM は、SQL を使用してデータベースにアクセスするアプリケーションサーバーを指すこともあります。
- **1PC** 1 フェーズコミット。1 つの RM を使用し、プレパレーションステージなしでコミットします。
- **2PC** 2 フェーズコミット。複数の RM を使用し、プレパレーションステージ付きでコミットします。
- **グローバルトランザクション** 複数の RM を必要とする場合があるトランザクション。通常、グローバルトランザクションは TM によって調整され、2PC プロトコルを使用する必要がありますが、例外もあります。詳細は、「ローカルトランザクションの最適化」を参照してください。このセクションで説明するすべてのトランザクションは、デフォルトでは（クライアントまたはサーバーのいずれかによって開始される）グローバルトランザクションです。
- **ローカルトランザクション** 1 つの RM だけを使用し、1 つの制御スレッドに制限された、TM による調整がないトランザクション。
- **クライアント初期化トランザクション（CIT）** クライアント境界トランザクションとも言います。クライアントによって開始および終了されるグローバルトランザクションを指します。クライアント初期化トランザクションは、TM によって調整される必要があります。
- **サーバー初期化トランザクション（SIT）** サーバー境界トランザクションとも言います。クライアント要求を処理するときにサーバー PMT エンジンによって開始および終了されるグローバルトランザクションを指します。このトランザクションの境界は、受け取ったクライアント要求です。このトランザクションは、ビジネスロジックの実行前

に開始され、クライアントに応答する前に終了します。サーバー初期化トランザクションはグローバルですが、リモート TM によって調整されるとは限りません。

メモ　SIT は、十分に文書化され、EJB と CCM（CORBA コンポーネントモデル）で幅広く使用される一般的なトランザクションモデルです。

- **ローカルトランザクションの最適化（LTO）** グローバルトランザクションが 1 つのローカル RM（データベース）だけにアクセスする場合、そのグローバルトランザクションを TM に依存せずにサーバーがローカルに開始および終了できるようにする技術。サーバーは、SIT だけを TM にエキスポートし、2 つめの RM（プロセス外の別のトランザクションオブジェクト）に一方向呼び出しを行う場合は、これを真のグローバルトランザクションに変更します。J2EE の資料に LTO のシナリオの例がありますが、OTS インプリメンテーションには適用できません。したがって、VisiTransact の LTO は、OMG に準拠し、他の OTS インプリメンテーションと相互運用および適用できるように、別の技術を使用します。
- **PMT** POA 管理のトランザクション。サーバー側のトランザクションおよびデータベース統合エンジンです。PMT は、データベース接続とトランザクションのすべての詳細をアプリケーションビジネスロジックから切り離し、開発者から隠蔽します。PMT により、サーバントインプリメンテーションビジネスロジックでは、データベース接続とトランザクションロジックをインプリメンテーションに記述する必要がなくなります。データベース接続とトランザクション制御は、ビジネスロジックに依存せず、アプリケーションアセンブリの間に設定と再設定を行うことができます。PMT により、特定のビジネスロジックインプリメンテーションに CIT と SIT の両方を入れることができます。また、特定のオブジェクトの複数オペレーションシグニチャをそれぞれ異なるトランザクション属性で設定できます。
- **CosTransactions::Current** 暗黙的なトランザクションモデルで、スレッド固有のグローバルトランザクションを開始および処理するためにクライアントアプリケーションで使用される単一のオブジェクト。サーバーインプリメンテーションも、このオブジェクトを使用して、現在関連しているグローバルトランザクションに関するスレッド固有の情報を取得できます。CosTransactions::Current は、`resolve_initial_references()` メソッドを使用して ORB から取得できます。
- **PMT::Current:** PMT エンジンによって調整されるスレッド固有のデータベース接続とトランザクションに関する情報を取得するためにサーバーアプリケーションで使用される単一のオブジェクト。接続名などの取得情報は SQL AT 節で必要です。他の取得情報は、PMT 診断に役立ちます。PMT::Current オブジェクトインスタンスは、PMT::Current::instance() を使用して取得できます。
- **XA:** X/Open によって標準化された API。XA API ドライバ（通常は共有ライブラリ）は、データベースベンダーから提供されます。PMT は、これらのドライバを使用して、データベース接続とトランザクションを処理します。PMT により、XA（およびデータベース接続とトランザクション）は、アプリケーションの開発者に透過的になります。XA は、アプリケーションアセンブリの際に正しく設定する必要があります。詳細は、160 ページの「XA リソースの設定」を参照してください。
- **セッションマネージャ（SM）** 以前のリリースで使用されたサービス。このサービスを今回のリリースで使用するには、`vbroker.its.its6xmode` プロパティを `true` に設定する必要があります。詳細は、164 ページの「VisiTransact のプロパティ」を参照してください。
- **リソースディレクタ（RD）** 以前のリリースで使用されたサービス。このサービスを今回のリリースで使用するには、`vbroker.its.its6xmode` プロパティを `true` に設定する必要があります。詳細は、164 ページの「VisiTransact のプロパティ」を参照してください。

グローバルトランザクションおよび PMT のシナリオ

クライアント開始グローバル 2PC および 1PC トランザクション

OTS と X/Open DTP モデルにおいて分散トランザクションは、クライアントが開始するグローバルトランザクションを意味します。この場合、クライアントは、トランザクション

が 2PC（図 1）の場合も 1PC（図 2）の場合も、TM にコンタクトしてトランザクションを開始および終了（コミットまたはロールバック）します。

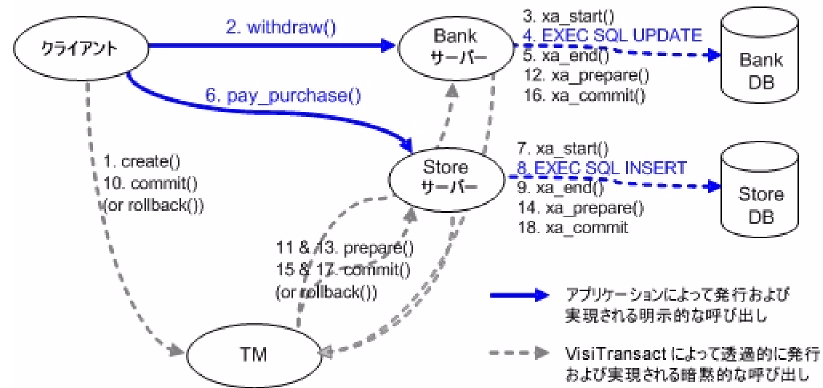
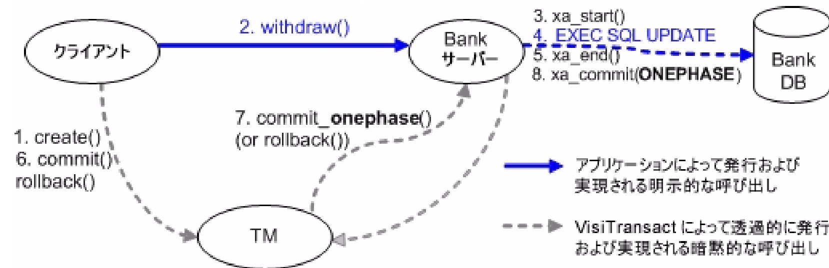


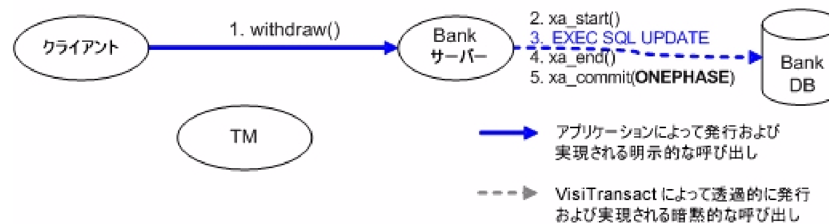
図 19.1　2 つの RM が参加するクライアント初期化トランザクション



PMT による透過的なサーバー初期化トランザクション

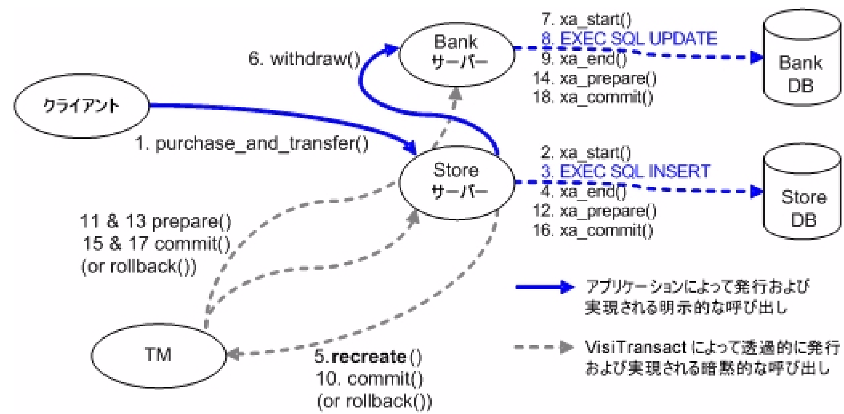
PMT は、トランザクションのすべての部分をビジネスロジック開発者から隠蔽し、クライアント開始またはサーバー開始グローバルトランザクションを最適に実行する方法を提供します。

たとえば、サーバー初期化トランザクションの場合、PMT は、次の図に示すように、ローカルトランザクションの最適化機能を使用して、グローバルトランザクションをローカルに開始します。



ローカルに開始されたグローバルトランザクションは、次の図のステップ 5 に示すように、サーバーが別の外部トランザクションオブジェクトに一方向呼び出しを行う前に、外部 TM にエキスポートされます。

図 19.2　1 つの RM が参加し、ローカルトランザクション最適化を含むサーバー初期化トランザクション



PMT は、データベースアクセスとトランザクションの詳細を完全に隠蔽します。サーバー側インプリメンテーションでは、未指定のデータベースにアクセスするために埋め込み SQL（または ODBC/CLI）を作成することだけが必要です。特定のデータベース接続もトランザクション制御ステートメントも必要ありません。すべてのデータベース接続およびトランザクション管理タスクは、サーバー側 ORB と POA エンジンに組み込まれた PMT によって暗黙的に実行されます。結果として、deposit() のコードは、次の例に示すように簡単になります。

```
void BankImpl::deposit(const char* id, float amount) {
        EXEC SQL BEGIN DECLARE SECTION;
            const char* account_id = id;
            float deposit_amount = amount;
        EXEC SQL END DECLARE SECTION;

        EXEC SQL UPDATE account_table
            WHERE account_id = :account_id
            SET balance = balance + :deposit_amount;
}
```

ここには、接続管理コードもトランザクション管理コードもありません。この同じビジネスロジックインプリメンテーションをクライアント初期化トランザクションでもサーバー初期化トランザクションでも透過的に使用できます。

PMT の概要

PMT はプログラムによって設定されます。サーバーの設定とトランザクション属性の設定は、通常の POA 作成コード、つまり POA 作成ポリシーで構成する必要があります。

PMT は、広く使用されている EJB と CCM のコンテナ管理トランザクション（CMT）にならってモデル化されています。したがって、ほとんどの CMT の概念と機能を PMT に直接適用できます。

メモ　POA 暗黙管理のトランザクション（PMT）では、アプリケーションからトランザクションを明示的に一時停止または再開したり、トランザクションコーディネータ／ターミネータを取得してはなりません。

PMT トランザクション属性値

PMT では、サーバントインプリメンテーションだけがビジネスロジックを実装します。関連するトランザクションの詳細は、特定のオブジェクトに割り当てられるトランザクション属性と、POA PMT_ATTRS_TYPE のポリシーに記述されるメソッドによって決まります。トランザクション属性は、POA ポリシーを使用して、次のように設定できます。

- *PMT_NotSupported*
  伝達コンテキストはトランザクション Current にコピーされません。POA は、クライアントのトランザクション（T1）に参加することも、サーバーを起動して新しいグローバルトランザクション（T2）を開始することもありません。これはデフォルトの PMT 属性です。この設定は、現在の作業スレッドをグローバルトランザクションに関連付けるオーバーヘッドを回避するために非トランザクションメソッドで使用する必要があります。
- *PMT_Required*
  POA は、クライアントからの要求がグローバルトランザクションコンテキストを実行する場合、クライアント開始グローバルトランザクション（T1）に参加または伝達します。そうでない場合は、その要求のために新しいグローバルトランザクション（T2）を開始および終了します。これは、トランザクションメソッドの最も役立つ PMT 属性設定です。これにより、ビジネスロジックは常に 1 つのトランザクションと 1 つの XA 接続によって実行されます。この属性は、従来の TP 製品では "AUTOTRAN" と呼ばれていました。
- *PMT_Supports*
  POA は、クライアントからの要求がグローバルトランザクションコンテキストを実行する場合、クライアント開始グローバルトランザクション（T1）に参加または伝達します。そうでない場合は、要求が CIT に含まれない限り、トランザクションを開始しません。ヌル XA リソース（160 ページの「XA リソースの設定」を参照）と組み合わされた場合、この PMT 設定は、通常、トランザクションの伝達に使用されます。
- *PMT_RequiresNew*
  POA はクライアントのグローバルトランザクションに参加または伝達しませんが、各クライアント要求に対して常に新しいグローバルトランザクション（T2）を開始または終了します。パフォーマンスを向上させるには、バックエンドデータベース上の読み取り（照会）実行オペレーションだけを含むすべてのビジネスロジックで、この PMT 設定を使用します。
- *PMT_Mandatory*
  POA は、コンテキスト内にある場合、常にクライアントのグローバルトランザクションに参加または伝達します。そうでない場合は、クライアントがトランザクションを開始していない限り、例外を生成します。
- *PMT_Never*
  POA は、クライアントトランザクションコンテキスト内にある場合、例外を生成します。

メモ PMT は、リモート要求にのみ適用されます。共用呼び出し（POA によってディスパッチされた場合でも）は、PMT 設定に関係なく、クライアントのトランザクション内（ある場合）に残ります。POA 開始サーバートランザクション（T2）が外部 TM にエキスポートされていない場合、サーバントインプリメンテーション内のトランザクション Current のメソッドを呼び出すことはできません。

次の表に、PMT トランザクション属性モードとそのセマンティクス動作をまとめます。

PMT 属性モード	クライアントのトランザクション	POA のトランザクション
NotSupported	なし	なし
	T1	なし
Required	なし	T2
	T1	T1
Supports	なし	なし
	T1	T1
RequiresNew	なし	T2
	T1	T2

PMT 属性モード	クライアントのトランザクション	POA のトランザクション
Mandatory	なし	TRANSACTION_REQUIRED
	T1	T1
Never	なし	なし
	T1	INVALID_TRANSACTION

アプリケーションプログラムでは、PMT 属性ポリシーを指定することで、POA の作成時に特定の POA と特定のオブジェクトのトランザクション属性を指定できます。

