Nextra
トランザクション制御ガイド

Version 6.5

目　次

第 1 章 はじめに

本書は、マルチベンダ DB アクセスとトランザクションプロセス機能のための情報ソースです。シンプルなデータアクセス機能については『サーバ開発者ガイド』をご参照ください。

本書の利用方法

本書では、Nextra ツールを使用して、3 層分散アプリケーションからリレーショナル DB 管理システム(RDBMS)と DB ゲートウェイにアクセスする方法を説明します。

対象読者

本書は、RDBMS にアクセスするトランザクション・サーバを構築または管理する開発者および管理者を対象にしています。

前提知識

本書は、読者がクライアント／サーバコンピューティングと 2 層構造アーキテクチャの制限について基本的に理解していることを前提にしています。本書を読む前に、『始めにお読みください』、『サーバ開発者ガイド』および『運用／設定ガイド』をお読みください。

本書中では、C または COBOL の知識が必要となる部分もあります。

本書の使用方法

本書では、タスクオリエントの手順と、簡単な言語の説明を通じて、Nextra 開発ツールを使用したトランザクション・サーバの構築と実行に必要となる基本的なスキルを提供します。また、DB アクセス・サーバを拡張するために使われる高度な機能についても説明します。

オープン分散環境でのトランザクション・サーバの構築について経験がほとんどない方、または初めての方は、本書を参照してください。さらに、開発プロセス中も本書を参考資料として使用することができます。

表記規則

文中の表記規則

本書で使用する規則を理解しておくと、ユーティリティの使用方法などを容易に理解できます。

形式	説明	例
sub-text	ユーザが指定する必要がある値を示します。	*text*.def
bold	本文中では Nextra ユーティリティを示します。サンプル中では、強調される部分を示します。	**broker**
[brackets]	| がない場合は、オプションテキストを示します。| がある場合は、いずれか 1 つを選択することを示します。	[NONE | ERROR | WARN | DEBUG]

次の形式で区別されているパラグラフは、コード例です。

```
#include <stdio.h>

main() {
      int i;
      printf("The number is %d\n",i);
}
```

本書で使用するシンボル

本書では、次のようなシンボルを使用しています。

警告メッセージ

このシンボルに続くメッセージに、特別な注意を払う必要があることを示しています。このメッセージには重要な情報が含まれており、この情報を正しく理解してから先に進んでください。

ヒントメッセージ

このシンボルに続く本文は、必須ではありませんが状況に応じて役立つ手順であることを示しています。

オプションメッセージ

このシンボルに続く本文はオプションであることを示しています。内容は、追加機能または代替手法の概要、ある概念を理解するために役立つプロセスステップの詳細などです。

デバッグのヒント

このシンボルに続く本文は、プロジェクトの現在のステップをデバッグする手順が含まれていることを示しています。この方法はあくまで参考であり、別の有効なデバッグ方法の使用を妨げるものではありません。

第 2 章 TPMake

この章では、**TPMake** の機能を紹介し、トランザクション・サーバ作成の開発プロセスを説明します。詳細なサーバ開発の説明は、以降の章にあります。

この章を読み、トランザクション・サーバを構築する前に、『サーバ開発者ガイド』で説明した内容をご理解ください。また、ANSI-SQL (Structured Query Language)を使用した DB アクセスルーチンのコーディングについての知識も必要です。さらに、C 言語、または COBOL 言語のコーディングについての知識も必要になります。

はじめに

データ整合性の保証は、銀行業務や会計から在庫管理やサービストラッキングにいたるほとんど全ての情報システムに必要です。このため、オンライン・トランザクション処理サービスは、操作の 1 つのセットで 2 つ以上の DB を対象に処理される、全てのアプリケーションに必要になります。

トランザクション処理(TP)は、一連の複雑な操作全体にわたって、データ整合性を保証する方法です。トランザクション処理は、定義されたセットの操作、または操作セット全体の成功を保証するか、あるいは 1 つ以上の操作が失敗した場合はセットのロールバックを保証し、データがトランザクション開始直前の状態に戻るようにします。トランザクション全体を正常に実行するか、または全く実行しないことにより、TP は関連する一連のデータ間の一致を保証します。

TPMake

TPMake は、トランザクションを保障し、さまざまな RDBMS の下で実行される、複数 DB にアクセスできるサーバの開発をサポートします。

コンポーネント

- **tpmake** (%ODEDIR%\bin にインストールされる実行可能なモジュール)
- **tpmk_cbl** (%ODEDIR%\bin にインストールされる実行可能なモジュール。COBOL での開発をサポート)

- **librpc.*ext*** (%ODEDIR%\lib にインストールされるライブラリ)
- **adcp_*dbms.ext*** (%ODEDIR%\lib にインストールされるオブジェクト)
- **transac.h** (%ODEDIR%\include にインストールされるヘッダ)
- **dbcommon.h** (%ODEDIR%\include にインストールされるヘッダ)

TPMake は、リソースマッピングファイル、および SQL ファイルを読み取り、次の複数のファイルを生成します。

- SQL ステートメントを含む C ファイル
- サーバを初期化する C ファイル
- サーバコンパイルのための **Makefile** (オプション)
- ヘッダファイル
- 静的 SQL では、照会を作成し実行するための ESQL/C コード

librpc.*ext* は、RPC ランタイムライブラリです。

adcp_*dbms*.o は、RDBMS 固有のオブジェクトファイルで、ESQL/C 関数と DB 関数を含みます。

transac.h は Nextra トランザクション制御関数、初期化／クリーンアップ関数、内蔵 RPC、およびグローバル変数定義の関数プロトタイプを含みます。

dbcommon.h は、RDBMS から返される可能性のあるエラーの戻り値を含みます。

使用方法

トランザクション・サーバは図 2.1 で示すように、2 つのフェーズで構築します。

第 1 フェーズでは、まず、各 DB 毎に、実行時にサーバが処理する SQL ファイルを作成します。次に、各 SQL ファイルを特定の DB にマッピングするリソースファイルを作成します。そして、このリソースファイルを **TPMake** ユーティリティで指定します。 **TPMake** ユーティリティは、先に記述した SQL ステートメントを含む C ファイルを生成します。

第 2 フェーズでは、 **TPMake** が生成したコードを使用して、トランザクション・サーバファイルを作成します。そして、サーバの実行ファイルをコンパイルします。

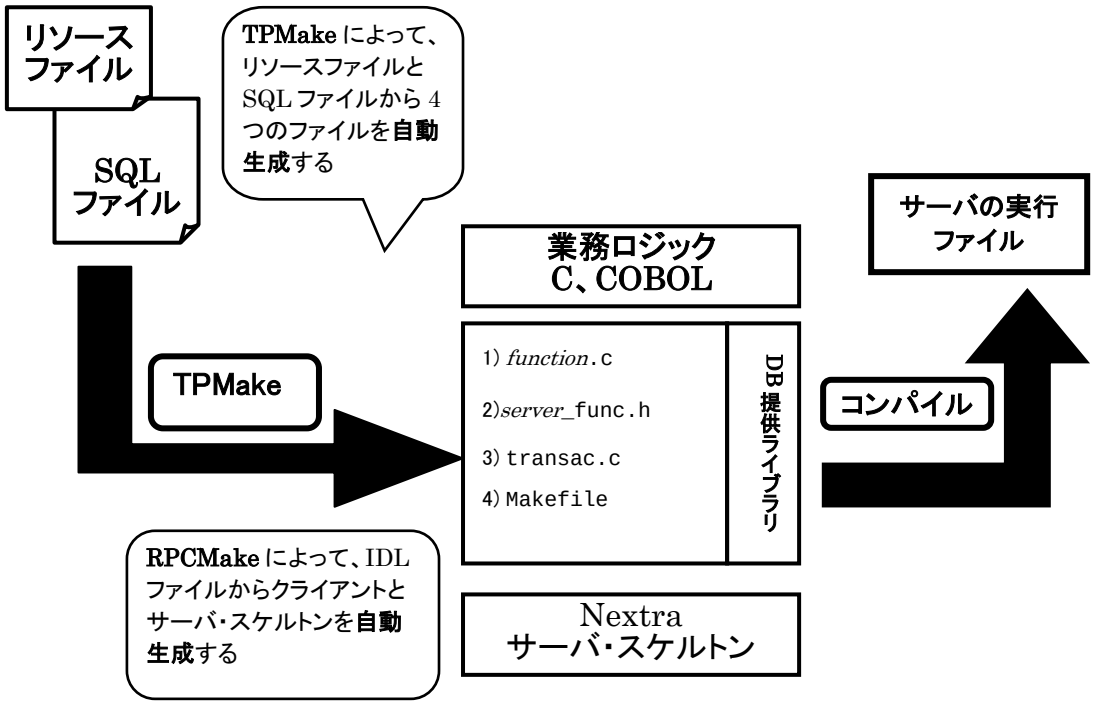

図 2.1: **TPMake** の使用

始める前に

プラットフォームの必要条件

Nextra 開発パッケージが、トランザクション・サーバを開発する全てのプラットフォーム上にインストールされていなければなりません。

Nextra 開発パッケージ、ANSI-C コンパイラ（または MF COBOL コンパイラ）が、トランザクション・サーバをコンパイルする全てのプラットフォーム上にインストールされていなければなりません。