PMT 属性の POA 作成ポリシーの値は、次のように定義される PMTAttr 構造体のシーケンスです。

```
module VISTransactions {
    ...
    enum PMTMode {
        PMT_NotSupported = 1,
        PMT_Required = 2;
        PMT_Supports = 3;
        PMT_RequiresNew = 4;
        PMT_Mandatory = 5;
        PMT_Never = 6;
    };

    struct PMTAttr {
        CORBA::OctetSequence    oid;
        string          method_name;
        PMTMode             mode;
        string          xa_resource;
    };

    typedef sequence<PMTAttr>    PMTAttrSeq;

};
```

この PMTAttr 構造体の定義で以下を指定できます：

- **oid** フィールドは、この PMT 属性設定を適用するオブジェクトの ID を指定します。oid が空のシーケンス（長さ 0）の場合、この属性設定は、この POA のすべてのオブジェクトに適用されます。詳細は、「動的規則」のリストを参照してください。
- **method_name** フィールドは、この PMT 属性設定を適用する要求のオペレーション名を指定します。method_name を * に設定すると、この属性設定は、この POA のオブジェクトに送信するすべての要求オペレーションに適用されます。
- **mode** フィールドは、この PMT モード属性設定のモードを指定します。
- **xa_resource** フィールドは、関連付ける事前設定 XA リソースの名前を指定します。詳細は、160 ページの「XA リソースの設定」を参照してください。このフィールドが空の文字列またはヌルの場合は、create_POA() で PortableServer::POA::InvalidPolicy 例外が発生します。リテラル `null` は、特別に予約された xa-resource 名です。この名前は、XA リソース記述子で物理 XA リソースの命名には使用できず、PMTAttr の xa_resource フィールドの値としてのみ使用できます。要求条件が、xa_resource フィールドがヌルに等しい特定の PMTAttr の 1 つと一致する場合、PMT エンジンは、要求処理作業スレッドをどの物理 XA 接続とも関連付けません。そのかわり、PMT エンジンは、サーバントインプリメンテーションメソッドが次の層への一方向呼び出しを行う場合に関連する OTS コンテキストが伝達されるようにします。`<installation_directory>¥examples¥vbroker¥Transaction` ディレクトリにある `oci` の例を参照してください。

1 つの要求に 0 個、1 個、または 2 個の属性設定を適用できます。PMT エンジンは、次の規則を順に使用して、適用する PMT モードまたは属性を決定します。

1. 一方向メソッド、擬似メソッド、または IDL インターフェース属性の設定／取得メソッドの場合は、PMT 属性設定に関係なく、PMT モード NotSupported が適用されます。
2. oid フィールドが要求ターゲットのオブジェクト ID に正確に一致し、method_name フィールドが要求のオペレーション名に正確に一致する PMT 属性が適用されます。

3 oid フィールドが要求ターゲットのオブジェクト ID に正確に一致し、method_name フィールドにワイルドカード（*）を含む PMT 属性が適用されます。

4 method_name フィールドが要求のオペレーション名に正確に一致し、oid フィールドが空（ワイルドカードを表す）である PMT 属性が適用されます。

5 oid フィールドが空（ワイルドカードを表す）で、method_name フィールドにワイルドカード（*）を含む PMT 属性が適用されます。

6 他のどの規則も適用されない場合は、PMT モード NotSupported が適用されます。

PMT は、OMG 標準化 POA OTS ポリシーに依存しません。サーバー側トランザクションエンジンは、最初にターゲット POA の OTS ポリシーを受け取った要求コンテキストと比較して、OMG 指定の例外（INVALID_TRANSACTION または TRANSACTION_REQUIRED）を発生させるかどうかを決定します。例外が発生しない場合、要求は PMT に転送されます。

- OTS ポリシーが POA 作成要素の属性として指定されておらず、PMT ポリシーが指定されている（空の PMTAttr シーケンス以外）場合、それは OTS ADAPTS を意味します。
- OTS ポリシーも PMT ポリシーも指定されていない（または PMT ポリシーに空の PMTAttr シーケンスが指定されている）場合は、OTS ポリシーが NONE であることを意味し、OTS コンポーネントはエキスポートする IOR に追加されません。

簡単な例

```
CORBA::PolicyList policies;
policies.length(1);

PortableServer::ObjectId_var objId=
PortableServer::string_to_ObjectId("account_object");
PMTAttrSeq pmt_seq;
pmt_seq.length(1);

pmt_seq[0].oid              = (CORBA::OctetSequence&) objId;
pmt_seq[0].method_name      = (const char*)"withdraw";
pmt_seq[0].mode             = VISTransactions::PMT_Required;
pmt_seq[0].xa_resource      = (const char*)"account_storage";

CORBA::Any policy_value;
 policy_value <<= pmt_seq;
 policies[0] = orb->create_policy(VISTransactions::PMT_ATTRS_TYPE,
policy_value);

// 適切なポリシーで myPOA を作成します。
   PortableServer::POA_var myPOA = rootPOA-
>create_POA("account_server_poa",
                              poa_manager,
                              policies);
```

この例では、次のようになります。

- PMT 対応の POA が `account_server_poa` という名前で作成されます。
- ID が `account_server_poa` のターゲットオブジェクトに対する呼び出しにより、`withdraw` に等しいオペレーションが `PMT_Required` ポリシーで実行されます。これにより、クライアント初期化トランザクション（T1）に参加するか、クライアントがトランザクションを開始しなかった場合は、POA が新しいグローバルトランザクション（T2）を開始します。
- 名前が `account_storage` の xa-resource が、このトランザクションによって使用されます。

PMT::Current および接続名

OMG OTS は、アプリケーションが情報を取得したり、スレッド固有のクライアント初期化トランザクションを処理するために使用する CosTransactions::Current オブジェクトを定義しています。

さらに PMT は、アプリケーションが POA によってスレッドに関連付けられたトランザクションと接続に関する情報を取得するために使用するオブジェクト PMT::Current を追加提供しています。

```
class PMT_Current {
 public:
    static const PMT_Current* instance();

    const char*    resourceName() const;
    const char*    connectionName() const;

    // XA 診断
 const xid_t*   xid() const;
 int            rmid() const;

 // PMT 診断（この 2 つのメソッドは例外を生成しません）
 int            attribute() const;

 int            decision() const;

;
```

現在の作業スレッドに関連付けられた接続の名前は、現在のオブジェクトの connectionName() から返されます。この名前を使用して、指定した接続を使用するように埋め込み SQL ステートメントに指示できます。それには、次の例に示すように、`AT <conn_name>` 節または `SET CONNECTION <conn_name>` ステートメントを使用します。

```
void BankImpl::deposit(const char* id, float amount) {
        EXEC SQL BEGIN DECLARE SECTION;
             const char* account_id = id;
             float deposit_amount = amount;
             const char* conn = current->connectionName();
        EXEC SQL END DECLARE SECTION;

        EXEC SQL   AT :conn UPDATE account_table
             WHERE account_id = :account_id
             SET balance = balance + :deposit_amount;
}
```

接続名は、CLI（Call Level Interface）の接続ハンドルの概念に似ています。

前の例の AT 節はオプションの場合があります。たとえば、Oracle には、制御スレッドによって最後に開かれた接続を表す「*デフォルト接続*」という概念があります。埋め込み SQL に AT 節がない場合、Oracle はデフォルト接続を使用します。Sybase などの他のデータベースには、デフォルト接続という概念がないため、これらのデータベースでは、AT 節または SET CONNECTION ステートメントを使用することをお勧めします。

次の PMT::Current メソッドは、診断のために使用されます。

- `const char* PMT::Current::resourceName() const:`
  関連付けられた XA 接続によって使用される XA リソース名を返します。詳細は、160 ページの「XA リソースの設定」を参照してください。
- `const xid_t* PMT::Current::xid() const:`
  現在のスレッドに関連付けられているトランザクションの XID を返します。トランザクションが関連付けられていない場合、このメソッドは `CosTransactions::unavailable` 例外を生成します。
- `int  rmid() const;`
  現在のスレッドに関連付けられた XA 接続の XA リソースマネージャ ID を返します。XA 接続およびトランザクションが関連付けられていない場合、このメソッドは `CosTransactions::Unavailable` 例外を生成します。
- `int  attribute() const;`
  現在の {POA, oid, method-name} の組み合わせに一致するか、ワイルドカードによって

一致する PMT 属性の PMT 属性モードを返します。PMT が POA 上で有効でない場合、戻り値は 0 です。

- `int decision() const;`
  値 1 または 2 を返し、それぞれ現在のスレッドがクライアント初期化トランザクションまたはサーバー初期化トランザクションに関連付けられていることを表します。PMT が POA 上で有効でない場合、戻り値は 0 です。

XA リソースの設定

xa-resource-descriptor

VisiTransact では、XA リソースも、`xa-resource-descriptor` という名前の XML 記述子を使用して設定されます。xa-resource-descriptor は xa-resource-descriptor XML ファイルのルート要素で、通常は次に示す構造を持ちます。

```
<?xml version="1.0"?>
<!DOCTYPE xa-resource-descriptor SYSTEM "xaresdesc.dtd">

<!-- xa-resource-descriptor の例 -->
<xa-resource-descriptor>
        ...

</xa-resource-descriptor>
```

<xa-resource-descriptor> ルート要素は、次の構造に示すように、さらに 1 個以上の <xa-resource> サブ要素と、0 個以上の <xa-resource-alias> 要素を持つことができます。

```
<!ELEMENT xa-resource-descriptor
      (xa-resource+,
      xa-resource-alias*)
>
```

xa-resource

<xa-resource> は XA リソースサプライヤを定義および設定します。サブ要素の <xa-connection> は、特定の xa-resource に対して開く接続を定義します。<xa-resource> の DTD は次のとおりです。

```
!ELEMENT xa-resource
    (xa-connection+)
>
ATTLIST xa-resource
   name          CDATA              "default"
   xa-library    CDATA              #IMPLIED
   xa-switch     CDATA              #REQUIRED
   xa-conn-scope  (thread|process)  #REQUIRED
>
```

<xa-resource> は、1 個以上の <xa-connection> サブ要素を指定します。これを使用して、次の属性を設定できます。

- *name*
  この xa-resource に固有の名前を指定します。この名前は、ディスパッチされた要求をどの xa-resource に関連付けるかを決定するために PMT <transaction> 要素によって使用されます。デフォルト値は、`default` です。
- *xa-library*
  XA API ライブラリのライブラリファイル名を指定します。これは、データベースベンダーから提供されます。この属性を指定しなかった場合、エンジンは、アプリケーション実行可能モジュール自体から XA を解決しようとします。
- *xa-switch*
  `xa_switch_t` 変数のシンボルを指定します。たとえば、Oracle XA の場合、このシンボルは `xaosw`、Informix の場合は `infx_xa_switch`、DB2 の場合は `db2xa_switch` です。

- *xa-conn-scope*
  XA ライブラリから提供される XA 接続の範囲を指定します。これは、使用される XA API ライブラリ、および使用される XA オープン文字列（<xa-connection> 要素内の `info` 属性）に依存します。

データベース	xa-library	xa-switch	xa-conn-scope
Oracle 9 および 10	libclntsh.so/sl/a Oraclient9.dll	Xaosw	"thread"（info に "+Threads=true" が含まれる場合） "process"（info に "+Threads=false"[1] が含まれる場合）
Informix 7	[lib]infxxa.[extension]	infx_xa_switch	"thread"
DB2 8	[lib]db2.[extension]	db2xa_switch	
Direct Connection Driver			"process"

1. "+Threads=false" の場合、Oracle XA ライブラリはスレッドセーフではないため、Oracle XA では xa-conn-scope="process" モードを使用しないでください。

xa-connection

<xa-connection> 要素は、特定の XA 接続の名前と xa_open info 文字列を指定します。<xa-connection> 要素の DTD は次のとおりです。

```
<!ELEMENT xa-connection
       EMPTY
>
<!ATTLIST xa-connection
          name   CDATA             #IMPLIED
          info   CDATA             #REQUIRED
>
```

- *name*
  接続の名前。この名前は、接続がスレッドに関連付けられている場合、PMT::Current::connectName() メソッドから返されます。この名前は、*info* 文字列で割り当てられている名前と一致する必要があります。

XA API	接続名を指定する Info サブ文字列
Oracle XA	"+DB=<name>"
Informix XA	"+CON=<name>"
Sybase XA	"-N=<name>"
DB2 XA	"DB=<name>"

■ *info*
xa_open() に渡される文字列。この文字列で指定される情報は、XA プロバイダによって異なります。次の表に、標準的な設定テンプレートを示します。

<table>
<tr><th>XA API</th><th>標準的な info 文字列テンプレート（[] 内はオプション）</th></tr>
<tr><td>Oracle XA</td><td>"Oracle_XA+Acc=P/[<uid>]/
[<pwd>]+SqlNet=<dblink>+SesTm=<timeout>[+Threads=<true|false>]
[+LogDir=<dir>][+DbgFlag=<0x0 to 0x7>][+DB=<conn_name>]"</td></tr>
<tr><td>Informix XA</td><td>"[DB=<dbname>][;USER=<uid>][;PASSWD=<pwd>][;RM=<server>][;CON=
<conn_name>]"</td></tr>
<tr><td>Sybase XA</td><td>"-U<uid> -P<pwd> [-L<logfile>] [-T<traceflg>] [-V12] [-O<1|-1>] [-N<lrm>]"</td></tr>
<tr><td>DB2 XA</td><td>"[UID=<uid>][,PWD=<pwd>][,TPM<tpm>][,DB=<conn_name>]"</td></tr>
</table>

PMT XA エンジンは、ユーザーから提供された情報を使用して XA オープン接続を開きます。<xa-resource> の `xa-conn-scope` 属性の値が `process` の場合、VisiTransact は、指定された接続を一度に 1 つ開いて、1 つのスレッドに関連付けます。この属性の値が `thread` の場合、VisiTransact は、特定の作業スレッドをトランザクションに関連付けるときに、作業スレッドごとに 1 つの接続を開きます。

xa-resource-alias

<xa-resource-alias> 要素は、すでに定義されている <xa-resource> 要素のエリアス名を定義します。

```
<!ELEMENT xa-resource-alias
    EMPTY
>
<!ATTLIST xa-resource-alias
      name         CDATA #REQUIRED
      xa-resource    CDATA #REQUIRED
>
```

■ *name*
xa-resource のエリアス。

■ *xa-resource*
このエリアスがポイントする実際の xa-resource。

xa-resource-alias の名前が PMT <transaction> 要素の xa-resource 属性から参照されると、実際の xa-resource が使用されます。

XA リソース記述子の例

次の例は、xa-resource-descriptor の完全な記述を示します。

```
<?xml version="1.0"?>
<!DOCTYPE xa-resource-descriptor SYSTEM "xaresdesc.dtd">

<!-- xa-resource-descriptor の例 -->
<xa-resource-descriptor>

        <!-- 1. xa リソースのリスト -->
        <xa-resource
               name="oracle"
               xa-switch="xaosw"
               xa-conn-scope="thread"
        <
               <!- 2. xa 接続のリスト -->
               <xa-connection
              info=
                        "Oracle_XA+Acc=P/scott/
tiger+SesTm=10+SqlNet=ora92a+Threads=true"
               />
        </xa-resource>

        <!-- 3. リソースエリアスのリスト -->
        <xa-resource-alias
               name="default"
               xa-resource="oracle"
        />

        <xa-resource-alias
            name="account-storage"
            xa-resource="oracle"
        />

</xa-resource-descriptor>
```

上の例の内容は次のとおりです。

- xa-resource-descriptor には、`oracle` という名前の 1 つの xa-resource が含まれています。
- この xa-resource は、xa-switch シンボル `xaosw` を指定しますが、xa-library ファイル名は指定しません。したがって、VisiTransact は、外部からロードされたライブラリではなく、現在の実行可能モジュール内で xa スイッチを解決します。アプリケーションがデータベースのクライアントライブラリにすでにリンクされており、そこに必要な XA API が含まれていることが多いため、これは標準的な使用例です。
- xa-conn-scope は `thread` に設定されます。これは、xa-connection の `info` 属性の `+Threads=true` サブ文字列と一致しています。この場合、VisiTransact は、スレッドをトランザクションに関連付けるときに、作業スレッドごとに専用の XA 接続を 1 つ開きます。
- xa-connection 要素では、名前属性と、`info` 文字列の `+DB=<name>` サブ文字列を省略しました。これは、スレッドモードの Oracle XA アプリケーションでは標準的な使用例です。埋め込み SQL はデフォルト接続を使用します。アプリケーションでは、必ずしも AT 節を使用する必要はありません。
- <xa-resource-alias> 要素が `default` という名前を使用して定義され、すでに定義されている oracle <xa-resource> をポイントします。PMT <transaction> 要素が <xa-resource> 名 `default` を使用して定義されると、参照された `oracle` xa-resource が使用されます。
- 追加の <xa-resource-alias> 要素が `account-storage` という名前を使用して定義され、すでに定義されている oracle <xa-resource> をポイントします。PMT <transaction> 要素が <xa-resource> 名 `account-storage` を使用して定義されると、参照された `oracle` xa-resource が使用されます。

VisiTransact のプロパティ

vbroker.its.its6xmode=<false|true>

`false` に設定すると、すべての VisiTransact PMT 機能と最適化が有効になります。`true` に設定すると、PMT 拡張機能と最適化は無効になり、次に示す非推奨の機能が有効になります。

- トランザクションアプリケーションはインプロセス OTS を使用します。
- POA は OTS ポリシー付きで作成されませんが、その POA 上のオブジェクトは CosTransactions::TransactionalObject から継承されます。
- アプリケーションはクライアント側で NonTxTargetPolicy を使用します。
- アプリケーションは SessionManager を使用します。
- VisiTransact OTS サーバーが VBJ Java クライアントと VBJ サーバーによって使用されます。

このプロパティは、パフォーマンス比較、バグの分離、および後方互換性の要件のために用意されています。デフォルト値は `false` です。

vbroker.its.verbose=<false|true>

true に設定すると、VisiTransact は、ランタイム情報として低レベルの例外と警告を出力します。デフォルト値は `false` です。

vbroker.its.xadesc=<xa-resource xml file name>

このプロパティを使用して、XA-resource 設定ファイルを指定します。デフォルト値は、`itsxadesc.xml` です。

RM リカバリユーティリティ

2 フェーズコミットメカニズムは、すべてのノードが一緒にコミットまたはロールバックされるようにします。2 フェーズコミットの途中で、ネットワークの問題、データベースのクラッシュ、または未処理のソフトウェアエラーによって障害が発生した場合、トランザクションは未確定になり、データベース内のリソースはロックされたまま解放されません。この問題を解決するために、VisiTransact には、RM リカバリユーティリティ rmrecover（Windows では rmrecover.exe）と自動 TM リカバリが用意されています。

Borland は、トランザクションに関わる Resource Manager ごとにこのユーティリティを実行してから、障害で終了した VisiTransact アプリケーションサーバーを再起動することをお勧めします。

rmrecover は次のように使用します。

% rmrecover <xa_resource_desc.xml> [<options>]

- <xa_resource_desc.xml> は、データベースに接続するために RM によって使用される xa-resource 設定です。
- <options> は xa-resource 名を指定します。

リカバリユーティリティを実行するには、次の手順を実行します。

1. <xa_resource_desc.xml> 内のユーザー ID とパスワードを変更して、データベース管理権限を取得します。
2. オペレーティングシステムに合わせて、<xa_resource_desc.xml> で Oracle クライアントライブラリを設定します。
   - Windows の場合：`xa-library="oraclient9.dll"`
   - UNIX の場合：`xa-library="libclntsh.so"`
3. 特定のポートでトランザクションサービス ots（Windows では ots.exe）を起動します。
   ots -Dvbroker.se.iiop_tp.scm.iiop_tp.listener.port=<port number>
4. rmrecover ユーティリティを起動します。
   rmrecover -ORBInitRef VisiTransactionService=corloc::<host>:<port>/VisiTransactionService <xa_resource_desc.xml> <xa-resource name>

RM リカバリユーティリティは、データベースにコンタクトして、未確定のトランザクションのリストを取得し、各トランザクションをコミットまたはロールバックします。

第 20 章

XA Session Manager for Oracle OCI、version 9i Client

この章では、OCI（Oracle Call Interface）データベースの Oracle9i バージョンを XA セッションマネージャインプリメンテーションとともに使用する場合の問題について説明します。この章には、次の節があります。

- 167 ページの「概要」
- 168 ページの「Oracle9i のソフトウェア要件」
- 169 ページの「Oracle9i のインストールと設定の問題」
- 170 ページの「必要な環境変数」
- 170 ページの「セッションマネージャの接続プロファイル属性」
- 171 ページの「プログラミングの制限事項」
- 172 ページの「トラブルシューティング」

概要

この章では、OCI（Oracle Call Interfaces）の Oracle9i バージョンと Oracle9i データベースを XA トランザクション調整を使った VisiTransact とともに使用するにあたって、データベースに固有の問題と要件について説明します。これには、ソフトウェア要件、インストールと設定情報、セッションマネージャと XA リソースディレクタの設定属性、およびプログラミングの制限事項についての説明が含まれます。

VisiTransact トランザクションのデータアクセスは、OCI と Oracle XA ライブラリに対してセッションマネージャを使用することによって発生します。データベース接続は、アプリケーションがセッションマネージャから接続オブジェクトを要求すると確立されます。これにより、アプリケーションはネイティブハンドルを取得して、通常の OCI 呼び出しに使用できます。

この章では、標準 XA コミットプロトコルを使った VisiTransact で Oracle DBMS にアクセスするための要件を中心にして説明します。Oracle を正しくインストールして設定するには、Oracle データベースに付属するマニュアルの説明にしたがう必要があります。

この章の想定読者

このデータベースを管理するシステム管理者とデータベース管理者は、トランザクション処理に使用する DBMS をインストールして設定する前に、この章をお読みください。特に、以下の節を参照してください。VisiTransact を使ってアプリケーションを構築するアプリケーション開発者は、171 ページの「プログラミングの制限事項」を参照してください。

節	システム管理者	データベース管理	アプリケーション開発者
Oracle9i のソフトウェア要件	X	X	
Oracle9i のインストールと設定の問題	X	X	
必須環境変数、トラブルシューティング	X	X	X
セッションマネージャの接続プロファイル属性	X	X	
プログラミングの制限事項		X	X
トラブルシューティング	X	X	X

Oracle9i のソフトウェア要件

XA リソースディレクタまたはセッションマネージャを使って構築されたアプリケーションを実行するすべてのマシンに Oracle9i クライアントライブラリをインストールする必要があります。XA リソースディレクタとセッションマネージャは、VisiTransact のコンポーネントです。

次の節では、プラットフォームごとのデータベースクライアントとサーバーの要件をリストします。

クライアントの要件

次の Oracle OCI 用 Oracle クライアントコンポーネントは、XA リソースディレクタまたはセッションマネージャを使って構築されたアプリケーションを実行している各ノードにインストールして設定する必要があります。

- Solaris の場合は Oracle OCI、バージョン 9i
- Oracle XA ライブラリ

サーバーの要件

データベースの各サーバーマシンでは、次の Oracle Server のコンポーネントをインストールして設定する必要があります。

- Oracle Server、バージョン 9i
- Oracle Distributed Database オプション

Oracle9i のインストールと設定の問題

次の節では、Oracle のインストールとソフトウェアの設定に関する問題を説明します。

インストール要件

Oracle をインストールするには、次の条件が必要です。

- Oracle のインストールと設定ガイド
- 対応するリリースニュース

データベース設定

次の表に示すように **init.ora** パラメータを使用して、データベースを XA Session Manager for Oracle OCI とともに使用するように設定します。

init.ora パラメータ	**説明**
transactions	データベースが同時に扱える分散トランザクションの数。
sessions	ユーザーセッションとシステムセッションの合計数。
processes	このパラメータの設定の詳細については、『*Oracle9i Server Administrator's Guide*』を参照してください。
distributed_lock_timeout	ロックされたリソースを分散トランザクションが待機する時間（秒）。

VisiTransact では、分散トランザクションの数はデータベースの **init.ora** パラメータ `transactions` によって制限されます。トランザクションは、最初の `getConnection()` 呼び出しまたは `getConnectionWithCoordinator()` 呼び出しから、コミットまたはロールバックが完了するまで、アクティブな状態が継続します。原則として、`transactions` のデフォルトはセッションマネージャで使用するには低すぎます。このデフォルトはシステムに依存しています。

Oracle OCI では、接続とは対照的に、それぞれの分散トランザクションが 1 つのデータベースセッションを使用します。分散トランザクションセッションやほかのアプリケーションセッションに対応できるだけの大きさに **init.ora** パラメータの `sessions` と `processes` を設定してください。

一部のプラットフォームでは、XA のような分散トランザクションを使用するとほかの Oracle 機能の使用が制限される場合があります。たとえば、一部のプラットフォームでは、Oracle Parallel Server オプションの使用が制限されることがあります。

メモ **init.ora** パラメータの設定方法について、および Oracle XA と Oracle Parallel Server や Oracle Replication などのほかの Oracle 機能の対話については、Oracle のマニュアルを参照してください。

DBA_PENDING_TRANSACTIONS ビュー

ビュー DBA_PENDING_TRANSACTION は、リカバリ処理の際にデータベースと VisiTransact トランザクションサービスの間でトランザクション情報を同期するために、XA リソースディレクタによって使用されます。Oracle9i セッションマネージャプロファイルの中で Oracle ユーザー ID を指定されたすべてのユーザーは、このビューで SELECT 権限を与えられる必要があります。

ビューへのアクセス許可が正しく、リカバリ処理を開始できることを確認するには、XA リソースディレクタのユーザー ID として **SQL*Plus** を使って Oracle にログインしてから、次のクエリーを実行します。

```
select count (*) from SYS.DBA_PENDING_TRANSACTIONS;
```

Oracle のエラー「ORA-00942：テーブルまたはビューが存在しません」が返された場合、XA リソースディレクタはこのビューにアクセスできません。ユーザー **sys** または system としてログオンするか、内部のサーバーマネージャから接続して、このビューの SELECT 権限を適切なユーザーに許可してください。

必要な環境変数

PATH 環境変数には、データベースクライアントライブラリがインストールされている Oracle クライアントディレクトリのパス、およびセッションマネージャライブラリのパスが含まれている必要があります。

LD_LIBRARY_PATH
PATH

PATH に $ORACLE_HOME/bin を追加し、LD_LIBRARY_PATH に $ORACLE_HOME/lib32（または 64 ビットアプリケーションでは $ORACLE_HOME/lib）を追加します。たとえば、Borne シェルでは次のようにします。

```
LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:${ORACLE_HOME}/lib32
PATH=${ORACLE_HOME}/bin:${PATH}
```

セッションマネージャの接続プロファイル属性

次の表は、XA Session Manager for Oracle OCI に固有の設定プロファイル属性を示します。

属性	UI ラベル	説明	範囲
heartbeat_retry_period	Heartbeat Retry Period	ハートビートが失われた後の VisiTransact トランザクションサービスインスタンスへのハートビート間隔の秒数。リカバリを開始するために VisiTransact トランザクションサービスのインスタンスの再アクティブ化を検出するときに使用されます。リソースディレクタ内でのみ使用されます。	> 0
heartbeat_watch_period	Heartbeat Watch Period	VisiTransact トランザクションサービスインスタンスへのハートビート間隔の秒数。VisiTransact トランザクションサービスのインスタンスの障害を自動的に検出するために使用されます。リソースディレクタ内でのみ使用されます。	> 0
oracle_txn_idle_timeout	Transaction Idle Timeout	準備されていないトランザクションのアイドル状態が続いてタイムアウトになり、Oracle がトランザクションをロールバックするまでの時間（秒）この属性を使用する場合は、VisiTransact トランザクションサービスに設定されたタイムアウト時間を考慮してください。	> 0
oracle_xa_logdir	Log Directory Path	Oracle XA ログファイルが書き込まれるディレクトリのパス。	0 から 256 文字まで
resource_director_name	Resource Director	使用するリソースディレクタの名前。	1 から 128 文字まで
native_handle_type	Native Handle Type	アプリケーションが要求するネイティブ接続ハンドルのタイプ。	有効な値は **Lda_Def** と **ITSoracle9i_handles**

OCI 9i API でのセッションマネージャの使用

Oracle9i では、OCI インターフェースは全面的に書き直されました。この新しいインターフェースでは、SQL 文を実行するためにいくつかのハンドルが必要です。この API をセッションマネージャで使用するには、次の手順を実行します。

1. 接続プロファイルで、属性 `native_handle_type` を **ITSoracle9i_handles** に設定します。
2. アプリケーションのソースでファイル ora9i_sessmgr.h をインクルードして、オブジェクト **ITSoracle9i_handles** を定義します。
3. `Connection::getNativeConnectionHandle()` の戻り値を **ITSoracle9i_handles ***（ITSoracle9i_handles 型のオブジェクトへのポインタ）型にキャストします。
4. クラス **ITSoracle9i_handles** が提供するアクセッサメソッドを使用して、必要な各種のハンドルを取得します。次のようなメソッドがあります。
   - OCISvcCtx *getSvcCtx();
   - OCIEnv *getEnv();
   - OCIError *getError();

ITSoracle9i_handles オブジェクトを通じて取得したオブジェクトを解放しないでください。これらのオブジェクトインスタンスはセッションマネージャが管理します。

プログラミングの制限事項

次の制限事項は、トランザクション処理のためにアプリケーションをプログラミングするときに適用されます。

接続オブジェクトは、それを作成したスレッドの中で使用する必要があります。
これは Oracle9i XA インプリメンテーションの制限であり、接続オブジェクトを取得したスレッドの中では、接続オブジェクトインスタンスから取得したネイティブ接続ハンドルだけを使用できます。この接続ハンドルをほかのスレッドで使用すると、予期しない結果が生じます。

アプリケーションの中では DDL 文を使用しません。
この制限は、DDL SQL 文は Oracle XA アプリケーションでサポートされていないということを意味します。これは、`CREATE TABLE` などの DDL SQL 文は暗黙のコミットを実行するためです。DDL 文が必要な場合は、XA プロトコルを使用しないプロセスで実行する必要があります。