トランザクション・サーバが、アクセスする DB とは別のプラットフォームで実行される場合は、RDBMS 固有のネットワーク対応ソフトウェアを、サーバが実行されるプラットフォームごとにインストールして、トランザクション・サーバが DB に接続できるようにする必要があります。さらに、Nextra サーバが実行されるプラットフォームには、Nextra 運用パッケージがインストール されていなければなりません。

環境変数の設定

Nextra ツールを使用する前に、トランザクション・サーバを開発および実行する全てのプラットフォームで、いくつかの環境変数を設定しておく必要があります。

UNIX 変数

UNIX と Windows 環境の両方で、ODEDIR 環境変数が tcp サブディレクトリを正しく指していることを確認します。次に、$ODEDIR を PATH 変数の一番前にインクルードします。

[Bourne シェルまたは ksh]

```
ODEDIR=/usr/nextra/tcp

PATH=$ODEDIR/bin:$PATH

export ODEDIR PATH
```

Windows 変数

Windows 環境では、次の環境変数が設定されていることを「コントロールパネル」の「システム」画面で確認してください。

```
ODEDIR = C:\NEXTRA\TCP
```

PATH = %ODEDIR%\BIN;*old_PATH*

RDBMS 変数

全ての RDBMS では、PATH 変数が、RDBMS の実行ファイルを含むディレクトリ(通常 *home_directory*/bin)を指すことを確認します。

浮動小数点の列から返されたデータの小数点以下の桁数を設定するために、PRECISION 環境変数を使用します。小数点の右に表示する桁数の値を設定します。PRECISION を設定しない、または 0 に設定する場合、小数点以下 6 桁が返されます。

e を PRECISION 値に追加することによって、指数表記を指定できます。g を PRECISION 値に追加することによって、非常に大きい値、または非常に小さい値だけの指数表記を指定することができます。

また、この PRECISION 環境変数は、adhoc のみに有効となります。(adhoc についての説明は、第 3 章の「adhoc 照会」を参照してください。)

表 2.1： PRECISION を用いた場合の値の例

PRECISION=5	PRECISION=4	PRECISION=4e	PRECISION=4g
7648.65431	7648.6543	7.6486e3	7648.6543
4.12457	4.1246	4.1246e0	4.1246
675391.00000	675391.0000	6.7539e5	6.7539e5
.02100	.0210	2.1000e-2	.0210

RDBMS 固有の変数

さらに、次の DB 固有の変数を設定しなければなりません。

RDBMS	DB 固有の変数
Oracle	ORACLE_SID=*name_of_database* ORACLE_HOME=*home_directory_of_ORACLE* ORALEN=*desired_length_of_`long'_data_fields* ***a** PATH に$ORACLE_HOME/bin を含めなければいけません。 注:異なる DB にアクセスするトランザクション・サーバがある場合は、それぞれ別の ORACLE_SID 環境変数を設定します。 ***a.** long Oracle データ型を含む Oracle のテーブルにアクセスする場合、トランザクション・サーバはデフォルトカラムサイズ 8000 を使用する。8000 文字以上を含む long フィールドは、8000 番目の文字の直後で切り捨てられます。8000 文字制限を変更するには、ORALEN 環境変数を使用してサイズを指定します。そして、トランザクション・サーバを起動します。トランザクション・サーバ実行中、ORALEN の値の変更は無効です。
SQL Server	DSQUERY=`name_of_remote_SQL_server`, PATH に `SQL_Server_dir`¥binn を含めなければなりません。
DB2	DB2INSTANCE=`user_name_of_RDBMS_owner`

実行時の処理

トランザクション・サーバは、次のように入力することによって起動します。

prog_name [-a *auditfile*] -e *env_file*

ここで、***prog_name*** は実行するサーバ名、***auditfile*** はトランザクションログ情報を含むファイル名、***env_file*** はサーバの環境ファイル名になります。

サーバは起動されると、次の処理を実行します。

- まず、必要な DB エンジンが全て起動され、実行されているかどうか、そして必要な DB が存在するかどうかをサーバはチェックする。
- 次に、サーバは環境ファイルを読み込み、自分のためのソケットを確立する。
- 最後に、サーバはブローカにトランザクション・サーバとして登録され、クライアント要求を待つ。

サーバが DB に個々の照会を行うたびに、Nextra は照会のステータスを示す固有の値を返します。値は正または負の整数です。値の意味は次のとおりです。

戻り値	意味
DCPSUCCESS	関数呼び出しが正常終了したことを示します。
DCPERROR	一般エラー。しばしば DB マネージャ内部のエラーのために、関数呼び出しに失敗したことを示します。RDBMS 固有のエラーコードについてはログファイル、詳細については RDBMS のドキュメントを参照してください。
<= -1000000	DB マネージャ外部のエラーのために関数呼び出しに失敗したことを示します。どのエラーが発生したかを調べるには、dce_errnum()または dce_errstr()を呼び出してください。また、発生するエラーの定義については、dbcommon.h 中のエラーコードを参照してください。

ユーザが呼び出した関数は、この戻り値を参照することになります。

DB に対するログインセッションは、トランザクション・サーバのプロセスが停止したときに終了します。

サポートされるデータ型

Nextra トランザクション・サーバ・ライブラリでサポートされる SQL '92 エントリレベルのデータ型は、character、integer、small integer、float、real、double です。これらの型に加えて、上記の全ての RDBMS では、拡張レベルの Binary Large OBject(BLOB)型がサポートされます。BLOB データを示す場合、binary が使用されます。サポートされるデータ型は、TPMake によって RDBMS 固有のデータ型にマップされます。decimal と numeric のデータ型は、対応する IDL 型がないためサポートされません。サポートされないデータ型は全て文字列として扱われます。

アーキテクチャ

図 2.2 に示すように、サーバは、トランザクション処理機能を実現するために、高度にモジュール化された多階層構造を使用します。

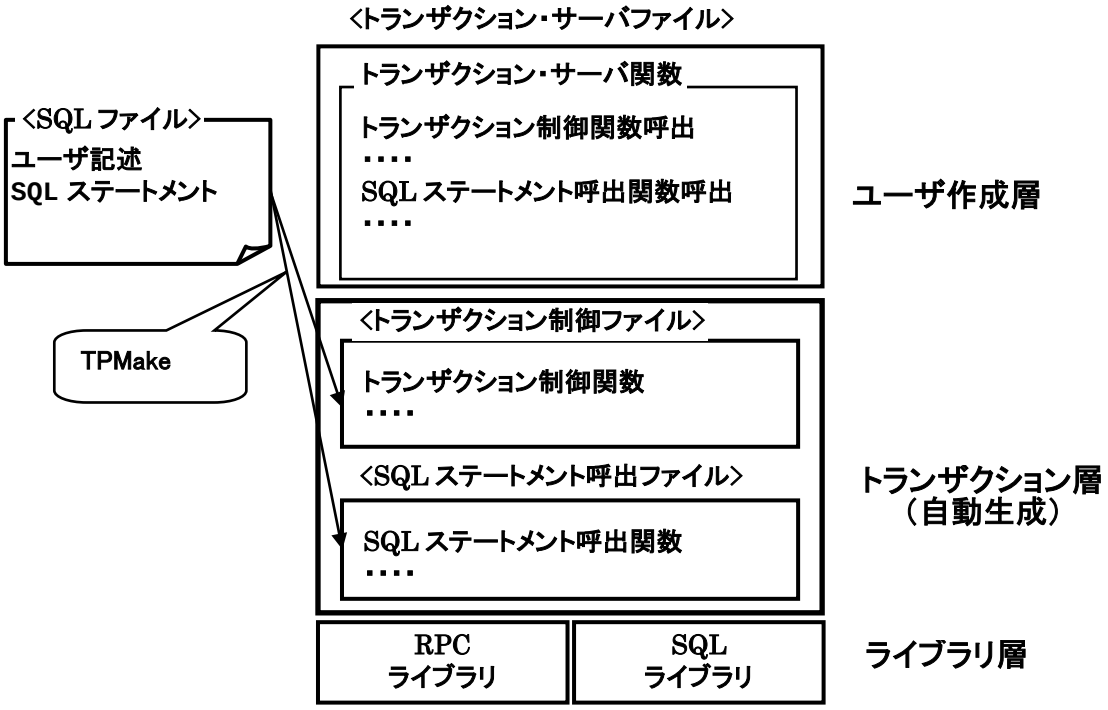

図 2.2： トランザクション・サーバのアーキテクチャ

クライアントは、開発者が記述したトランザクション・サーバ関数を直接呼び出します。次に、トランザクション・サーバ関数は、**TPMake** が作成したトランザクション制御関数と SQL ステートメント呼出関数を呼び出します。そして、SQL ステートメント呼出関数が SQL のライブラリを呼び出します。

トランザクション・サーバ関数の内部で、Nextra トランザクション制御関数である、begin_work(), commit_work(), abort_work()を使用して、トランザクションを制御します。SQL ステートメント呼出関数の内部で、各 SQL ステートメントが実行され、チェックされます。DB 操作に失敗した場合(commit を除く)、トランザクション・サーバ関数内で行