次の表に示した操作は、セッションマネージャを通して取得された接続では使用できません。

操作	許可されていない SQL コマンド	9i API で許可されていない OCI 呼び出し
接続操作	CONNECT	OCISvcCtxLogon OCISvcCtxLogoff
トランザクション操作	COMMIT ROLLBACK SAVEPOINT SET TRANSCTION (READ ONLY|READWRITE|USE ROLLBACK SEGMENT)	OCITransCommit OCITransRollback OCIStmtExecute (OCI_COMMIT_ON_SUCCESS モード)
暗黙的操作	DDL SQL 文（CREATE TABLE、CREATE INDEX など）	

トラブルシューティング

この節では、XA Session Manager for Oracle OCI を Oracle データベースとともに使用する場合に発生する問題を明らかにして、問題のトラブルシューティングを提案します。

VisiTransact メッセージログ

VisiTransact メッセージログには、接続エラーまたはトランザクションエラーが発生したときのセッションマネージャとネイティブの Oracle のエラーメッセージが含まれています。

xa_trc ファイルの使い方

XA コードの問題を示すエラーが発生した場合、Oracle エラーについての詳細な情報は、**xa_*.trc** ファイルで調べることができます。これらのファイルは、定義された接続プロファイルの中で指定されているログディレクトリにあります。セッションマネージャ接続プロファイルの中でログディレクトリが指定されていない場合、**xa_*.trc** ファイルはプロセスの開始時に $ORACLE_HOME が利用できれば **$ORACLE_HOME/rdbms/log** ディレクトリに、**$ORACLE_HOME** が利用できなければ現在のディレクトリに置かれます。

メモ　ディレクトリが指定されていても存在しない場合、ログファイルは作成されず、そのことは警告されません。

分散更新の問題

ネットワーク障害またはシステム障害により、次のタイプの問題が発生することがあります。

- 障害が発生したときに、処理中の準備またはコミットがセッションの一部のノードで完了していないことがある。
- 障害が継続すると（たとえば、ネットワークが長時間ダウンする）、未確定トランザクション（準備されたがコミットもロールバックもされていない）によって排他的にロックされたデータをほかのトランザクションの文で使用できない。

メモ　1 つの Oracle ノードが別の Oracle データベースのサブコーディネータとして機能する場合の分散更新動作の詳細については、Oracle のマニュアルを参照してください。

データアクセス障害

ユーザーが SQL 文を発行する場合、Oracle9i は文を実行するために必要なデータをロックしようとします。ただし、要求されたデータがほかのコミットされていないトランザクションの文によって処理され、長時間ロックされたままの場合は、タイムアウトが発生します。

未確定トランザクションによるロック

未確定分散トランザクションのリソースがロックされているために、ローカルデータベースのロックを必要とするクエリーまたは DML 文がいつまでもブロックされることがあります。この場合は、次のエラーメッセージがユーザーに返されます。

ORA-01591: 未確定分散トランザクション <IDt> がロックを保持しています。

この場合、SQL 文はただちにロールバックされます。SQL 文のロールバックによってトランザクションのロールバックが自動的に行われることはありません。文を実行したアプリケーションは、後で文を再実行しようとします。ロックが継続する場合は、未確定分散トランザクションの ID を含めて管理者に問題を報告してください。

未確定トランザクションとは、準備状態のままでコミットもロールバックもされていないトランザクションです。

トランザクションタイムアウト

リモートデータベースをロックする必要がある DML 文は、要求したデータを別のトランザクションが現在ロックしていると、ブロックされることがあります。データを要求する SQL 文をこれらのロックがブロックし続けるとタイムアウトが発生し、文はロールバックされて次のエラーメッセージがユーザーに返されます。

ORA-02049: タイムアウト : 分散トランザクションがロックを待機しています。

この場合、SQL 文はただちにロールバックされます。SQL 文のロールバックによってトランザクションのロールバックが自動的に行われることはありません。アプリケーションは、デッドロックが起きたときと同様に続行します。文を実行したアプリケーションは、後で文を再実行しようとします。ロックが継続する場合は、管理者に問題を報告する必要があります。

前の状況で説明されているタイムアウト間隔は、初期化パラメータ `distributed_lock_timeout` で制御できます。この間隔は秒単位です。たとえば、インスタンスのタイムアウト間隔を 30 秒に設定するには、関連するパラメータファイルに次の行を加えます。

DISTRIBUTED_LOCK_TIMEOUT=30

このタイムアウト間隔の場合、利用できないリソースを 30 秒待機した後もトランザクションを継続できないと、前節で説明されているタイムアウトエラーが発生します。

`distributed_lock_timeout` パラメータについては、169 ページの「データベース設定」を参照してください。

Oracle エラーメッセージ

VisiTransact メッセージログには Oracle エラーメッセージが含まれており、次のような接続エラーやトランザクションのエラーのトラブルシューティングに役立ちます。

エラーメッセージ	説明	ソリューション
ORA-12154	Solaris でのファイルデスクリプタのプロセス限度（`ulimit`）が、マルチスレッドアプリケーションに対する設定としては低すぎます。	プロファイル名で正しいデータベース名を確認します。 **tnsnames.ora** ファイルで、対応するサービス名のエントリを探します。 データベースと Oracle リスナープロセスが実行中であることを確認します。 Solaris で、ファイルデスクリプタ限度（**ulimit**）の設定が接続を開くのに十分な大きさであることを確認します。 **ulimit** コマンドの設定については、Solaris オペレーティングシステムのマニュアルを参照してください。

ヒューリスティックな完了の強制

強制的にヒューリスティックなトランザクションを完了させるには、COMMIT FORCE <local transaction id> または ROLLBACK FORCE <local transaction id> を使用します。ここで、<local transaction id> は dba_2pc_pending テーブルに基づいています。詳細については、「*Oracle9i Distributed Database Systems*」マニュアルを参照してください。

第 21 章

DirectConnect Session Manager for Oracle OCI、version 9i Client

この章では、OCI（Oracle Call Interface）データベースの Oracle9i バージョンを DirectConnect セッションマネージャインプリメンテーションとともに使用する場合の問題について説明します。この章には、次の節があります。

- 175 ページの「概要」
- 176 ページの「Oracle9i のソフトウェア要件」
- 176 ページの「Oracle9i のインストールと設定の問題」
- 177 ページの「必要な環境変数」
- 177 ページの「セッションマネージャの接続プロファイル属性」
- 178 ページの「プログラミングの制限事項」
- 178 ページの「トラブルシューティング」

概要

この章では、OCI（Oracle Call Interface）の Oracle9i バージョンと Oracle9i データベースを DirectConnect セッションマネージャインプリメンテーションとともに使用するにあたって、データベースに固有の問題と要件について説明します。これには、ソフトウェア要件、インストールと設定情報、セッションマネージャの設定属性、およびプログラミングの制限事項についての説明が含まれます。これは、このマニュアルで説明されているほかの DirectConnect セッションマネージャインプリメンテーションと対照的です。

VisiBroker VisiTransact トランザクションのデータアクセスは、OCI と Oracle ライブラリに対してセッションマネージャを使用することによって発生します。データベース接続は、アプリケーションがセッションマネージャから接続オブジェクトを要求すると確立されます。これにより、アプリケーションはネイティブハンドルを取得して、通常の OCI 呼び出しに使用できます。

この章では、VisiBroker VisiTransact で Oracle DBMS にアクセスするための要件を中心にして説明します。Oracle を正しくインストールして設定するには、Oracle データベースに付属するマニュアルの説明にしたがう必要があります。

DirectConnect セッションマネージャインプリメンテーションの詳細については、99 ページの「セッションマネージャの概要」と 119 ページの「セッションマネージャを使用したデータアクセス」を参照してください。

この章の想定読者

このデータベースを管理するシステム管理者とデータベース管理者は、トランザクション処理に使用する DBMS をインストールして設定する前に、この章をお読みください。特に、以下の節を参照してください。VisiTransact を使ってアプリケーションを構築するアプリケーション開発者は、178 ページの「プログラミングの制限事項」を参照してください。

節	システム管理者	データベース管理	アプリケーション開発者
Oracle9i のソフトウェア要件	X	X	
Oracle9i のインストールと設定の問題	X	X	
必須環境変数、トラブルシューティング	X	X	X
セッションマネージャの接続プロファイル属性	X	X	
プログラミングの制限事項		X	X

Oracle9i のソフトウェア要件

セッションマネージャを使って構築されたアプリケーションを実行するすべてのマシンに Oracle9i クライアントライブラリをインストールする必要があります。セッションマネージャは、VisiBroker VisiTransact のコンポーネントです。

次の節では、プラットフォームごとのデータベースクライアントとサーバーの要件をリストします。

クライアントの要件

次の Oracle OCI 用 Oracle クライアントコンポーネントは、セッションマネージャを使って構築されたアプリケーションを実行している各ノードにインストールして設定する必要があります。

- Solaris の場合は Oracle OCI、バージョン 9i

サーバーの要件

UNIX

データベースの各サーバーマシンでは、次の Oracle Server のコンポーネントをインストールして設定する必要があります。

- Oracle9i Server

Oracle9i のインストールと設定の問題

次の節では、Oracle のインストールとソフトウェアの設定に関する問題を説明します。

インストール要件

Oracle をインストールするには、次の条件が必要です。

- Oracle のインストールと設定ガイド
- 対応するリリースニュース

データベース設定

次の表に示すように **init.ora** パラメータを使用して、データベースを DirectConnect Session Manager for Oracle OCI とともに使用するように設定します。

init.ora パラメータ	説明
sessions	ユーザーセッションとシステムセッションの合計数。
processes	このパラメータの設定の詳細については、『*Oracle9i Server Administrator's Guide*』を参照してください。

DirectConnect Session Manager for Oracle OCI では、各トランザクションはデータベースセッションを使用します。トランザクションセッションやほかのアプリケーションセッションに対応できるだけの大きさに **init.ora** パラメータの sessions と processes を設定してください。

DirectConnect セッションマネージャが開いたそれぞれの接続は、トランザクションが完了するまでセッションを必要とします。そのため、sessions パラメータは、データベースにアクセスする DirectConnect 同時処理トランザクションの最大値より大きい数値に設定する必要があります。

メモ　**init.ora** パラメータの設定方法については、Oracle のマニュアルを参照してください。

必要な環境変数

PATH 環境変数には、データベースクライアントライブラリがインストールされている Oracle クライアントディレクトリのパス、およびセッションマネージャライブラリのパスが含まれている必要があります。

UNIX　LD_LIBRARY_PATH
PATH

PATH に ORACLE_HOME/bin を追加し、LD_LIBRARY_PATH に $ORACLE_HOME/lib32（または 64 ビットアプリケーションでは $ORACLE_HOME/lib）を追加します。たとえば、Bourne シェルでは次のようにします。

```
LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:${ORACLE_HOME}/lib32
PATH=${ORACLE_HOOME}/bin:${PATH}
```

セッションマネージャの接続プロファイル属性

次の表は、XA Session Manager for Oracle OCI に固有の設定プロファイル属性を示します。

次の表は、XA Session Manager for Oracle OCI の属性を示します。

属性	UI ラベル	説明	範囲
native_handle_type	Native Handle Type	アプリケーションが要求するネイティブ接続ハンドルのタイプ。	有効な値は、Lda_Def または ITSOracle9i_handles です。

OCI 9i API でのセッションマネージャの使用

Oracle9i では、OCI インターフェースは全面的に書き直されました。この新しいインターフェースでは、SQL 文を実行するためにいくつかのハンドルが必要です。この API をセッションマネージャで使用するには、次の手順を実行します。

1 接続プロファイルで、属性 `native_handle_type` を `ITSOracle9i_handles` に設定します。

2 アプリケーションのソースでファイル ora9i_sessmgr.h をインクルードして、オブジェクト `ITSOracle9i_handles` を定義します。

3 `Connection::getNativeConnectionHandle()` の戻り値を `ITSOracle9i_handles *`（`ITSOracle9i_handles` 型のオブジェクトへのポインタ）型にキャストします。

4 クラス `ITSOracle9i_handles` が提供するアクセッサメソッドを使用して、必要な各種のハンドルを取得します。次のようなメソッドがあります。

- OCISvcCtx *getSvcCtx();
- OCIEnv *getEnv();
- OCIError *getError();

`ITSOracle9i_handles` オブジェクトを通じて取得したオブジェクトを解放しないでください。これらのオブジェクトインスタンスはセッションマネージャが管理します。

プログラミングの制限事項

次の制限事項は、VisiBroker VisiTransact と Oracle OCI を使ってトランザクション処理のアプリケーションをプログラミングする場合に適用されます。

VisiTransact トランザクションサービスとセッションマネージャは接続とトランザクション管理を制御するため、プラットフォームは次の表に示した許可されていない操作を使用しないようにしてください。

操作	許可されていない SQL コマンド	9i API で許可されていない OCI 呼び出し
接続操作	CONNECT	OCISvcCtxLogon OCISvcCtxLogoff
トランザクション操作	COMMIT ROLLBACK SAVEPOINT SET TRANSACTION	OCITransCommit OCITransRollback OCIStmtExecute (OCI_COMMIT_ON_SUCCESS モード)
暗黙的操作	DDL SQL 文（CREATE TABLE など）	

トラブルシューティング

この節では、DirectConnect Session Manager for Oracle OCI を Oracle データベースとともに使用する場合に発生する問題を明らかにして、問題のトラブルシューティングを提案します。

VisiBroker VisiTransact メッセージログ

VisiTransact メッセージログには、接続エラーまたはトランザクションエラーが発生したときのセッションマネージャとネイティブの Oracle のエラーメッセージが含まれています。

Oracle エラーメッセージ

VisiTransact メッセージログと `VISSessionManager::Error` 例外には Oracle エラーメッセージが含まれており、次のような接続エラーやトランザクションのエラーのトラブルシューティングに役立ちます。

エラーメッセージ	説明	ソリューション
ORA-01017	無効なユーザー名／パスワード	接続プロファイルのユーザー名とパスワードが正しいことを確認します。
ORA-12154	サービス名を解決できない	プロファイル名で正しいデータベース名を確認します。 **tnsnames.ora** ファイルで、対応するサービス名のエントリを探します。 データベースと Oracle リスナープロセスが実行中であることを確認します。 Solaris で、ファイルデスクリプタ限度（**ulimit**）の設定が接続を開くのに十分な大きさであることを確認します。 **ulimit** コマンドの設定については、 Solaris オペレーティングシステムのマニュアルを参照してください。