われた全ての作業がアボートされ、トランザクション開始直前の状態にロールバックします。コミットの失敗については、以下を参照してください。

トランザクションのロギング

サーバ起動時に監査ファイル（auditfile）を指定する（*prog_name* [-a *auditfile*] -e *env_file*）と、サーバが実行した各 SQL の結果が監査ファイルに記録され、ファイルは削除されるまで大きくなり続けます。監査ファイルを指定しないと、*server*_transaction.log というデフォルトのログファイルに、最新のトランザクションだけが記録されます。このデフォルトの監査ファイルは、トランザクションが正常終了する度に、最初の状態にリセットされます。データの破壊を引き起こす可能性のあるエラーが発生した場合、障害に先だってコミットされたトランザクションの一部がログファイルに書き込まれます。

理論上は、トランザクションが実行されてからコミットされるまでの間に、エラーが生じることもあり得ます。そのようなエラーが原因で commit_work()文が失敗すると、データが破壊される可能性があります。サーバプログラムは、複数の DB に全く同時に指示を送ることはできません。したがって、サーバは各 DB に順にアクセスし、commit を実行しなければなりません。最初の commit に失敗すると、それ以降の commit は試行されることなく、トランザクション全体が失敗に終わります。しかし、ある commit に成功し、次の commit に失敗した場合、障害発生に先立つ成功した commit をロールバックすることはできません。この場合、ログファイル（指定されている場合は監査ファイル）のトランザクションを調査し、commit に失敗する前にトランザクションで処理されたデータを手動で前の値に戻して、整合性をとる必要があります。

開発プロセス

ここでは、**TPMake** を使用してトランザクション・サーバを構築する方法を説明します。手順の詳細については、次の章を参照してください。

トランザクション・サーバ構築の概要

サーバのコーディングにはさまざまな方法がありますが、トランザクション・サーバを構築する開発プロセスには一貫性があります。トランザクション・サーバの構築は 2 つのフェーズから成ります。

SQLファイルより C コードの生成

1. 最終的にトランザクション・サーバが実行することになる ANSI SQL ステートメントを、DB 毎に 1 ファイルずつ作成します。

2. SQL ステートメントを Nextra 準拠の形式に変換します。

3. SQL ファイルを対応する DB にマッピングするリソースファイルを作成します。

4. SQL 文を含む C コード TPMake にて生成します。

サーバの C ソースコード(および、オプションで、コンパイルのガイドラインとして使用するための Makefile)を自動的に生成するために、リソースファイルを **TPMake** ユーティリティの入力として渡してください。

トランザクションのコーディング

1. トランザクション・サーバの RPC インタフェースを定義する IDL ファイルを作成します。

COBOL でサーバを開発する場合はサイズファイルも作成します。

2. TPMake が生成した SQL ステートメント呼出関数を含むトランザクション・サーバ関数を作成します。

3. IDL ファイルを RPCMake ユーティリティに渡して、サーバ・スケルトンを生成します。

COBOL でサーバを開発する場合はサイズファイルも **RPCMake** に渡します。 このステップは、オプションで生成される Makefile のように、コンパイル時に Makefile で実行される場合もあります。 トランザクション・サーバをコンパイルしてください(**TPMake** がオプションで作成した Makefile を使用することをお勧めします)。

4. 環境ファイルを作成します。

これで、トランザクション・サーバを実行できるようになります。

デバッグ

RPC Developer に含まれる RPCDebug にて、新しいサーバプログラムをテストしてください。

第 3 章 トランザクション・サーバの構築

この章では、**TPMake** の使用に必要な基本的な知識を提供します。TPMake 使用方法の詳細については『リファレンス』を参照してください。

この章を読む前、またはトランザクション・サーバの構築を試みる前に、『サーバ開発者ガイド』の内容、および『運用／設定ガイド』の一部に精通している必要があります。また、本書の前の章にも精通している必要があります。C 言語（または COBOL 言語）と ANSI-SQL のコーディング方法の知識も必要です。

開発プロセス

以下のセクションで、トランザクション・サーバのコーディングとコンパイルの方法の詳細について説明します。

2 フェーズ・コミット制御

TPMake によって、複数の DB からなる RDBMS、または異なる RDBMS から構成される DB に対するトランザクションを処理するための 2 フェーズ・コミット制御を行うトランザクション・サーバを作成することができます。高度なデータの整合性を必要とするミッションクリティカルなアプリケーションでは、2 フェーズ・コミット制御によって、複数の DB に対するオペレーションを含むトランザクションを、確実に不可分(atomic)なものにすることができます。「不可分な」トランザクションでは、全てのオペレーションは正常に終了する必要があり、1 つ以上のオペレーションが失敗すると、全てのオペレーションの結果がロールバックされます。

2 フェーズ・コミット制御を使用すると、2 つのフェーズでトランザクションを処理することによって、容易に不可分性を確実なものにすることができます。第 1 フェーズでは、全てのリソースマネージャに対して更新準備ができたかどうかを確認します。第 2 フェーズで、DB への変更をコミットするが、全リソースマネージャがコミットの準備ができている場合にのみ行われます。注) 現在のバージョンでは、XA インタフェースはサポートしていません。

SQL Server の DB にアクセスするトランザクション・サーバは、常に 2 フェーズ・コミット制御を行います。したがって、SQL ファイルに変更を行う必要がありません。

Oracle の DB にアクセスするトランザクション・サーバで、2 フェーズ・コミット制御を有効にするには、照会で非デフォルト DB テーブルにグローバルオブジェクト名を指定します。グローバルオブジェクト名の形式は、RDBMS によって異なります。

Oracle では、各 DB ごとにグローバルオブジェクト名を作成します。それから、svrmgrl (または、sqlplus)ユーティリティを使用して、各グローバルオブジェクト名を、tnsnames.ora ファイル内の各 SQL-Net エイリアスとリンクさせます。たとえば、以下のようになります。

```
create public database link dbtest.oltp using `dboltp`;
```

ここで、*dbtest.oltp* はグローバルオブジェクト名、*dboltp* はファイル tnsnames.ora 内の SQL-Net エイリアス名です。*dboltp* エイリアス名は、マシン oltp 上の DB dbtest を指します。

「デフォルト DB」とは、リソースファイル内の各エントリごとに指定された DB です。2 フェーズ・コミット制御では、リソースマネージャは、デフォルト DB に対してセッションをオープンし、このセッションを使用してトランザクションを調整します。2 フェーズ・コミット制御の SQL ファイルとリソースファイルの作成の詳細については、「TPMake 用のリソースファイルの作成」を参照してください。

ANSI SQL ファイルの作成

トランザクションは 1 つ以上の個々の SQL ステートメントで構成され、まとめて実行するためにグループ化されています。後でそれらをどのようにグループ化するかに関わらず、テキストファイルに個々の SQL ステートメントを書き込んで、開発プロセスを始めてください。同じ DB にアクセスする SQL は、全て 1 つのファイルに格納しなければなりません。

エディタを使用して個々の SQL ステートメントを入力してください。

- 指定したファイル中の SQL ステートメントは 1 つの DB にのみアクセスできる(たとえば、Oracle の下の「vendorinfo」DB と、SQL Server の下の「accounting」DB にアクセスするには、2 つのファイルが必要)。
- SQL ステートメントは特定の順序に並べる必要はありません。
- デリミタ(delimiter、たとえばセミコロン)毎に 1 つの SQL ステートメントが記述されなければいけません。

サンプルの ANSI SQL ファイルを以下に示します。

```
select amount_val from checking where act_num = '1001';
```

列名が有効な C 変数名でない場合、**TPMake** は名前を有効にしようとします。このステップで有効な変数名が作成されない場合、 **TPMake** は変数に outvars#という名前を付けます。ここで、#は変数の順序を示す値です。SQL ステートメントの列名のエイリアスを作成した場合、エイリアスは変数名として使用されます。使用方法については、RDBMS のドキュメントを参照してください。

SQL ファイルのデバッグ

個々のファイルを完成したら、RDBMS 固有のインタラクティブ SQL ユーティリティを通じて、DB マネージャへファイルをパイプし、SQL ステートメントのシンタックスをテストしてください。

例:

```
cat test_file | DB_access_method > resultfile
```

引数の意味は次のとおりです。

引数	意味
test_file	SQL ステートメントを含むファイル名。
DB_access_method	RDBMS 固有のインタラクティブ SQL ユーティリティを起動する文字列。(Oracle に対しては sqlplus) 必要に応じて、ユーザ名、パスワード、DB 名の引数を渡します。
resultfile	RDBMS が SQL を実行した後に、SQL の出力を受け取るファイル名。

このステップは省略しないでください

SQL ステートメントが、インタラクティブ SQL ユーティリティにて正しく動作しない場合、それはサーバコードとしても機能しません。必ず、SQL ステートメントが RDBMS に対して正しく動作することを確認してください。

注:RDBMS によってシンタックスは異なるかもしれません。重要なことは、SQL を DB にパイプし、出力をファイル、プリンタ、または画面にリダイレクトすることです。RDBMS が提供する他の方法によって、この作業を実行してもかまいません。

別のオプションとしては、SQL ファイルを実行可能にして(UNIX 環境では chmod +x *filename*)、インタラクティブ SQL ユーティリティーを起動するために、先頭にコマンドを追加する方法があります。例を示します。

[Oracle の場合]
pathname/sqlplus *user/passwd* <<!
statements
!

[MS SQL Server の場合]
次のラインを含むファイルを作成する。

use *db_name*
go
statement
go
...