第 22 章

コマンド、ユーティリティ、引数、および環境変数

この付録では、VisiTransact コマンドと ORB_init() の引数、および VisiTransact で使用される環境変数について説明します。

VisiTransact コマンドの概要

次の表に示すように、以下で説明するコマンドは相互に関連しています。

VisiTransact コンポーネント	関連するコマンド
VisiBroker コンソール	vbconsole。このコマンドは VisiBroker コンソールを起動します。
VisiTransact Transaction Service	ots。このコマンドは、VisiTransact Transaction Service のインスタンスを起動します。 vshutdown。このコマンドは、VisiTransact Transaction Service のインスタンスをシャットダウンします。
セッションマネージャ	xa_resdir. このコマンドは、セッションマネージャの一部である XA リソースディレクタのインスタンスを起動します。 smconfig_server. このコマンドは、セッションマネージャ設定サーバーのインスタンスを起動します。 vshutdown。このコマンドは、XA リソースディレクタまたはセッションマネージャ設定サーバーのインスタンスをシャットダウンします。
セッションマネージャ設定のセットアップ	smconfigsetup。このユーティリティは、カスタマイズされたセッションマネージャを作成するためにプラグイン可能リソースインターフェースで使用する接続プロファイルを作成します。

vbconsole

このコマンドは VisiBroker コンソールを起動します。VisiBroker コンソールの実行可能ファイルがインストールされている任意のノードで実行できます。VisiBroker コンソールは、管理する VisiTransact Transaction Service のインスタンスまたはセッションマネージャ設定サーバーのインスタンスに対してローカルである必要はありません。ただし、管理するインスタンスは、VisiBroker コンソールの起動時に実行されている必要があります。

構文

```
prompt>vbconsole
```

サンプル

```
prompt>vbconsole
```

引数

なし。

ots

このコマンドは、VisiTransact Transaction Service のインスタンスを起動します。

構文

```
prompt>ots [-Dvbroker.ots.defaultTimeout=<seconds>]
      [-Dvbroker.ots.defaultMaxTimeout=<seconds>]
      [-Dvbroker.ots.name=<transaction_service_name>]
      [-Dvbroker.ots.logDir=<directory_name>]
      [-Dvbroker.log.enable=<Boolean>]
      [vbroker.ots.logPurgeTransactions=<true|false>]
      [vbroker.ots.logSleep=<milliseconds>]
      [vbroker.ots.logCache=<cache_size_in_kilobytes>]
      [vbroker.ots.logUnit=<transaction_log_size>]
```

サンプル

```
prompt>ots -Dvbroker.ots.defaultTimeout=60 -
Dvbroker.ots.defaultMaxTimeout=120
-Dvbroker.ots.name=Sales -Dvbroker.log.enable=true
```

引数

このコマンドでは、次の引数を使用できます。

引数	説明
-Dvbroker.ots.defaultTimeout=<seconds>	VisiTransact Transaction Service インスタンスのデフォルトのトランザクションタイムアウト値を設定します。 設定しないと、 デフォルトの 600 秒になります。
-Dvbroker.ots.defaultMaxTimeout=<seconds>	VisiTransact Transaction Service インスタンスの最大のトランザクションタイムアウト値を設定します。 設定しないと、 デフォルトの 3600 秒になります。
-Dvbroker.ots.name=<transaction_service_name>	スマート エージェント に VisiTransact Transaction Service のインターフェースを登録する際に使用されるインスタンス名を設定します。デフォルト は <host_name>_ots です。

引数	説明
-Dvbroker.ots.logDir=<directory_name>	ログとロガー情報を保存するディレクトリの名前を指定します。 指定しないと、デフォルトの <VBROKER_ADM>¥its¥<transaction_service_name>¥logger になります。
-Dvbroker.log.enable=<Boolean>	このサーバーのデバッグログステートメントを表示するには、このプロパティを true に設定します。デバッグログフィルタのさまざまなソース名オプションについては、『VisiBroker for C++ 開発者ガイド』の「デバッグログのプロパティ」を参照してください。
vbroker.ots.logPurgeTransactions=<true|false>	トランザクションログが新しいファイルかどうかを示します。
vbroker.ots.logSleep=<milliseconds>	キャッシュがいっぱいで物理ファイルにフラッシュする必要があるかどうかをチェックするまでのスリープ時間をミリ秒単位で示します。デフォルトは 0 です。
vbroker.ots.logCache=<cache_size_in_kilobytes>	物理ファイルにフラッシュするまでのキャッシュのサイズを示します。デフォルトは 64k です。
vbroker.ots.logUnit=<transaction_log_size>	ログファイルのサイズを示します。 デフォルトは 8M です。

smconfig_server

このコマンドは、セッションマネージャ設定サーバーのインスタンスを起動するために使用します。セッションマネージャ設定サーバーをエージェントとして使用して、データベースにアクセスする接続プロファイルを作成します。

構文

```
prompt>smconfig_server [-Dvbroker.sm.pstorePath=<path>]
        [-Dvbroker.sm.configName=<name>] [-m{32|64}]
```

サンプル

```
prompt>smconfig_server -
Dvbroker.sm.pstorePath=C:¥vbroker¥adm¥its¥session_manager
-Dvbroker.sm.configName=athena_smcs -m64
```

引数

このコマンドでは、次の引数を使用できます。

引数	説明
-Dvbroker.sm.pstorePath=<path>	永続的ストアファイルが存在するディレクトリのパスを指定します。 デフォルトでは、 永続的ストアファイルは <VBROKER_ADM>¥its¥session_manager にあります。

引数	説明
-Dvbroker.sm.configName=<name>	使用するセッションマネージャ設定サーバーの名前を指定します。デフォルトでは、セッションマネージャ設定サーバーに割り当てられた名前は <host>_smcs です。ここで host は、セッションマネージャプロファイルを作成したホストの名前です。
-m{32|64}	32 ビットまたは 64 ビットの共有プラグインライブラリ名に基づいてプロファイルを生成します。 -m32 は 32 ビットの命名に使用 -m64 は 64 ビットの命名に使用

vshutdown

このコマンドを使用して、VisiTransact Transaction Service、XA リソースディレクタ、およびセッションマネージャの接続マネージャをシャットダウンできます。

デフォルトでは、これを使って VisiTransact Transaction Service のインスタンスをシャットダウンすると、未処理のトランザクションが完了してからシャットダウンが実行されます。ただし、新しいトランザクションは受け付けられません。トランザクションを解決しないで VisiTransact Transaction Service のインスタンスをシャットダウンするには、オプションの -immediate 引数を使用します。

メモ　アプリケーションの ORB_init() メソッドに -OTSexit_on_shutdown 引数が渡された場合は、このコマンドを使用して、アプリケーションプロセス内に埋め込まれた VisiTransact Transaction Service のインスタンスをシャットダウンできます。アプリケーションプロセスに埋め込まれた VisiTransact Transaction Service のインスタンスをシャットダウンする方法については、188 ページの「VisiTransact Transaction Service インスタンスが埋め込まれたアプリケーションの引数」を参照してください。

構文

```
prompt>vshutdown -help
prompt>vshutdown -type <object_type>
        [-name <object_name>]
        [-host <host_name>]
        [-immediate]
        [-noprompt]
```

サンプル

```
prompt>vshutdown -type ots -name myTxnSvc
```

引数

このコマンドでは、次の引数を使用できます。

引数	説明
-help	このコマンドの使用方法を表示します。 この引数を使用すると、ほかの引数は無視され、 使用方法だけが表示されます。
-type	有効なタイプは次のとおりです。 ots - VisiTransact Transaction Service rd - VisiTransact XA リソースディレクタ smcs - セッションマネージャ設定サーバー 型だけを指定すると、その型のすべてのサービスが表示され、それらをシャットダウンするかどうかをたずねるメッセージが表示されます。
-name <object_name>	シャットダウンするオブジェクトの名前。 デフォルトでは、 指定した型のオブジェクトが検索され、 それらをシャットダウンするかどうかをたずねるメッセージが表示されます。

引数	説明
-host <host_name>	シャットダウンするサービスがあるホストマシン。 デフォルトでは、特定の型と名前（指定した場合）のオブジェクトがネットワーク上で検索され、 それらをシャットダウンするかどうかをたずねるメッセージが表示されます。
-immediate	未処理のトランザクションを解決しないで、VisiTransact Transaction Service のインスタンスをすぐにシャットダウンします。
-noprompt	すべてのオブジェクトの型、 名前、 またはホストのリストが取得されたときに、それらのシャットダウンを確認するメッセージを表示しない場合は、この引数を使用します。

xa_resdir

このコマンドを使って XA リソースディレクタのインスタンスを起動します。VisiBroker コンソールを使ってデータベースにアクセスする接続プロファイルを作成済みである必要があります。

構文

```
prompt>xa-resdir -Dvbroker.sm.profileName=<profile>
        [-Dvbroker.sm.pstorePath=<path>]
        [-Dvbroker.sm.configName=<name>]
        [-Dvbroker.sm.connectionIdleTimeout=<seconds>]
```

サンプル

```
prompt>xa-resdir -Dvbroker.sm.profileName=quickstart
 -Dvbroker.sm.pstorePath=C:¥vbroker¥adm¥its¥session_manager
 -Dvbroker.sm.configName=athena_smcs
```

引数

このコマンドでは、次の引数を使用できます。

引数	説明
-Dvbroker.sm.profileName=<profile>	データベースとの接続を確立するために使用するセッションマネージャ接続プロファイルの名前を入力します。 必須です。
-Dvbroker.sm.pstorePath=<path>	永続的ストアファイルが存在するディレクトリのパスを指定します。 デフォルトでは、 永続的ストアファイルは <VBROKER_ADM>¥its¥session_manager にあります。
-Dvbroker.sm.configName=<name>	使用するセッションマネージャ設定サーバーの名前を指定します。 デフォルトでは、 セッションマネージャ設定サーバーに割り当てられた名前は <host>_smcs です。 ここで host は、 セッションマネージャプロファイルを作成したサーバーの名前です。
-Dvbroker.sm.connectionIdleTimeout=<seconds>	接続は、 アイドルでトランザクションに関連付けられていないまま、ここに指定した秒数が経過すると、セッションマネージャ ConnectionPool によって自動的に閉じられます。これを使用して、プール内の未使用の接続数を減らすことができます。 このパラメータのデフォルトは 300 秒です。

VisiTransact ユーティリティ

smconfigsetup

smconfigsetup ユーティリティを使用して、接続プロファイルを作成できます。このユーティリティを使用して、セッションマネージャで使用するプロファイルを作成するには、次の手順にしたがいます。太字の文字は、ユーザー入力を示します。smconfigsetup ユーティリティの終了時にプロファイルが作成されます。

セッションマネージャで使用するプロファイルを作成する

セッションマネージャで使用するプロファイルを作成するには、次の手順にしたがいます。

1 コマンドプロンプトに **smconfigsetup** と入力します。

    prompt>smconfigsetup

2 数字の 1 を入力してプロファイルを作成します。

    Do you wish to
    (0) Quit
    (1) Add a profile
    (2) List all profiles
    (3) List attributes of a profile
    (4) Copy a profile
    (5) Delete a profile
    (6) Create metadata files
    (7) Add pluggable data resources

    Enter the number of your selection: **1**

3 データベースの種類に対応する数字を入力します。

    2 known Session Manager implementations:
    (0) Oracle OCI 9i DirectConnect
    (1) Oracle OCI 9i XA

    Please enter the database type you are trying to create: **0**

4 接続プロファイル名を入力します。

    Please enter the name for the new profile: **quickstart**

5 データベース名を入力します。

    Attribute name "database_name"
    New value for attribute Database Name (default value <>) : **itso9idb**

6 ユーザー名を入力します。

    Attribute name "userid"
    New value for attribute User Name (default value <>) : **scott**

7 ユーザーのパスワードを入力します。

    Attribute name "password"
    New value for attribute Password (default value <>) : **tiger**

8 ネイティブハンドル型を入力します。

    Attribute name "native_handle_type"
    New value for attribute Native Handle Type
    (default value <ITSoracle9i_handles>) : **ITSoracle9i_handle**

9 0（ゼロ）を入力してユーティリティを終了します。

    Do you wish to
    (0) Quit
    (1) Add a profile

```
(2) List all profiles
(3) List attributes of a profile
(4) Copy a profile
(5) Delete a profile
(6) Create metadata files
(7) Add pluggable data resources

Enter the number of your selection: 0

Bye!
```

smconfigsetup ユーティリティの終了時にプロファイルが作成されます。

アプリケーションのコマンドライン引数

ORB_init() に、VisiTransact Transaction Service やアプリケーションコンポーネントの動作に影響を及ぼす引数を渡すことができます。以下の節で、各オプションについて説明します。

argc と argv を使用して、コマンドライン引数を ORB_init() に渡す

コマンドライン引数は、VisiBroker のコンポーネントとして、VisiBroker ORB の初期化呼び出し ORB_init() を介して VisiTransact のコンポーネントに渡されます。したがって、コマンドラインで指定された引数を特定のアプリケーションプロセスの VisiTransact オペレーションで利用するには、アプリケーションのメインプログラムから元の argc 引数と argv 引数を ORB_init() に渡す必要があります。たとえば、次のようにします。

```
int main(int argc, char * const* argv)
{
    try
    {
        // ORB の初期化
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
    }
...
}
```

ORB_init() 関数は、ORB の引数と VisiTransact の引数の両方を解析し、それらを argv ベクタから削除してから戻ります。

トランザクションを開始するアプリケーションの引数

デフォルトでは、Current::begin(), を使って初めてトランザクションを開始する際に、スマートエージェントを使って VisiTransact Transaction Service のインスタンスが検索されます。ここで説明する引数を使用して、使用する VisiTransact Transaction Service のインスタンスとトランザクションのタイムアウト値を指定できます。

これらの引数は、トランザクションサーバーを手動で起動する際に、コマンドラインで渡します。187 ページの「argc と argv を使用して、コマンドライン引数を ORB_init() に渡す」で説明するように、アプリケーションは、これらのコマンドライン入力引数を ORB_init() メソッドで処理します。

次の表では、トランザクションを開始するアプリケーションのコマンドラインから ORB_init() に渡すことができる引数について説明しています。

ORB_init() に渡すことができる引数	説明
-Dvbroker.ots.currentFactory	VisiTransact は、要求されたトランザクションサービスに対して指定された IOR（CosTransactions::TransactionFactory）を使用して、ネットワーク上で VisiTransact Transaction Service のインスタンスを探します。この引数により、VisiTransact は、スマートエージェント（osagent）を使用しなくても操作を実行できます。
-Dvbroker.ots.currentHost	スマートエージェントは、指定されたホストで、使用できる VisiTransact Transaction Service のインスタンスを探します。
-Dvbroker.ots.currentName	スマートエージェントは、ネットワーク上で、指定された VisiTransact Transaction Service のインスタンスを探します。
-Dvbroker.ots.currentTimeout	Current のトランザクションタイムアウト値を設定します。タイムアウトを過ぎてもトランザクションが存続している場合、そのトランザクションは自動的にロールバックされます。

たとえば、Accounting VisiTransact Transaction Service を使用する billing C++ トランザクションサーバーを起動するには、次のコマンドを使用します。

prompt>billing -Dvbroker.ots.currentName=Accounting

Accounting VisiTransact Transaction Service を使用し、タイムアウトまでの時間が 2400 秒である Billing トランザクションサーバーを起動するには、次のコマンドを使用します。

prompt>billing -Dvbroker.ots.currentName=Accounting -Dvbroker.ots.currentTimeout=2400

-Dvbroker.ots.currentHost と -Dvbroker.ots.currentName を組み合わせて指定すると、スマートエージェントは、指定されたホストで指定された VisiTransact Transaction Service のインスタンスを探します。-Dvbroker.ots.currentFactory を -Dvbroker.ots.currentHost または -Dvbroker.ots.currentName とともに指定すると、スマートエージェントは、VisiTransact Transaction Service インスタンスを IOR だけで検索します。ほかの引数は無視されます。

VisiTransact Transaction Service インスタンスが埋め込まれたアプリケーションの引数

ここで説明する引数を使用して、使用する VisiTransact Transaction Service のインスタンスを指定できます。また、VisiTransact Transaction Service の埋め込みインスタンスを終了する際に、アプリケーションプロセスを停止するかどうかも指定できます。

これらの引数は、トランザクションサーバーを手動で起動する際に、コマンドラインで渡します。187 ページの「argc と argv を使用して、コマンドライン引数を ORB_init() に渡す」で説明するように、アプリケーションは、これらのコマンドライン入力引数を ORB_init() メソッドで処理します。

次の表では、VisiTransact Transaction Service のインスタンスが埋め込まれたアプリケーションのコマンドラインから ORB_init() に渡すことができる引数について説明しています。

ORB_init() に渡すことができる引数	説明
-Dvbroker.ots.defaultTimeout=<seconds>	VisiTransact Transaction Service インスタンスのデフォルトのトランザクションタイムアウト値を設定します。 設定しないと、デフォルトの 600 秒になります。
-Dvbroker.ots.defaultMaxTimeout=<seconds>	VisiTransact Transaction Service インスタンスの最大のトランザクションタイムアウト値を設定します。 設定しないと、デフォルトの 3600 秒になります。
-Dvbroker.ots.name=<transaction_service_name>	スマート エージェント に VisiTransact Transaction Service のインターフェースを登録する際に使用されるインスタンス名を設定します。デフォルト は <host_name>_ots です。
-Dvbroker.ots.logDir=<directory_name>	ログとロガー情報を保存するディレクトリの名前を指定します。 指定しないと、デフォルトの <VBROKER_ADM>¥its¥<transaction_service_name>¥logger になります。
-Dvbroker.ots.exitOnShutdown	true に設定すると、 vshutdown または VisiBroker コンソールを使ってリモートに VisiTransact Transaction Service をシャットダウンする際に、 インプロセスの VisiTransact Transaction Service インスタンスが終了し、 アプリケーションプロセスが停止します。 これが設定されていないか、false に設定されている場合は、スマートエージェントに登録されている VisiTransact Transaction Service オブジェクトが非アクティブ化されますが、アプリケーションプロセスが停止することはありません。

セッションマネージャを使用するアプリケーションの引数

デフォルトでは、セッションマネージャ接続プロファイルが作成されたマシン（<host>_smcs）のセッションマネージャ設定サーバーが使用されます。デフォルトの永続的ストレージは、<VBROKER_ADM>¥its¥session_manager にあります。

これらの引数は、トランザクションサーバーを手動で起動する際に、コマンドラインで渡します。187 ページの「argc と argv を使用して、コマンドライン引数を ORB_init() に渡す」で説明するように、アプリケーションは、これらのコマンドライン入力引数を ORB_init() メソッドで処理します。

次の表では、セッションマネージャを使用するアプリケーションのコマンドラインから ORB_init() に渡すことができる引数について説明しています。

ORB_init() に渡すことができる引数	説明
-Dvbroker.sm.configName	セッションマネージャ設定サーバーの名前 (profileset)。デフォルトでは、この値は <host>_smcs です。ここで host は、セッションマネージャプロファイルを作成したサーバーの名前です。
-Dvbroker.sm.pstorePath	永続的ストアクラスが存在するディレクトリのパス。デフォルトでは、<VBROKER_ADM>¥its¥session_manager です。
-Dvbroker.sm.connectionIdleTimeout	接続は、アイドルでトランザクションに関連付けられていないまま、ここに指定した秒数が経過すると、セッションマネージャ ConnectionPool によって自動的に閉じられます。これを使用して、プール内の未使用の接続数を減らすことができます。このパラメータのデフォルトは 300 秒です。

環境変数

VisiBroker VisiTransact 用に次の環境変数を設定できます。

環境変数	説明
VBROKER_ADM	ITS 固有のファイルが保存されているディレクトリのパスを定義します。

第 23 章

エラーコード

この付録では、VisiTransact のエラーコードについて説明します。

VisiTransact の一般的なエラーコード

次の表に、VisiTransact の一般的なエラーコードを示します。

エラーコード	説明	考えられる原因	解決策
201	ファイルまたはディレクトリへのアクセスが拒否されました。	プロセスに、そのファイルやディレクトリにアクセスするために必要な許可がありません。	ファイルまたはディレクトリのアクセス許可を変更して、そのプロセスがアクセスできるようにします。
202	プロセスが要求されたファイルを開くことができません。	ファイルが格納されているディレクトリが異なります。 プロセスに、そのファイルにアクセスするための許可がありません。	ファイルが格納されているディレクトリを確認し、再試行します。 ファイルのアクセス許可を変更して、そのプロセスがアクセスできるようにします。
203	ファイルの読み取り中にエラーが発生しました。	プロセスに、そのファイルを読み取るための許可がありません。	ファイルのアクセス許可を変更して、そのプロセスでファイルを読み取ることができるようにします。

エラーコード	説明	考えられる原因	解決策
204	ファイルへの書き込み中にエラーが発生しました。	プロセスが持つ許可は読み取り専用です。ファイルに書き込むための許可がありません。 ストレージがいっぱいで、変更をファイルに書き込むための容量がシステムにありません。	ファイルのアクセス許可を変更して、そのプロセスでファイルに書き込むことができるようにします。 ストレージをクリーンアップし、再試行します。
801	指定された型のオブジェクトをリストする際にエラーが発生しました。	ロケーションサービスがありません。 スマートエージェントが実行されていません。 プロセスに通信の問題が発生しています。	ロケーションサービスが有効であることを確認します。詳細については、VisiBroker ORB のマニュアルを参照してください。 **osagent** コマンドを使ってスマートエージェントを開始します。詳細については、VisiBroker ORB のマニュアルを参照してください。 必要なプロセスがすべて実行中であり、すべてのマシンが稼動中であることを確認し、再試行します。

VisiTransact トランザクションサービスのエラーコード

次の表に、VisiTransact トランザクションサービスのエラーコードを示します。

エラーコード	説明	考えられる原因	解決策
4000	VisiTransact Transaction Service のインスタンスが正常に開始されました。	これは情報メッセージです。	必要な操作はありません。
4001	VisiTransact Transaction Service のインスタンスが要求に応じてシャットダウンされます。	管理者またはほかのユーザーが、**vshutdown** コマンド、**〔Ctrl〕+〔C〕** キー、**kill** コマンドのいずれかを使用して、VisiTransact Transaction Service のインスタンスをシャットダウンしました。	必要な操作はありません。
4002	VisiTransact Transaction Service のインスタンスをシャットダウンする準備が完了しましたが、未処理のトランザクションが完了段階に入るのを待っています。	`-immediate` 引数なしで、VisiTransact Transaction Service のインスタンスをシャットダウンする要求が発行されました。インスタンスは、未処理のトランザクションが完了段階に入ってからシャットダウンします。	必要な操作はありません。 未処理のトランザクションが完了段階に入るのを待たずに VisiTransact Transaction Service のインスタンスをシャットダウンするには、`-immediate` 引数を指定して **vshutdown** コマンドを発行します。
4003	未処理のトランザクションが完了段階に入るのを待たずに、VisiTransact Transaction Service のインスタンスをシャットダウンします。	`-immediate` 引数付きで、VisiTransact Transaction Service のインスタンスをシャットダウンする要求が発行されました。インスタンスは、未処理のトランザクションが完了段階に入るのを待たずにシャットダウンします。	必要な操作はありません。 未処理のトランザクションが完了段階に入るのを待ってから VisiTransact Transaction Service のインスタンスをシャットダウンするには、`-immediate` 引数を指定しないで **vshutdown** コマンドを発行します。

エラーコード	説明	考えられる原因	解決策
4004	Resource によって HeuristicHazard 例外が生成されました。この例外の詳細については、ヒューリスティックログファイルの出力を参照してください。	Resource はヒューリスティックな決定を行いましたが、少なくとも 1 つの関連する更新の結果を認識していません。	データの整合性が失われている可能性があります。ヒューリスティックログでエラーを調査し、データベース管理者にトランザクション ID を報告してください。データベース管理者は、このエラーを Resource で確認し、問題を手動で修正する必要があります。
4005	Resource によって HeuristicCommit 例外が生成されました。この例外の詳細については、ヒューリスティックログファイルの出力を参照してください。	Resource はヒューリスティックな決定を行い、すべての関連する更新をコミットしました。	データの整合性が失われている可能性があります。ヒューリスティックログでエラーを調査し、データベース管理者にトランザクション ID を報告してください。データベース管理者は、このエラーを Resource で確認し、問題を手動で修正する必要があります。
4006	Resource によって HeuristicRollback 例外が生成されました。この例外の詳細については、ヒューリスティックログファイルの出力を参照してください。	Resource はヒューリスティックな決定を行い、すべての関連する更新をロールバックしました。	データの整合性が失われている可能性があります。ヒューリスティックログでエラーを調査し、データベース管理者にトランザクション ID を報告してください。データベース管理者は、このエラーを Resource で確認し、問題を手動で修正する必要があります。
4007	Resource によって HeuristicMixed 例外が生成されました。この例外の詳細については、ヒューリスティックログファイルの出力を参照してください。	Resource がトランザクションの結果とは異なるヒューリスティックな決定を行いました。一部の更新がコミットされ、それ以外はロールバックされました。	データの整合性が失われている可能性があります。ヒューリスティックログでエラーを調査し、データベース管理者にトランザクション ID を報告してください。データベース管理者は、このエラーを Resource で確認し、問題を手動で修正する必要があります。
4008	特定の警告（メッセージに表示）に対して、コールバック中に例外がキャッチされて無視されました。	このメッセージは、システムリソースが不足した場合など、さまざまな理由で生成されます。	このメッセージは無視してください。
4009	内部アプリケーションエラーが発生しました。	不明な例外が原因で、複数の VisiTransact コンポーネントによって使用されている VisiTransact Transaction Manager の内部モジュールを初期化できませんでした。	VisiBroker のテクニカルサポートにお問い合わせください。
4010	メッセージに記述されている内部アプリケーションエラーが発生しました。	メッセージに表示されている例外が原因で、複数の VisiTransact コンポーネントによって使用されている VisiTransact Transaction Manager の内部モジュールを初期化できませんでした。	VisiBroker のテクニカルサポートにお問い合わせください。
4011	メッセージに表示されている初期化引数の解析中に例外が生成されました。	VisiTransact コマンドの実行時に、誤ったコマンドライン引数が入力されました。	コマンドライン引数を確認し、再試行します。181 ページの「コマンド、ユーティリティ、引数、および環境変数」を参照してください。
4012	一部の初期化引数の解析中に例外が生成されましたが、どの引数が不正であるかは不明です。	VisiTransact コマンドの実行時に、誤ったコマンドライン引数が入力されました。	コマンドライン引数を確認し、再試行します。181 ページの「コマンド、ユーティリティ、引数、および環境変数」を参照してください。

エラーコード	説明	考えられる原因	解決策
4014	VisiTransact Transaction Service のインスタンスを開始する際に、メッセージに示されている初期化エラーが発生しました。	誤った設定ファイルが使用されたか、初期化パラメータに不正な値が入力されました。 内部アプリケーションエラーが発生しました。	正しい設定ファイルを使用しており、初期化パラメータに正しい値を入力していることを確認します。 VisiBroker のテクニカルサポートにお問い合わせください。
4015	実行中の VisiTransact Transaction Service インスタンスで、実行時例外が生成されました。	内部アプリケーションエラーが発生しました。	VisiBroker のテクニカルサポートにお問い合わせください。
4016	デフォルトのトランザクションタイムアウト値が、その最大値に変更されました。	デフォルトのトランザクションタイムアウト値が、その最大値を超えています。	VisiTransact Transaction Service のインスタンスを開始する際に、アプリケーションとコマンドライン引数の間でタイムアウトの設定が調整されていることを確認します。
4017	デフォルトのトランザクションタイムアウトに無効な値が指定されました。タイムアウト値が 600 秒にリセットされました。	デフォルトのタイムアウト値に 0 または負の値が設定されました。デフォルトのタイムアウト値は 1 秒以上に設定する必要があります。	デフォルトのタイムアウト値を設定する際は、1 秒以上の値を指定してください。推奨値は 600 秒です。
4018	トランザクションの完了中に、VisiTransact Transaction Service によって予期しない例外が生成されました。 VisiTransact Transaction Service によってトランザクションの完了が再試行されます。	内部アプリケーションエラーが発生しました。	VisiBroker のテクニカルサポートにお問い合わせください。
4019	トランザクションの完了中に、VisiTransact Transaction Service によって予期しない CORBA 例外が生成されました。 VisiTransact Transaction Service によってトランザクションの完了が再試行されます。	内部アプリケーションエラーが発生しました。	VisiBroker のテクニカルサポートにお問い合わせください。

セッションマネージャのエラーコード

次の表に、セッションマネージャのエラーコードを示します。

エラーコード	説明	考えられる原因	解決策
6001	セッションマネージャは必要なメモリを割り当てることができませんでした。	メモリが不足しています。	スワップ領域を増やし、不要なプロセスをシャットダウンして、使用できるメモリを増やします。
6002	プロファイルが複数のセッションマネージャのインプリメンテーションを使用しているため、プロファイルに互換性がありません。	アプリケーションは、1 つのプロセスで、OCI 向けの XA セッションマネージャを指定するプロファイルと、Oracle OCI 向けの DirectConnect セッションマネージャを指定するプロファイルを使用しようとしています。	1 つのプロセスのセッションマネージャで使用される Oracle 接続の種類は、必ず 1 つだけにしてください。
6003	セッションマネージャは必要なライブラリをロードできませんでした。	PATH または LIBRARY_PATH 環境変数が正しく設定されていないか、関連ライブラリがありません。	環境変数が正しく設定されていることを確認します。それでも問題が解決しない場合は、VisiBroker VisiTransact と VisiBroker ORB を再インストールする必要があります。
6004	セッションマネージャは、ロードされたライブラリで必要な関数シンボルを見つけることができませんでした。	必要なデータベースライブラリの一部が見つかりませんでした。	PATH（Windows NT）および LD_LIBRARY_PATH（Solaris）環境変数に、使用するデータベースライブラリのパスが含まれていることを確認します。
			データベースクライアントライブラリが使用できることを確認します。
		ロードされたライブラリファイルが破損しています。	VisiBroker ORB および VisiBroker VisiTransact ソフトウェアを再インストールしてください。
6005	セッションマネージャはデータベースへの接続を開くことができませんでした。	データベースを使用できません。	データベースが正常に起動していることを確認します。
		無効なユーザー名またはパスワードを使用しています。	セッションマネージャ接続プロファイルをチェックして、データベース名、ユーザー名、およびパスワードが正しく入力されていることを確認します。
		データベースソフトウェアが正しくインストールまたは設定されていません。	データベースソフトウェアが正しくインストールおよび設定されていることを確認します。
6006	セッションマネージャによるデータベース接続の初期化時にエラーが発生しました。	データベースソフトウェアのバージョンがセッションマネージャと互換性がありません。	167 ページの「XA Session Manager for Oracle OCI、version 9i Client」と 175 ページの「DirectConnect Session Manager for Oracle OCI、version 9i Client」を参照して、サポートされているバージョンのデータベースソフトウェアを使用していることを確認します。