そして、次のようにインタラクティブ SQL ユーティリティを実行する。

pathname\isql - U *user* -P *password* -S *server* <<*filename*>
!

この方法を使用して出力をファイルに直接送り、次のように SQL ファイルを実行してもかまいません。

```
C:\testbed> test_file > resultfile
```

出力ファイルを確認してください。SQL 実行の結果、適切に動作しない SQL があることが示されたら、その SQL ステートメントを変更します。

そして、再度テストしてください。

Nextra 準拠形式に合わせるための SQL の修正

SQL ステートメントのシンタックスをそれぞれテストした後、次のステップでは、個々の SQL ステートメントを Nextra 準拠形式に変換します。このためには、

a) 個々の SQL ステートメントに関数ラベルを付け、

b) リテラル値(ハードコード値)の代わりに、ドル記号($)を付けた引数変数に置き換えてください。そして、

c) データ型を示す属性文字列を、引数変数の場合は変数名の後、出力値の場合はカラム名の前に挿入してください。

大括弧[ ]で囲んでデータ型または受け入れられるその省略形を追加してください。

引数変数は次のようになります。

variable[char|int|smallint|real|float|double|bin]

出力値は次のようになります。

[char|int|smallint|real|float|double|bin]*output*

ここで、*variable* は引数変数の名前になります。*output* はカラム名になります。

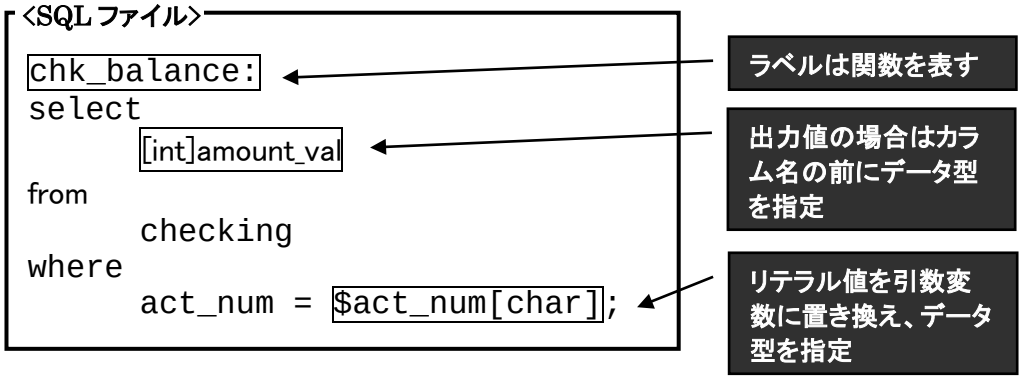

図 3.1： Nextra 準拠の SQL ファイルの作成

データがネイティブのデータ型としてではなく文字列として返される必要がある場合は、各カラム名の前にデータ型を指定する必要はありません。

また、引数変数にデータ型を指定することを推奨しますが、もしも、引数変数にデータ型を指定しない場合は、文字フィールドの場合、クォーテーションの中を変数名に置き換えるだけです。数字フィールドの場合、クォーテーションは不要です。よく分からない場合は、照会を RDBMS に送るときのように、通常、リテラル値に使用するのと同シンタックスを使用してください。

ほとんどの RDBMS はシングルクォーテーションを受け付けますが、ダブルクォーテーションを受け付けないものもあるため、シングルクォーテーションの使用をお勧めします。

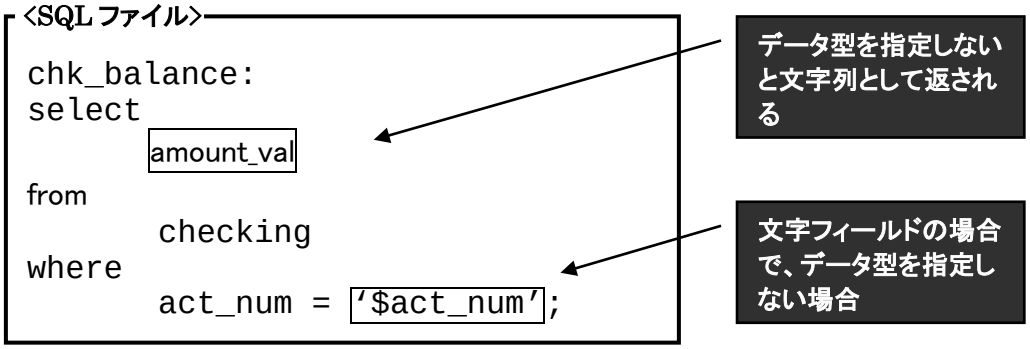

図 3.2： データ型を指定しない場合

各ファイルの SQL ステートメントを全て Nextra 準拠形式に変換すると、正式な SQL ファイルが完成します。いくつかの異なる RDBMS が操作される場合は、内容を区別できるようにファイル名を付ける必要があります。
COBOL でサーバを開発する場合は『サーバ開発者ガイド』の「COBOL サイズ情報の挿入」に従って、SQL ファイル中に COBOL のサイズ情報を記入してください。

adhoc 照会

adhoc 照会を許可するには、adhoc_*txt* という関数を SQL ファイルに挿入します。*txt* は任意の値です。生成される C ファイルには、同じ名前の対応する関数が含まれます。これで、トランザクション中に adhoc 関数を呼び出すために、この関数を使えるようになります。

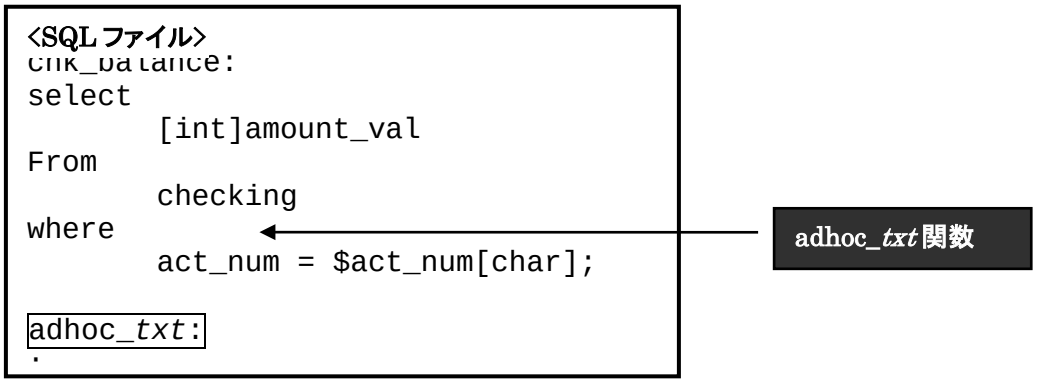

図 3.3： adhoc 関数を含む SQL ステートメント

参照

Nextra 準拠の SQL ファイルについては、『リファレンス』の「ファイル仕様」の章を参照してください。

TPMake 用のリソースファイルの作成

トランザクション・サーバに必要な C コードを生成するために、**TPMake** ユーティリティは、SQL ファイルがそれぞれどの DB と RDBMS にアクセスするかを知る必要があります。この情報はリソースファイルに書き込みますが、各ラインは、『リファレンス』の「トランザクション・サーバ用リソースファイル」で説明されているシンタックスに従います。

通常のトランザクション・サーバ

リソースファイルの例を示します。

```
#tag file        db_name RDBMS  login     passwd

1    tpserv1.sql acct_db mssql  act_staff rt45wau
2    tpserv2.sql sales   oracle staff     k5tga8K
```

デディケイテッド・サーバの場合のみ、クライアントは sql_connect_*interface*() 関数を使用して、実行時にリソースファイルで指定されたデフォルト DB、RDBMS、ログイン、パスワードを無効にすることができます。詳細については、「トランザクション・サーバのセッションの開始」を参照してください。

2 フェーズ・コミットの場合

Oracle データベースへのアクセス時に 2 フェーズ・コミット制御を使用するトランザクション・サーバを構築するには、リソースファイル内のエントリごとに同じ DB と RDBMS を指定します。非デフォルト DB を指定するときには、DB 名フィールドでパイプ(||)デリミタを使用しないでください。

たとえば、次のようになります。

```
#tag  file    db_name  RDBMS  login   passwd

1 tpserv1.sql  empl  oracle  act_staff  rt45wau
2 tpserv2.sql  emp1  oracle  act_staff  rt45wau
3 tpserv3.sql  emp1  oracle  act_staff  rt45wau
```

上記の例では、1 つまたは複数のトランザクションは、Oracle データベースに対する 3 つの SQL ファイル内にオペレーションを含めます。リソースファイルは 1 つの DB-- empl --だけをリストしていますが、複数の DB が含まれている場合があります。リソースファイル内に指定された DB がデフォルト DB です。RDBMS はデフォルト DB のためにセッションをオープンし、そのセッションを使用して 2 フェーズ・コミット制御を調整します。

トランザクションにおいてデフォルト DB 以外の DB にアクセスするには、その DB の完全なグローバルオブジェクト名とテーブルを SQL ファイルに指定します。SQL 記述方法については、各 RDBMS のドキュメントを参照してください。

図 3.4 では、3 つの異なる Oracle データベースを含む 2 フェーズ・コミット制御を実行する、トランザクション・サーバを示します。以下では、実行時のアクションを説明します。