エラーコード	説明	考えられる原因	解決策
6007	セッションマネージャによるデータベース接続の割り当て時に、接続関連付けエラーが発生しました。	データベースサーバーを使用できないか、設定が不正です。	システム管理者に問い合わせて、データベースサーバーが正常に起動していることを確認してください。データベース設定の要件については、167 ページの「XA Session Manager for Oracle OCI、version 9i Client」と 175 ページの「DirectConnect Session Manager for Oracle OCI、version 9i Client」を参照してください。
		非 ITS 管理トランザクションの使用時に、無効な Coordinator を使用した可能性があります。	`getConnection()` メソッドの呼び出し時に、有効な Coordinator を使用していることを確認します。
6008	最大接続数の制限を超えました。	最大接続数を超えています。	このデータベースへの最大接続数の制限を引き上げます。
		データベースサーバーの負荷が高すぎます。	データベースサーバーの負荷を削減します。
6009	このトランザクションのデータベース接続は、すでに別のスレッドが保持しています。	DirectConnect セッションマネージャを使用して、このトランザクションのデータベース接続を複数回試行しました。	DirectConnect セッションマネージャを使用している場合は、トランザクションに対して一度に 1 つの接続オブジェクトだけを使用します。
6010	セッションマネージャは XA ライブラリからデータベースのネイティブ接続ハンドルを取得できませんでした。	データベースソフトウェアのバージョンがセッションマネージャと互換性がありません。	167 ページの「XA Session Manager for Oracle OCI、version 9i Client」と 175 ページの「DirectConnect Session Manager for Oracle OCI、version 9i Client」を参照して、サポートされているバージョンのデータベースソフトウェアを使用していることを確認します。
6011	接続を取得したスレッドと、現在その接続を処理しようとしているスレッドとが異なります。	接続をスレッド間で共有しようとしています。	制限の詳細については、167 ページの「XA Session Manager for Oracle OCI、version 9i Client」と 175 ページの「DirectConnect Session Manager for Oracle OCI、version 9i Client」を参照してください。
6012	セッションマネージャは、接続プロファイルで指定されたリソースディレクタを登録できませんでした。	リソースディレクタが指定された OSAGENT_PORT で実行されていません。	リソースディレクタのインスタンスが指定された OSAGENT_PORT で実行されていることを確認します。111 ページの「XA リソースディレクタの起動」を参照してください。
		設定プロファイルに指定されているリソースディレクタの名前が不正です。	接続プロファイルの［Resource Director Name］フィールドに入力した名前が正しいことを確認します。手順については、107 ページの「セッションマネージャを使用した VisiTransact とデータベースの 統合」を参照してください。
6013	トランザクションコンテキストが接続の要求時に確立されませんでした。	`getConnection()` を呼び出す前に `begin()` を呼び出していなかったため、トランザクションコンテキストが存在しませんでした。	プログラムで `getConnection()` を呼び出す前に、`begin()` を呼び出してあることを確認します。
6014	`SMconnection_idle_timeout` 属性に無効な値が指定されました。	`SMconnection_idle_timeout` に負の値が指定されました。	この属性に正の値を指定します。

エラーコード	説明	考えられる原因	解決策
6015	セッションマネージャは、明示的に使用されている Coordinator の有効性を確認できませんでした。	明示的な伝達を使ってトランザクションを処理しており、すでにロールバックされたトランザクションの Coordinator オブジェクトをセッションマネージャに渡しました。	明示的な伝達を使ってトランザクション処理するときは、渡す Coordinator オブジェクトが有効であることを確認する必要があります。
6016	接続の解放プロセス時にエラーが発生しました。	セッションマネージャによる接続の解放時に、おそらくデータベースサーバーのクラッシュにより、接続が無効になりました。	データベースサーバープロセスが実行されていることを確認します。
6017	セッションマネージャは、データベースから要求された切断を実行できませんでした。	セッションマネージャによる接続の解放時に、おそらくデータベースサーバーのクラッシュにより、接続が無効になりました。	データベースサーバープロセスが実行されていることを確認します。
6018	接続プロファイルに未知の属性がありました。	このセッションマネージャのバージョンと互換性がない古い接続プロファイルを使用していると思われます。	VisiBroker コンソールを使って接続プロファイルを再作成します。、107 ページの「セッションマネージャを使用した VisiTransact とデータベースの 統合」を参照してください。
		接続プロファイルが損傷しています。	VisiBroker コンソールを使って接続プロファイルを再作成します。 先の説明を参照してください。
6019	接続の取得に必要なプロファイル属性が設定されていません。	接続プロファイルには、この接続属性の値が設定されていません。	VisiBroker コンソールを使用して、 接続プロファイルの設定を確認します。
		接続プロファイルが損傷しています。	VisiBroker コンソールを使って接続プロファイルを再作成します。
6020	`getInfo()` メソッドに渡された引数の infotype が無効です。	プログラムの `getInfo()` 呼び出しで、不正な infotype を渡しています。	`getInfo()` メソッドが受け入れる共通の infotype については、「VisiBroker for C++ API Reference」を参照してください。 個々のデータベースで許容される情報のタイプの詳細については、167 ページの「XA Session Manager for Oracle OCI、 version 9i Client」と 175 ページの「DirectConnect Session Manager for Oracle OCI、 version 9i Client」を参照してください。
6021	`hold()` オペレーションはサポートされていません。	`hold()` オペレーションをサポートしていない接続に対して、アプリケーションが `hold()` オペレーションを呼び出しました。	`isSupported()` を使用して、このオペレーションが有効な接続であるかどうかを判定します。
6022	メソッドの呼び出しに引数として渡した接続プロファイル名が無効です。	アプリケーションのメソッド呼び出しで、 無効な接続プロファイル名（多くの場合 null または長さが 0 の文字列）を渡しています。	メソッドが有効な接続プロファイル名を使用していることを確認します。また、VisiBroker コンソールを使用して、接続プロファイルが実際に設定されていることを確認します。
6024	接続を使用できる最大待機時間を超えたため、セッションマネージャが接続を取得できませんでした。	現在のスレッド（接続を要求しているスレッド）と別のスレッド（要求された接続を保持するスレッド）の間にデッドロックが発生しました。デッドロックを解決するために、 現在のスレッドがキューから削除されました。	デッドロックを避けるようにコードを最適化するには、成功するまで `getConnection()` を呼び出すように再試行ループを実装します。詳細については、セッションマネージャを使用したデータアクセスを参照してください。

エラーコード	説明	考えられる原因	解決策
6025	コマンドラインに無効な引数が指定されました。	コマンドライン引数に入力ミスがあります。	コマンドラインのすべての引数のスペルを確認します。「コマンド、ユーティリティ、引数、および環境変数」を参照してください。
6026	コマンドライン引数に無効な値が指定されました。	コマンドライン引数に無効な値が指定されました。	コマンドラインのすべての引数の値が有効であることを確認します。「コマンド、ユーティリティ、引数、および環境変数」を参照してください。
6032	このリソースのコミットが失敗しました。	コミット時に不正なデータベース操作が行われたため、データベースがこの操作を拒否しました。	個々のデータベースのセッションマネージャのトラブルシューティング情報の詳細については、167 ページの「XA Session Manager for Oracle OCI、version 9i Client」と 175 ページの「DirectConnect Session Manager for Oracle OCI、version 9i Client」を参照してください。
6033	このリソースのロールバックが失敗しました。	ロールバック時に不正なデータベース操作が行われたため、データベースがこの操作を拒否しました。	個々のデータベースのセッションマネージャのトラブルシューティング情報の詳細については、167 ページの「XA Session Manager for Oracle OCI、version 9i Client」と 175 ページの「DirectConnect Session Manager for Oracle OCI、version 9i Client」を参照してください。
6034	表示されたエラーメッセージは、使用しているデータベースに固有のメッセージです。	データベースに対するネイティブ接続呼び出しまたはネイティブトランザクション管理呼び出しが失敗しました。	個々のデータベースのセッションマネージャのトラブルシューティング情報の詳細については、「167 ページの「XA Session Manager for Oracle OCI、version 9i Client」と 175 ページの「DirectConnect Session Manager for Oracle OCI、version 9i Client」を参照してください。
6035	内部エラーが発生しました。	セッションマネージャで内部アプリケーションエラーが発生しました。	VisiBroker のテクニカルサポートにお問い合わせください。
6040	セッションマネージャは、VisiTransact Transaction Service のインスタンスにリソースを登録できませんでした。	VisiTransact Transaction Service のインスタンスが指定された OSAGENT_PORT で実行されていません。	VisiTransact Transaction Service のインスタンスが指定された OSAGENT_PORT で実行されていることを確認します。
6042	指定されたリソースディレクタを使用できません。	リソースディレクタのインスタンスが指定された OSAGENT_PORT で実行されていません。	リソースディレクタのインスタンスが指定された OSAGENT_PORT で実行されていることを確認します。
6043	回復中にエラーが発生しました。	データベースの XA リソースマネージャが実行されていません。	データベースの XA リソースマネージャが実行されていて使用できることを確認します。
		データベースの XA リソースマネージャにエラーが発生しました。	データベースのエラーログで、発生したエラーの詳細な報告を確認します。
6046	セッションマネージャの XA 呼び出しにより、メッセージに示されたエラーコードが返されました。	データベースエラーが発生しました。	トラブルシューティングのヒントについては、201 ページの「問題の判定」を参照してください。

エラーコード	説明	考えられる原因	解決策
6047	同じ LRM 名を持つ 2 つの接続に異なるユーザー名とパスワードが使用されていることを Sybase CTLib のセッションマネージャが検出しました。	同じサーバープロセスの 2 つの接続プロファイルが同じデータベース名を共有していますが、パスワードが異なります。	どちらかのプロファイル用に、異なる LRM を使用する新しい接続プロファイルを作成します。必要に応じて新しい接続プロファイルを使用するように、サーバープロセスを変更します。
6048	接続プロファイルの形式を認識できません。	永続的ストレージファイルが保存されているディレクトリに外部ファイルが書き込まれたか、接続プロファイルが損傷している可能性があります。	VisiBroker コンソールを使って接続プロファイルを再作成します。
6049	セッションマネージャは、接続プロファイルを開くことができませんでした。	ファイルアクセス許可の問題が発生しています。	ファイルに正しいアクセス許可があることを確認します。
6050	接続プロファイルの読み取り時にエラーが発生しました。	永続的ストレージファイルが保存されているディレクトリに外部ファイルが書き込まれたか、接続プロファイルが損傷している可能性があります。	VisiBroker コンソールを使って接続プロファイルを再作成します。
6051	接続プロファイルへの書き込み時にエラーが発生しました。	ファイルまたはディレクトリに正しいアクセス許可がありません。	ファイルとディレクトリに正しいアクセス許可があることを確認します。
		ディスクがいっぱいなので、セッションマネージャは接続プロファイルを保存できません。	セッションマネージャが接続プロファイルを保存できるように、ディスクスペースを解放します。
		接続プロファイルへの不正なパスが指定されました。	正しいパスを指定します。
6052	指定されたパスが不正です。	接続プロファイルへの不正なパスが指定されました。	正しいパスを指定します。
6053	指定された接続プロファイルはすでに存在します。	すでに使用されている名前で接続プロファイルを作成しようとしました。	この接続プロファイルに別の名前を付けるか、別の接続プロファイルを削除します。
6054	指定された接続プロファイルは存在しません。	接続プロファイル名に入力ミスがあります。	接続プロファイル名を正しく指定していることを確認します。
6055	接続プロファイルが削除されませんでした。	ファイルまたはディレクトリに正しいアクセス許可がありません。	ファイルとディレクトリに正しいアクセス許可があることを確認します。
6056	XA リソースディレクタに、DirectConnect（非 XA）セッションマネージャ接続プロファイルが指定されました。	DirectConnect セッションマネージャ用に作成された接続プロファイルを使用して、XA リソースディレクタを起動しようとしました。	XA セッションマネージャ用に作成された接続プロファイルを使用して、XA リソースディレクトリを起動します。
6058	`hold()` 呼び出しがタイムアウトになりました。接続を解放する必要があります。	`hold()` 呼び出しに指定されたタイムアウト値の期限を超えたため、接続オブジェクトは使用できません。	この接続に対して `release()` を呼び出し、必要な場合は `getConnection()` を再び呼び出します。

VisiTransact トランザクションログのエラーコード

次の表に、VisiTransact トランザクションログのエラーコードを示します。

エラーコード	説明	考えられる原因	解決策
8001	ロガーモジュールに内部エラーが発生しました。 詳細については、メッセージログを参照してください。	メッセージログのエラーテキストを参照してください。	メッセージログに表示されているエラーテキストからエラーを解決できない場合は、VisiBroker のテクニカルサポートにお問い合わせください。

第 24 章

問題の判定

この付録では、問題の原因を判定する方法について説明します。主に開発者固有の問題を扱い、デプロイメントよりも開発に重点を置きます。

一般的な方法

発生している可能性がある問題を調べるには、まず **vbroker¥adm¥its** にあるホストのメッセージログを参照します。

トランザクションの問題の処理

トランザクションと VisiBroker VisiTransact を使用するアプリケーションで発生する可能性がある典型的な問題は、次のとおりです。

- **トランザクションがタイムアウトになる。**タイムアウト期間に関しては、いくつかの状況が考えられます。トランザクションオリジネータが `commit()` を発行する前にトランザクションがタイムアウトになってロールバックしたり、リソースを登録しようとしたときに `CORBA::OBJECT_NOT_EXIST` 例外が発生することがあります。この問題が発生した場合は、設定されているタイムアウト時間の長さが十分であることを確認します。
- **VisiTransact Transaction Service が消失する。**VisiTransact Transaction Service インスタンスが再起動されたり、トランザクションの処理中に失敗した場合、VisiTransact メソッドを呼び出すと、`CORBA::NO_IMPLEMENT` 例外が一時的に発生することがあります。
- **リソースディレクタを使用できない。**セッションマネージャがリソースマネージャへの新しい接続を取得しようとしたときに XA リソースディレクタを見つけることができないと、`VISSessionManager::Error` 例外が発生します。さらに、参加しているリソースディレクタを準備時に使用できない場合は、VisiTransact Transaction Service がトランザクションをロールバックします。これは、ほかのリソースがコミット時に使用できない場合に発生する動作と同じです。
- **`CosTransactions::NoTransaction` 例外が発生する。**この例外は、トランザクションコンテキストがない場合に発生します。これは、アプリケーションが最初にトランザクションを開始することなく、接続しようとしたことを意味します。
- **セッションマネージャ設定ファイルを使用できない。**誤った接続プロファイルを要求している可能性があります。