1. トランザクション・サーバを起動します。

2．トランザクション・サーバは、empl デフォルト DB を実行するリソースマネージャによって、パーマネントセッションをオープンします。

3．トランザクション・サーバは、トランザクションに関連する 3 つの DB 全てのリソースマネージャにデータアクセス照会を送ります。

4．デフォルト DB のリソースマネージャは、トランザクション中でそのリソースマネージャが行う部分のデフォルト DB を準備し、他の 2 つの DB のリソースマネージャに対して、トランザクション中でそれぞれのリソースマネージャが行う部分の DB を準備するように指示します。

5．3 つのリソースマネージャ全てが、トランザクションのそれぞれの部分を正常に終了すると、デフォルト DB のリソースマネージャはそのトランザクション部分をコミットし、他の 2 つのリソースマネージャにそれぞれのトランザクション部分をコミットするように指示します。

1 つ以上のリソースマネージャがそのトランザクション部分を正常終了することができない場合、デフォルト DB のリソースマネージャは、自分のトランザクション部分の結果をロールバックし、他のリソースマネージャに対して、それぞれのトランザクション部分の結果をロールバックするように指示します。

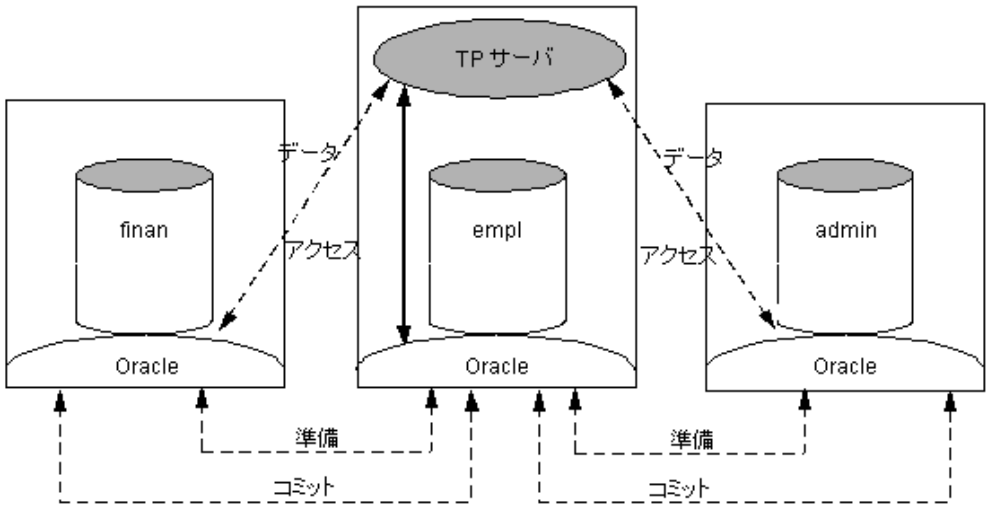

図 3.4： 2 フェーズ・コミット制御を使用するトランザクション・サーバ

SQL Server の DB にアクセスするトランザクション・サーバは、必ず 2 フェーズ・コミット制御を使用します。リソースフィールド内の最初のエントリに指定されたセッションがトランザクションを調整します。

環境変数の無効化

リソースファイル中の DB 名、ログイン、パスワード値を無効にするために使用できる環境変数があります。詳細については、『リファレンス』の「デフォルト値の無効化」を参照してください。

C コードの生成

トランザクション・サーバの SQL ステートメント呼出ファイルとトランザクション制御ファイルなどを自動的に生成するため、リソースファイルを **TPMake** ユーティリティに渡します。ユーティリティは、コマンドラインまたは GUI で起動することができます。

コマンドラインインタフェース

コマンド発行時に-s および -f 引数を指定すると、**TPMake** のコマンドラインモードが起動されます。

```
> tpmake -s server_name [-f resource_file] [-M]
```

COBOL でサーバを開発する場合は、tpmake コマンドのかわりに tpmk_cbl コマンドを使用してください。オプションは同じです。

ここで、*server_name* は、トランザクション・サーバの実行ファイルを示すことになるサーバ名であり、かつ IDL ファイルで指定されたインタフェース名でもある。
resource_file は、各 SQL ファイルがアクセスする DB と RDBMS をリストしているリソースファイル名です。-M を指定して Makefile を生成することができます。全ての **TPMake** の引数の詳細については、『リファレンス』の「TPMake」を参照してください。

ファイルのネーミング

SQL ファイル名にはインタフェース名を付けないでください。付けると、同名のトランザクションコードファイルが上書きします。

GUI

コマンドライン引数なしでコマンドを実行すると、**TPMake** を GUI で起動することができます。

```
> tpmake
```

または

```
> tpmk_cbl
```

tpmk_cbl は TPMake と同じ GUI を使用しています。サーバ言語の選択で、MF Cobol を選択することで、tpmk_cbl を実行したときと同じ結果が得られます。

UNIX の場合、「Can't Open Display」というエラーメッセージが通知されたら、DISPLAY 環境変数が X/window で必要とされる値に設定されているかどうかを確認してください。

TPMake の GUI については、図 3.5 を参照してください。

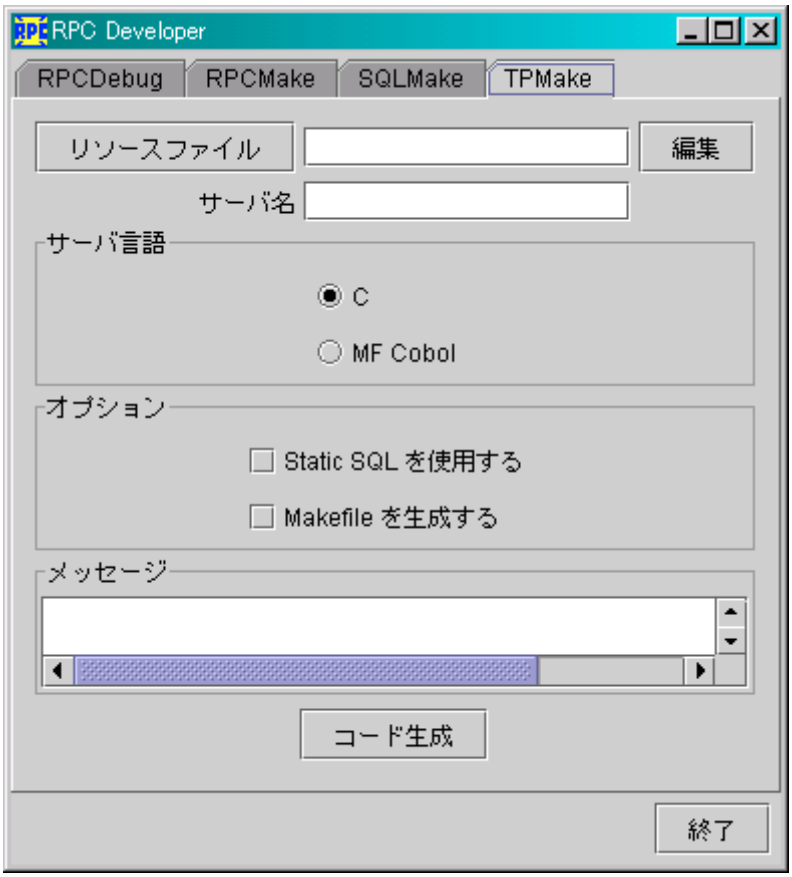

図 3.5： **TPMake** の GUI 画面

GUI を使用するには、次のステップに従ってください。

1. 「リソースファイル」ボックスにリソースファイル名を入力します。

正しいリソースファイルを見つけるために、ディレクトリを検索するには、「リソースファイル」ボタンをクリックしてください。

2．「サーバ名」ボックスに、インタフェース名を入力します。

TPMake はこの文字列を使用して、出力ファイル名を付けます。また TPMake は、最終的にコンパイルされたサーバコードを含むファイルとして、Makefile にもこの名前を指定します。

3．サーバ言語を「C」または「MF Cobol」より選択します。オプションとして、Makefile を生成するかどうかを指定します。

4．「コード生成」ボタンをクリックして、選択したものを生成します。

5．「終了」ボタンをクリックして、ユーティリティを終了します。

出力ファイル

以上で、次のファイルが生成されています。

ファイル	内容
name1.c *name2*.c ...	リソースファイル中にリストされる各 SQL ファイルのための C コードのファイル。*name1* と *name2* は、個々の SQL ファイル名の接頭語に対応します。
transac.c	トランザクション・サーバを初期化する C コードのファイル。このファイルには、固有の RDBMS のためのトランザクション API 関数を含みます。この関数には、begin_work(), abort_work(), commit_work(), sql_connect_interface(), sql_set_max_rows_*interface*()が含まれます。
*server*_funcs.h	SQL ファイルに含まれる SQL ステートメントのための関数プロトタイプを含むヘッダファイル。*server* は-s フラグに続けてコマンドラインにて **TPMake** に渡す文字列です。また、function.o (function.obj) で提供される関数のためのプロトタイプと、グローバル変数の外部宣言も含みます。 このヘッダファイルは、開発者が記述するトランザクション・サーバファイルにインクルードしなければなりません。
Makefile	(オプション) サーバの実行ファイルをコンパイルする際に使用する Makefile。

TPMake はサーバの実行に必要なコードを全て生成しますが、ユーザが作成しなければならないファイルが 2 つあります。

1. サーバ・スケルトン: トランザクション・サーバ開発のために必要なファイル。IDL ファイルを作成し RPCMake を実行し生成される。

2. トランザクション・サーバファイル: commit と abort の機能を組み合わせ、関数をトランザクションにグループ化するコードを作成する必要がある。

TPMake に関連する入力ファイルと出力ファイルの関係を図 3.6 に示します。

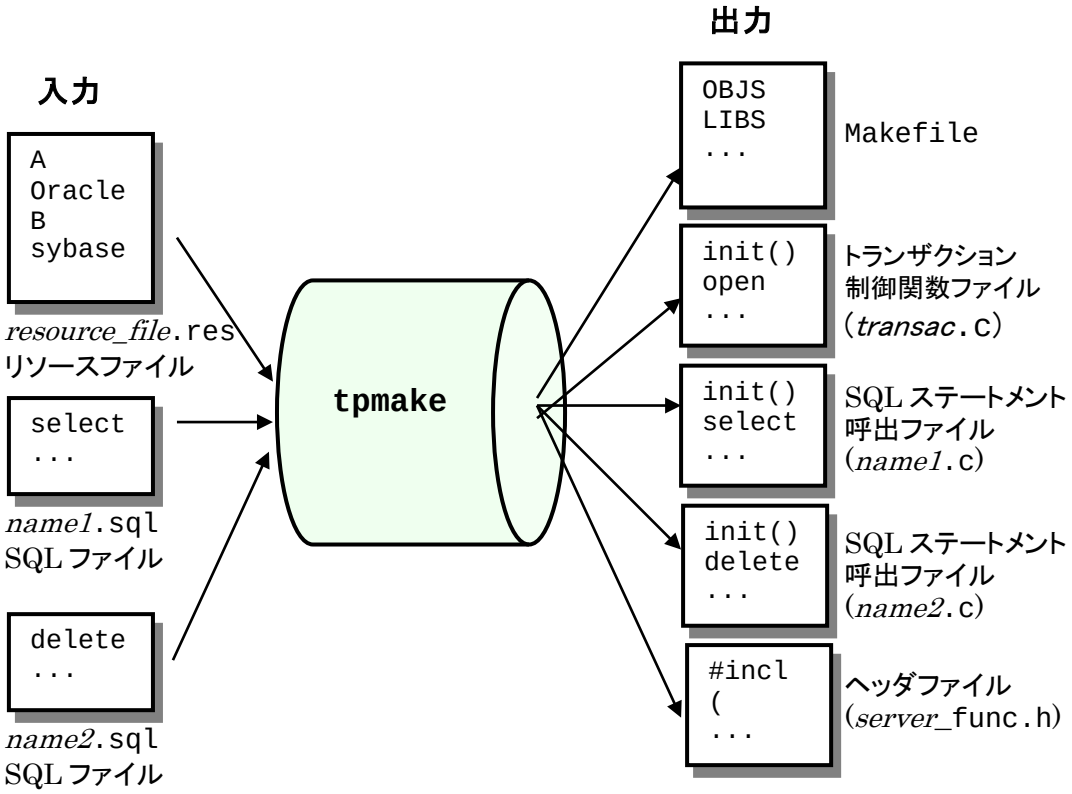

図 3.6： **TPMake** の関連ファイル

静的 SQL の使用

TPMake では、トランザクション・サーバのパフォーマンスを最適化するために、デフォルトの動的 SQL ではなく、静的 SQL を使用するオプションがあります。静的 SQL を使用するには、コマンドラインで-E 引数を指定します。あるいは、**TPMake** の GUI を起動し、オプションで「Static SQL を使用する」ボタンをクリックします。これによって、**TPMake** は、動的 SQL の場合に生成する関数呼び出しの代わりに、組み込み SQL/C(ESQL/C)コードを生成します。

リソースファイル内のエントリごとに、**TPMake** は ESQL ファイルを生成し、以下のネーミングルールに従って名前を付けます。

```
query_file_static.ext
```

表 3.1 は全ての RDBMS の拡張子をリストしたものです。

表 3.1： ESQL ファイルの RDBMS 固有の拡張子

RDBMS	拡張子
Oracle	.pc
SQL Server	.c
DB2	.sqc

たとえば、Oracle データベースにアクセスする SQL ファイル tpserv1.sql と tpserv2.sql を含むリソースファイルでは、**TPMake** は、tpserv1_static.pc と tpserv_static.pc という名前の ESQL ファイルを生成します。

IDL ファイルの作成

サーバ・スケルトンを作成するには、別のファイル、つまり IDL ファイルを作成する必要があります。このファイルは、サーバ・スケルトンコードを生成するために、**RPCMake** ユーティリティに対する入力として使用されます。

IDL ファイルは、トランザクションにおける個々の関数のための入力変数と出力変数を記述します。IDL ファイルは、開発者が記述したトランザクション・サーバファイル中の関数とパラメータをリストします。

『リファレンス』の「インタフェース定義（IDL）ファイル」、または『サーバ開発者ガイド』の「IDL ファイルの作成」で説明されているシンタックスを使用してください。

各関数名はトランザクション・サーバ関数名と同じで、入力と出力は、各トランザクション・サーバ関数のパラメータリストに対応づけられなければなりません。

トランザクション・サーバをデディケイテッド・サーバとして実行するつもりなら、sql_connect_interface()と sql_set_max_rows_interface()の定義を含めてください。詳細については、「デディケイテッド・サーバにおけるカーソルの使用」を参照してください。

COBOL でサーバを開発する場合は、『サーバ開発者ガイド』の「COBOL サイズ情報のインクルード」に従ってサイズファイルを記述してください。その際、char 型の配列変数を宣言するなら、ターミネータキャラクタとして最後に NULL 用に 1byte 必要な点に注意してください。『サーバ開発者ガイド』の「C 配列に変換される COBOL 配列の例」を参照してください。

トランザクション・サーバ関数コードの作成

ここまでは、個々の SQL ステートメント呼出関数は技術的にはトランザクションにグループ化されていません。ここで、個々の SQL ステートメント呼出関数を呼び出すトランザクション・サーバ関数を(C または COBOL で)コーディングする必要があります。トランザクション制御関数 abort_work()、begin_work()、および commit_work()については「Nextra トランザクション制御関数」を参照してください。

個々のトランザクション・サーバ関数の本体は、begin_work()の呼び出しで始めなければなりません。この関数は、1 つのパラメータ、つまり起動する関数名を受け付けます。この関数は、個々の DB に対して、指示があるまでは実行せずに各 SQL を受け付けるように通知します。

個々のトランザクション・サーバ関数の本体は、commit_work()呼び出しで終えなければなりません。この関数はパラメータをとりません。この関数は、個々の DB に対して、begin_work()が呼び出されてから受け取った SQL を実行(commit)するように通知します。

begin_work()と commit_work()の間に、トランザクションを作成する個々の SQL ステートメント呼出関数を書き込んでください。各関数の戻り値をチェックして、SQL の実行に成功したかどうかを確認してください。関数が動作していないことを値が示す場合は、abort_work()を呼び出して、それまでの関数呼び出しを全てキャンセルすることができます。この関数はパラメータをとりません。この関数は、begin_work()関数を呼び出してからの DB 操作を全てキャンセルします。begin_work()を実行して、個々の DB をロールバック可能な状態にしているため、SQL ステートメント呼出関数は、commit_work()関数を呼び出すまで実行されません。

SQL ファイルから生成された C コード(*name1*.c, *name2*.c, ...) または *server*_funcs.h ヘッダファイルを調べて、SQL ステートメント呼出関数の呼び出し方を決めることができます。これらのルーチンは、パラメータの順番とそのデータ型を示します。次の例で、当座預金口座から、異なる RDBMS 下の普通預金口座に資金を移すトランザクションを説明します。トランザクションでは、口座が両方とも同じ DB にある場合も、このコードは動作します。

生成された Makefile を使用してサーバをコンパイルするには、トランザクション・サーバコード *server*.c を呼び出す必要があります。ここで、*server* は **TPMake** に与えるサーバ名です。

```
#include <transac.h>
#include "bank_funcs.h"
long  xfer2savings(int  amount,char  *chk_act,char  *sav_act,
char **message)
{
   int i;

    long out_rows;
```

```
        long *balance;

        *message = (char *)dce_malloc(256);

        if(begin_work("xfer2savings")!=DCPSUCCESS) {
            strcpy(*message, "BEGIN_WORK failed");
            return(0);
        }

        if(chk_balance(chk_act, &out_rows, &balance) !=1) {
            abort_work();
             dce_dbgwrite(DCE_LOG_ERROR,"abort_work:
                     chk_balance chk_act=[%s]\n", chk_act);
            strcpy(*message, "chk_balance failed");
            return(0);
        }

        if(balance[0] < amount){
            dce_dbgwrite(DCE_LOG_ERROR,"Sorry,
                                        overdrawn!\n");
            strcpy(*message, "Sorry, overdrawn");
            abort_work();
            return(0);
        }

        if(withdraw_check(amount,chk_act) != 1) {
             dce_dbgwrite(DCE_LOG_ERROR,"Sorry,could          not
                             withdraw check!\n");
             strcpy(*message,"Sorry,could not withdraw check");
            abort_work();
            return(0);
        }
        if(deposit_check(amount,sav_act) != 1) {
            dce_dbgwrite(DCE_LOG_ERROR,"Sorry, could not deposit
    check!\n");
             strcpy(*message,"Sorry,could not deposit check");
            abort_work();
            return(0);
        }

        commit_work();
        dce_dbgwrite(DCE_LOG_DEBUG, "Transaction
                                Successfully completed!\n");
         strcpy(*message,"Transaction Successfully completed!");
        return(1);

    }
```