索引

記号

... 省略符 4
[ ] ブラケット 4
| 縦線 4

数字

1 フェーズコミットのまとめ 86
2 フェーズコミットのまとめ 86
2PC のまとめ 86

B

Borland Web サイト 4, 5
Borland 開発者サポート、連絡 4
Borland テクニカルサポート、連絡 4

C

C++ VisiTransact アプリケーション 53
C++ サンプル 19
    bank オブジェクトの記述 29
    bank サーバーの記述 27
    IDL 22
    ORB を初期化します。23
    概要 19
    コミットとロールバック 26
    実行 35
    スマートエージェント 35
    トランザクションオブジェクトの記述 31
    トランザクションオリジネータ 23
    トランザクションの開始 24
    バインディング 23
    ビルド 34
    ファイル 20
    メソッドの呼び出し 26
    目的 21
    要件 21
    リファレンスの取得 25
    例外処理 27
C++ によるアプリケーション 53
    embedding 54
    スタンドアロン 53
C++ によるサンプル 19
    bank オブジェクトの記述 29
    bank サーバーの記述 27
    IDL 22
    ORB を初期化します。23
    概要 19
    コミットとロールバック 26
    実行 35
    スマートエージェント 35
    トランザクションオブジェクトの記述 31
    トランザクションオリジネータ 23
    トランザクションの開始 24
    バインディング 23
    ビルド 34
    ファイル 20
    メソッドの呼び出し 26
    目的 21
    要件 21
    リファレンスの取得 25
    例外処理 27
checked behavior、実行 76
commit 85
Connection オブジェクト
    取得 121
ConnectionPool リファレンス
    取得 120
    使い方 121
contexts
    明示的なトランザクション 121
control オブジェクト 69
CORBA サービス仕様 11
CORBA 準拠 10
CORBA トランザクションサービス 14
CORBA の概要 14
Current
    インターフェース 59
    使い方 57, 58
    明示的なコンテキストの取得 72
    リファレンスの取得 59
Current オブジェクトリファレンスの取得 59
Current に対する拡張 65
Current の拡張機能 65
Current の使用 57, 58

D

DirectConnect 109
    Oracle OCI とセッションマネージャ 175
DirectConnect セッションマネージャ 104, 175
DirectConnect と XA の共存 106
DirectConnect に関する問題 129
DirectConnect の制約 106

H

heuristic.log ファイル 90

I

init.ora
    Oracle DirectConnect の 177
    Oracle XA の 169
init.ora パラメータ 169
InvocationPolicy インターフェース 46
ITSDataConnection クラス 137

N

NonTxTargetPolicy インターフェース 46
NoTransaction 例外 201

O

OAD、XA リソースディレクタを登録 112
OCI 167
OMG 拡張機能 11
ORA-01017 179

ORA-12154 173, 179
Oracle
Oracle XA でのプログラミングの制限事項 171
Oracle XA のデータベース設定 169
Oracle と XA のインストール要件 169
XA セッションマネージャの使用 167
XA のインストールに関する問題 169
XA のクライアント要件 168
XA のサーバー要件 168
XA のソフトウェア要件 168
XA のトラブルシューティング 172
XA の必須環境変数 170
設定
XA のに関する問題 169
Oracle Call Interface 167
Oracle OCI
DirectConnect のエラーメッセージ 179
DirectConnect のクライアント要件 176
DirectConnectXA のサーバー要件 176
DirectConnectXA のソフトウェア要件 176
XA で許可されていない OCI 呼び出し 171
XA のエラーメッセージ 173
XA のソフトウェア要件 168
エラーメッセージ 172, 178
ヒューリスティックな完了の強制 173
Oracle XA
データベース設定 169
Oracle XA セッションマネージャ 167
Oracle データベースの統合 8
Oracle の要件
Oracle と XA 168
Oracle7
DirectConnect のソフトウェア要件 176
Oracle9i
DirectConnect での Oracle9i のインストール要件 176
DirectConnect のインストールに関する問題 176
DirectConnect のデータベース設定 177
DirectConnect のトラブルシューティング 178
DirectConnect の必須環境変数 177
Oracle DirectConnect でのプログラミングの制限事項 178
XA のトラブルシューティング 178
設定問題
と DirectConnect 176
OTS
起動 142
ots 181, 182
OTS の例外 80
OTS ポリシーインターフェース 46

P

PDF マニュアル 3

S

smconfig_server 117, 142, 181, 183
smconfigsetup 181, 186

T

Terminator インターフェース 72
Terminator によるコミット 72
Terminator によるロールバック 72
Terminator、コミット 72
Terminator、ロールバック 72
timeout
Oracle OCI と XA のトランザクションタイムアウト 173
timeouts 201
コンソール
141, 143
TransactionFactory 68

V

vbconsole 181, 182
VisiBroker ORB 9
VisiBroker コンソール 9, 141
VisiBroker の概要 1
VisiTransact
CORBA 準拠 10
アーキテクチャ 8
監視 10
起動 142
機能 10
基本 7
データベースの統合 8
トランザクションサービス 8
VisiTransact の埋め込みインスタンス 55
vshutdown 181, 184

W

Web トランザクション 53
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

X

XA 108
Oracle トランザクションの調整 167
XA セッションマネージャ 167
XA と DirectConnect の共存 106
XA に関する問題 129
XA の意味 108, 109
XA のパフォーマンスチューニング 114
XA パフォーマンス 114
XA パフォーマンスのチューニング 114
XA プロトコル 103
XA リソースディレクタ 103
OAD に登録 112
起動 111
シャットダウン 112
接続プロファイル 112
使い方 111
デプロイメント 111
xa_resdir 181, 185
xa_trc ファイル
Oracle OCI と XA 172

あ

アプリケーションの統合 52

い

移行 97
インスタンス、検索 62
インストール
  Oracle DirectConnect の問題 176
  Oracle XA の問題 169
  Oracle の要件 169
  Oracle9i の要件 176
インターフェース
  ネイティブハンドル取得 138
インターフェース、Current 59
インターフェースの定義 137

え

永続的ストア 114
永続的ストア、共有ファイルシステム 116
永続的ストア、デプロイメント 116
永続的ストアファイル 113
エラーコード 191
  セッションマネージャ 195
  トランザクションサービス 192
  トランザクションログ 200
エラーメッセージ
  Oracle OCI と DirectConnect 179
  Oracle OCI と XA 173
  Oracle OCI トレースファイル 172
  メッセージログ 172, 178

お

オンラインヘルプトピック、アクセス 3

か

下位互換性 97
開発者サポート、連絡 4
回復、トランザクション 104
回復に関する問題 129, 130
各ノード上の永続的ストア 116
環境変数 181, 190
  Oracle と XA で必須 170
  Oracle9i と DirectConnect で必須 177
完了 75
  参加 82
  実行 75
  理解 81
  リソースオブジェクトによる調整 81
完了、ヒューリスティック 78
完了に関する問題 129, 130
概要 1
  CORBA 14
  トランザクション処理 13

き

記号
  省略符 ... 4
  縦線 | 4
  ブラケット [ ] 4
切り替え
  DirectConnect から XA へ 130
  XA から DirectConnect へ 130

く

クライアント側、デプロイメント 112
クライアントの動作
  ポリシーインターフェース 47
クライアントの要件
  Oracle7 と DirectConnect 176
クラス
  ITSDataConnection 137

こ

公開標準のトランザクション処理 11
コード
  error 191
コピーされた永続的ストア 116
コマンド 181
  概要 181
コマンド、規約 4
コマンドライン引数 187
コミットの実行 85
コンソール 141
  概要 141
  起動 142
  セッションマネージャ 142
  トランザクションサービスのインスタンスの検索 143
セクション 141, 143
コンテキスト
  複数トランザクション 61
コンテキスト、Current からの明示的なコンテキストの取得 72
コンテキスト管理 51
  直接的と間接的 49
互換性 97

さ

サーバーの動作
  ポリシーインターフェース 46
サーバーの要件
  Oracle9i と DirectConnect 176
  XA セッションマネージャと Oracle 168
サポート、連絡 4

し

障害からの回復 87
障害の回復 87
準備、リソース 84
状態、トランザクション、取得 73
情報、取得 65, 73

す

スレッド
  複数トランザクション 61
スレッド、複数 61
スレッド管理 102
スレッドの要件 122

せ

セッションマネージャ 107

Oracle XA の接続プロファイル属性 170, 177
概要 99
情報の取得 126
使い方 119
定義 100
トラブルシューティング 201
のまとめ 120
引数 189
セッションマネージャセキュリティ 117
セッションマネージャ設定サーバー 142
セッションマネージャ設定サーバーへのアクセス 148
セッションマネージャ接続プロファイル 147
セッションマネージャのパフォーマンス 114
セッションマネージャのパフォーマンスチューニング 114
セッションマネージャのパフォーマンスのチューニング 114
セッションマネージャプロセス、起動 113
セッションマネージャプロファイルセット 147
設定
Oracle DirectConnect のデータベース設定 177
Oracle DirectConnect の問題 176
Oracle XA の問題 169
Oracle と XA 169
設定サーバー 117
接続
解放 123
接続、解放 102
接続、設定 101
接続、トランザクションとの関連付け 101
接続管理 133
接続の解放 102
接続のプール 102
接続の割り当て解除 124
接続ハンドル
ネイティブ 122
接続プール 102
接続プールの最適化 122
最適化 122
接続プロファイル 101, 136, 147
XA リソースディレクタ 112
更新 149
削除 149
作成 148
設定 148
フィルタリング 149
編集 149
接続プロファイル、変更 110
接続プロファイルセット 110

そ

ソフトウェアの更新 5
ソフトウェア要件
Oracle DirectConnectXA のクライアント要件 176
Oracle DirectConnectXA のサーバー要件 176
Oracle XA のクライアント 168
Oracle XA のサーバー 168
Oracle7 と DirectConnect 176
属性
Oracle XA のセッションマネージャ接続プロファイル 170, 177
表示 125

つ

通知、ヒューリスティック情報の有効化 79

て

テクニカルサポート、連絡 4
ディレクトリ構造 114
データアクセス障害
Oracle OCI と XA 172
トランザクションタイムアウト 173
未確定トランザクションによるロック 172
データベース準備 109
データベース設定
Oracle DirectConnect 用 177
Oracle と XA 169
データベース接続
開く 100
データベースと VisiTransact の統合 99
データベースと XA の統合 108
データベースの統合 8, 107, 109
データベースの問題
DirectConnect セッションマネージャと Oracle OCI 175
Oracle Call Interface（OCI）167
データベースリソース
プラグイン可能 131
デプロイメントに関する問題 105
伝達 51
暗黙的と明示的 50
伝達、明示的 70
伝達、明示的から暗黙的に 71

と

統合の準備 119
統合の例 128
統合例 128
登録、リソース 83
トラブルシューティング 201
DirectConnect セッションマネージャと Oracle OCI 178
Oracle OCI と DirectConnect のエラーメッセージ 179
Oracle OCI トレースファイル 172
Oracle XA の分散更新の問題 172
Oracle9i と DirectConnect セッションマネージャ 178
VisiTransact メッセージログ 178
XA セッションマネージャと Oracle 172
XA セッションマネージャと Oracle OCI 172, 178
XA セッションマネージャと Oracle9i 178
XA の Oracle OCI エラーメッセージ 173
データアクセス障害 172
トランザクションタイムアウト 173
ヒューリスティックな完了の強制 173
未確定トランザクションによるロック 172
メッセージログ 172
トランザクション
VisiTransact の基礎 7
web 53
アプリケーションの統合 52
構築方法 49
分散 13
マルチスレッド 52
モデル 15
トランザクション、確認 63

トランザクション、作成 67
トランザクション、制御 144
トランザクション、伝達 63, 67
トランザクションオブジェクト
インターフェースの継承 45
インターフェースの実装 45
の作成 45
ポリシーインターフェース 46
トランザクションオブジェクトからの移行 98
トランザクションコンテキスト
明示的な 121
トランザクションサービス
起動 142
トランザクション処理
公開標準 11
トランザクション処理の概要 13
トランザクション情報の取得 65
トランザクションの確認 63
トランザクションの監視 143
トランザクションの構築 49
トランザクションの作成 57, 67
トランザクションの詳細 144
トランザクションの消失 201
トランザクションの伝達 57, 63, 67
トランザクションのフィルタリング 145
トランザクションの例外 124
トランザクションリスト、再表示 144
同期
オブジェクトの実装 93
オブジェクトの登録 94
概要 93
コミット前の使用 94
障害による影響 95
トランザクションオブジェクトでの役割 95
ロールバックまたはコミット後の使用 94

な

名前、トランザクション、取得 73

に

ニュースグループ 5

ね

ネイティブ接続ハンドル 122
ネイティブハンドル取得インターフェース 138

は

ハングしたトランザクション 145
ハンドル
ネイティブ接続 122
パフォーマンスのチューニング 114
パフォーマンスの調整 114

ひ

引数 181, 187
非共有トランザクション 47
必要条件
Oracle ソフトウェアと XA 168
Oracle でのインストール 169
Oracle のインストールに関する問題 176
ヒューリスティック 89
Oracle OCI と XA の強制的完了 173
はじめに 89
ヒューリスティック、適用 114
ヒューリスティック、表示 145, 146
ヒューリスティック情報通知、有効化 79
ヒューリスティックな完了 78
ヒューリスティックの適用 114
ヒューリスティックログ、解釈 91

ふ

複数トランザクション 61
分散更新の問題
Oracle XA の 172
分散トランザクション 13
プール
接続の最適化 122
プラグイン可能データベースリソース 131
プラグイン可能モジュール、作成 136
プラグイン可能リソースインターフェース API
プログラミングの制限事項 139
プログラミングの制限事項
Oracle XA セッションマネージャとともに Oracle を使用する 171
Oracle9i を Oracle DirectConnect セッションマネージャとともに使用 178
許可されていない呼び出し 171
プラグイン可能リソースインターフェース API 139
プロファイルセット 147

へ

ヘッダーファイル 55
ヘルプトピック、アクセス 3

ほ

ポリシーインターフェース
InvocationPolicy 46
NonTxTargetPolicy 46
OTS 46
トランザクションオブジェクト 46
ポリシーの移行 98

ま

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 2
VisiBroker VisiNotify ガイド 2
VisiBroker VisiTelcoLog ガイド 2
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 11, 61
マルチスレッドトランザクション 52

み

未確定トランザクション 145
Oracle OCI と XA 172

め

明示的から暗黙的に 71
明示的なコンテキスト、Current からの取得 72
明示的な伝達 70
明示的なトランザクションコンテキスト 121
メッセージログ 146
VisiTransact メッセージログ 178
メッセージログ 172
メッセージログ、調整 147
メッセージログの調整 147
メッセージログのフィルタリング 146

ゆ

唯一の関数 137
ユーティリティ 181, 186

ら

ライブラリ、デプロイメント 112

り

リソースオブジェクト、完了の調整 81
リソースから戻される提案 84
リソースによる提案の戻し 84
リソースの準備 84
リソースの登録 83, 101
DirectConnect 105
利用できないリソースディレクタ 201

れ

例外 124
NoTransaction 201
OTS 80

ろ

ロールバック 85
ロールバック、マーク 64, 73
ロールバックするトランザクションのマーク 64, 73
ロールバックのまとめ 86
ログ、ヒューリスティック 90
ログ、ヒューリスティック、解釈 91