エラーのロギングは自動的に行われます。**TPMake** が生成したコードは、トランザクション障害を記録するロギングルーチンを含みます。

サーバ・スケルトンの生成

生成された Makefile を使用しない(または、ユーザが作成した Makefile の中で **RPCMake** を呼び出さない)場合、IDL ファイルを **RPCMake** ユーティリティに渡して、サーバ・スケルトンを生成してください。次のシンタックスを使用します。

```
> rpcmake -d interface.def [-size sizefile.size] -s c -i
interface:interface_init -e interface:interface_end
```

ここで、-d には IDL ファイル名、 -size には COBOL サイズファイル、-s にはサーバ・スケルトンの言語、-i オプションは、サーバ起動時に最初に呼び出さなければならないサーバのルーチン名を指定します。このオプションの詳細については、『リファレンス』の「RPCMake」を参照してください。

RPCMake GUI では、-i または e の引数は使用できないため、コマンドラインを使用してください。

最終的なスケルトンファイルは *interface*_s.c、または *interface*_s.cbl という名前になります。

サーバの実行ファイルのコンパイル

TPMake が Makefile を生成する場合

Makefile を使用してコンパイルを成功するためには、次の前提条件を満たさなければなりません。

- 開発者が作成したトランザクション・サーバファイルの名前は、*interface*.c または *interface*.cbl としなければならない。ここで、*interface* は **TPMake** 使用時に指定したサーバ名。
- IDL ファイルの名前は *interface*.def としなければならない。
- サーバ・スケルトンの名前は *interface*_s.c または *interface*_s.cbl として RPCMake により生成されます。
- 実行ファイルの名前は *interface* としなければならない。
- リソースマッピングファイルの名前は *interface*.res としなければならない。
- 静的 SQL では、ESQL ファイルの名前は *query_file*_static.<RDBMS_suffix> としなければならない。(表 3.1)

注:ODEDIR 環境変数が設定されていることを確認してください。

自分で Makefile を作成する場合

次のファイルを実行ファイルとともにコンパイルしなければなりません。

- SQL ファイルに対応して生成された SQL ステートメント呼出ファイル(*name1*.c、*name2*.c など)
- TPMake で生成されたトランザクション制御ファイル (*transac*.c)
- **RPCMake** が生成したサーバ・スケルトン (*interface*_s.c)
- 開発者が作成したトランザクション・サーバファイル(*interface*.c または *interface*.cbl)

また、コンパイルを行うマシンにインストールされている次の Nextra ライブラリをリンクしなければなりません。

- librpc.*ext* (RPC ランタイム)
- adcp_*dbms*.o (オブジェクトファイル)

最終的には、固有の RDBMS ライブラリもリンクしなければなりません。(UNIX のみ)

```
getplatform RDBMS LIBS
```

RDBMS は ORA などで、RDBMS がリンクされる適切なライブラリになります。

例

C 言語用の Makefile の例を示します。このサーバは、Oracle の DB にアクセスします。この場合、**TPMake** により生成された C ファイルは、savings.c、checking.c、bank_funcs.h、transac.c です。bank.c には、開発者が定義したトランザクション・サーバ関数が含まれています。Makefile を使用する場合は、開発者が記述した C コード *interface*.c を必ず呼び出してください。使用しない場合には、Makefile を編集する必要があります。

```
#####################################################
#
# Makefile for the "bank" functionality server
#
# This file was generated by tpmake
#
# If you have a special compiler, or special options
```

```
# to compile ANSI files, change $ODEDIR/bin/getplatform.
#    (e.g. CC = cc -Aa -D_HPUX_SOURCE)
# `getplatform` is a shell utility used for portability.
# Refer to the documentation for details.
#
# If you have other object files, add them to
# the OBJS variable, and add targets at the end of the
# file.
#
#####################################################

CC = `getplatform cc`
DBMSLIBS = `getplatform RDBMSLIBS`

LIBS =  `getplatform libdir` `getplatform adaplib`
`getplatform lib` -L$(RDBMS_HOME)/lib $(DBMSLIBS)
INCL = `getplatform inc`
OBJS = bank.o bank_s.o savings.o checking.o transac.o

bank: $(OBJS)
        $(CC) -o bank $(OBJS) $(LIBS)

bank.o: bank.c
        $(CC) -c bank.c $(INCL)

bank_s.o: bank_s.c
        $(CC) -c bank_s.c $(INCL)

bank_s.c: bank.def
        rpcmake -d bank.def -s c -y -i bank:bank_init -e
bank:bank_end

savings.o: savings.c
        $(CC) -c savings.c $(INCL)

checking.o: checking.c
        $(CC) -c checking.c $(INCL)

transac.o: transac.c
        $(CC) -c transac.c $(INCL)

realclean:
        rm -f *.o *_static.*  bank bank_s.c bank_funcs.h
savings.c checking.c
        touch *.qfile bank.res

clean:

      rm -f *.o bank bank_s.c
```

注:**TPMake** は、特定のアプリケーションに合わせて、Makefile を調整するので、実際に生成された Makefile は、上記とは異なる個所があります。

環境ファイルの作成

『リファレンス』の「環境ファイルの作成」の説明に沿って、環境ファイルを作成、あるいは編集してください。

これでトランザクション・サーバが実行できます。

トランザクション・サーバのテスト

デバッグ

『サーバ開発者ガイド』の「サーバのテスト」のデバッグ方法に従って実行してください。

1. 『サーバ開発者ガイド』の指示に沿って、Perl クライアント・スタブを生成し、ブローカを起動します。

2. トランザクション・サーバを起動します。

トランザクション・サーバを起動するコマンドは次のようになります。

```
> server_name [-a auditfile] -e env_file
```

もちろん、-log や -s といった標準のサーバオプションも使用できます。

3. 『サーバ開発者ガイド』の指示に沿って、RPCDebug ユーティリティを起動します。

関数を順次選択し、必要な値を入力してください。「実行」ボタンをクリックし、返されたデータが適切なものであるか確認してください。

RPCDebug のコマンドラインユーティリティを使用できます。『リファレンス』の「rpctest」を参照してください。

サーバコードの配布

Nextra 運用版パッケージがインストールされており、必要な環境変数が設定されていることを確認してください。トランザクション・サーバの実行ファイル、およびその環境ファイルを、トランザクション・サーバが実行されるマシン上へ移動してください。この時点でクライアント・スタブが生成されている場合は、クライアントのインタフェースが置かれるプラットフォームに、クライアント・スタブを移動してください。クライアント・スタブが生成されていない場合は、『クライアント開発者ガイド』を参照してください。

特殊機能の使用

ここでは、次の 4 つの追加機能について説明します。

- 任意の DB アクセス
- ストアド・プロシージャ
- DB カーソル (デディケイテッド・サーバのみ)
- サーバトランザクションの自動分散 2 フェーズ・コミット

任意の DB のアクセス

Nextra では、2 つの SQL Server DB であろうと、3 つの Oracle DB であろうと、指定された DB のセットにアクセスすることができます。1 つの DB にマルチ接続することもできます。

既に開発者が記述したトランザクション・サーバファイルを修正する必要はありません。sql_prepare_interface()と sql_rows_interface()関数は全く同じように動作し、以前のリリースと同じ関数定義のフォーマットを示します。しかし、これらはバックワード互換性のためだけのものです。新しいサーバを作成する場合は、sql_connect_*interface*()と sql_set_max_rows_*interface*()を使用してください。

各 RDBMS 固有の説明については、以下の見出しを参照してください。

Oracle

デフォルトでない、または複数の Oracle データベースにアクセスするには、関連する全てのマシンが(1 つしかなくても)Oracle SQL-Net を実行していなければなりません。リソースファイルも変更しなければなりません。DB 名フィールドで、DB 名とエイリアスの間にパイプ(|)デリミタを入れ、SQL-Net のエイリアスをリソースファイル内のエントリ

を固有に識別する DB 名に加えます。たとえば、tnsnames.ora ファイル中のエイリアスエントリは次のようになります。

```
jakesdb =

(DESCRIPTION =

(ADDRESS_LIST =

(ADDRESS =

(COMMUNITY = TCP_1)
(PROTOCOL = TCP)
(HOST = netserver)
(PORT = 1521)

  )

)
(CONNECT_DATA =(SID = dbjake)
)

  )
```

エイリアスと DB 名は同じである必要はありません。リソースファイルのエントリは次のようになります。

```
#tag file    db_name        RDBMS  login passwd
1    tps.sql dbjake|jakesdb oracle Jake  di85jDU
```

Windows 上の SQL Server

デフォルト(通常はローカル)サーバを除く全ての SQL Server にアクセスするには、環境変数 DSQUERY を設定しなければなりません。

この SQL Server は、「SQL クライアント設定ユーティリティ」を使用して定義しなければなりません。これは、SQL Server プログラムグループにあります。SQL クライアント設定ダイアログボックスで、次のステップを実行します。

1. 「拡張」タブをクリックします。

2. 「サーバー」で、サーバを識別するためのエイリアス名を入力します。

   これは、定義された SQL Server 間でユニークであれば、どんな名前でもかまいません。

3.「DLL 名」で、DLL を常に dbmssocn にします。

4.「コネクション」で、指定したエイリアスと関連づける SQL サーバのホスト名とポートを入力します。

適切なシンタックスは、hostname, portnum です。

5. 変更を保存して、ユーティリティを終了します。

DB アクセス・サーバを起動するときに、DSQUERY 環境変数で指定された SQL Server を通じて使用できる DB を、-d オプションを使用して指定しなければなりません。

たとえば、デフォルトの SQL Server の下で実行されている DB bigdb にアクセスするサーバを起動するとします。

```
> sql_start -e data.env -s server_name -d bigdb
```

別の SQL Server の下の DB にアクセスするには、DSQUERY をそのサーバの名前に設定し、同じような起動コマンドを指定します。

```
> sql_start -e data.env -s server_name -d tech
```

ストアド・プロシージャの使用

Nextra ツールは、Oracle のためのストアド・プロシージャの呼び出しをサポートします。

Nextra では、Oracle ストアド・プロシージャからの出力は返されません。

DB アクセス・サーバが実行する他の関数と同じように、ストアド・プロシージャを呼び出す関数は、SQL ファイル中で定義されていなければなりません。

たとえば、次のストアド・プロシージャを Oracle で定義するものとします。

```
create procedure listname (person_age in integer, raise in real) as

BEGIN

update namelist
set salary=salary * raise where age > person_age;

END listname;
```

次の方法で、SQL ファイル中でプロシージャを呼び出すことができます。

```
update_namelist:
{BEGIN listname($a, $b)\; END\;\;};
```

参照

『リファレンス』の「SQL ファイル」

デディケイテッド・サーバにおけるカーソルの使用

デディケイテッド・サーバでは、カーソルを使用することができます。トランザクション・サーバが、環境ファイル属性 DCE_DEDICATED に 0 より大きな値に設定された状態で起動されると、接続するクライアントごとにサーバのコピーが作成され、各サーバは 1 つのクライアントのために RPC を実行することになります。通常のサーバと比べて、この種のサーバは「ステータス(Status)」を維持しています。つまり、クライアント要求と対応する応答を適切に記録し、応答を調整できるということです。トランザクション・サーバでカーソルを使用するには、デディケイテッド・サーバにしなければいけません。

カーソルとアドバンスド DB コネクティビティサーバ

TPMake が生成した C コードには、 デディケイテッド・トランザクション・サーバが使う 2 つの関数、sql_connect_*interface*() と sql_set_max_rows_*interface*()が含まれます。これらの関数は、デディケイテッド・トランザクション・サーバでのみ使用されるもので、通常のトランザクション・サーバから呼び出されることはありません。クライアントが 1 つ以上のデディケイテッド・トランザクション・サーバにアクセスする場合に、別のサーバの同じような関数と区別するために、関数名にはインタフェース名が付けられます。

トランザクション・サーバのセッションの開始

リソースファイルで指定した RDBMS の数に基づき、**TPMake** はサーバコードの一部として sql_connect_*interface*()関数を生成します。この関数は、1 つの RDBMS アクセスに対して 4 つのパラメータが必要となります。

```
long sql_connect_interface (
```

char *dbname1, char *dbstr1, char *login1,
char *passwd1,

char *dbname2, char *dbstr2, char *login2,
char *passwd2,
...)

この関数呼び出しを受け取ると、サーバは指定した RDBMS に対するログインを、指定したユーザ名とパスワードを使用して初期化します。サーバは、リソースファイルに指定された DB にアクセスします。この関数は正常終了時には値 DCPSUCCESS を返し、エラー時には適切なエラーコードを返します。

sql_connect_*interface*()が、最初に呼び出された RPC でない場合、デディケイテッド・サーバは、各 RDBMS のリソースファイル中のログイン名とパスワードフィールドを使用して、通常のサーバのように DB にログインします。

バックワード互換性

関数 sql_prepare_*interface*()を使用するトランザクションコードがある場合は、以前と同様にコンパイルして実行することができます。しかし、新しいサーバを作成する場合は、すでに説明したように、sql_connect_*interface*()関数を使用してください。

ユーザがトランザクションを定義するので、トランザクション・サーバのためにユーザ自身の IDL ファイルを作成しなければなりません。**TPMake** が生成した C コードを調べて、sql_connect_*interface*()と sql_set_max_rows_*interface*()を見つけてください。対応するように、この 2 つの関数の定義を作成してください。

カーソル値の設定

カーソル機能を使用するには、クライアントは関数 sql_set_max_rows_*interface*()を呼び出さなければなりません。

long sql_set_max_rows_*interface* (long *maxrows*)

この関数は引数を 1 つとります。1 回に検索する行の最大数です。正常終了時は DCPSUCCESS が、エラー発生時は負の値が返されます。*maxrows* が負の値を含む場合は、DCPERROR が返されます。

たとえば、照会で 500 行を選択し、値 50 で sql_set_max_rows_*interface*()を呼び出すと、照会が最初に行われたときの先頭の 50 行のみが返されます。同じ照会を続けて行うと、次の 50 行が返されます。

デディケイテッド・サーバのみでサポートされる理由

同じサーバに対する同じ照会を別のクライアントが行うことは避けられないため、通常のサーバではカーソルは使用できません。同じ照会を行うということは、オリジナルのクライアントが次に照会を行うときに、カーソルがはるか先に設定されるということです。あるいは、別のクライアントが異なった照会を行って、最初のクライアントが全体を検索し終える前に、カーソルをゼロに戻してしまう可能性もあります。デディケイテッド・サーバがサービスするのは1つのクライアントだけなので、別のクライアントが照会を妨げることはありません。

sql_set_max_rows_*interface*()を一度も呼び出さない場合、または sql_set_max_rows_*interface*(0)が呼び出された場合は、サーバは、最初の照会で選択された全ての行を返します。同様に、選択された行数よりも *maxrows* が大きい場合は、最初の照会で、選択された全ての行が返されます。最初の照会が選択された全ての行を返す前に、2 回目の照会が呼び出されると、2 回目の最初の *maxrows* 行が検索されて、最初の照会のカーソルは 0 にリセットされます。一度に返される行数を変更するには、新しい値で sql_set_max_rows_*interface*()だけを呼び出す必要があります。

たとえば、2 つの RPC、 *rpc1* と *rpc2* があるとします。両方とも合計で 20 行を返し、sql_set_max_rows_*interface*(4)が呼び出されているとします。*rpc1* に対する最初の呼び出しは行 A1-A4 を返します。*rpc1* に対する次の呼び出しは A5-A8 を返します。*rpc2* に対する呼び出しは B1-B4 を返します。rpc1 に対する次の呼び出しでは A1-A4 を返します。sql_set_max_rows_*interface*(8)を呼び出してから、*rpc1* を再度呼び出すと、A5-A12 が返されます。

注:sql_set_max_rows_*interface*(4)を使って rpc1 が 6 回呼び出されると、5 回目の呼び出しは 16-20 行を返しますが、6 回目の呼び出しでは 0 が返されます。7 回目の呼び出しは行 1-5 を返します。

全部で 19 行しかない場合は、5 回目の呼び出しは行 16-19 を返し、6 回目の呼び出しは行 1-5 を返します。

一連の照会の最後のものが、全選択を完了するために十分な行を正確に返した場合、照会の次の実行では 0 行が返されます。このようになっていないと、全選択が返されたときに、行数が必ずしも返されないという状態になります。この処理では、照会が *maxrows* 行より少ない行を返す場合、その照会では全選択が返されます。

カーソルとバリアブル・ネームド DB アクセス・サーバ

バリアブル・ネームドインタフェースでは、全ての RPC についての特別パラメータが自動的にスタブに含まれます。このパラメータはパラメータリストの最初のパラメータになり、関数呼び出しが送られるインタフェースを指定します。

クライアントが、2 つ以上のバリアブル・ネームド・デディケイテッド DB アクセス・サーバにアクセスする場合、カーソルの機能が、異なる名前を付けた関数に複製されます(1 スタブにつき 1 回)。しかし、これらの関数それぞれが、パラメータとしてインタフェース名をとるので、sql_set_max_rows_*interface*()関数を使用して、サーバのいずれかにカーソルを設定できます。実際に、必要であれば、指定したクライアントがアクセスする IDL ファイルのセットから、1 つだけを残して、余分な関数定義を削除することもできます。sql_connect_*interface*()が必要とするパラメータ数は、このタイプのサーバではインタフェース間で異なるので、IDL ファイルから、定義を削除しないでください。また、この関数をバリアブル・ネームド・サーバで呼び出すときは必ず、interface パラメータの値が関数名の *interface* の部分に対応するようにしてください。

参照

『運用／設定ガイド』の「デディケイテッド・サーバ」。
Nextra ライブラリ関数:
『リファレンス』の「sql_set_max_rows_()」、「sql_prepare_()」、「sql_connect_()」。

第 4 章 Nextra トランザクション制御関数

ここで説明する関数は、トランザクション・サーバで使用可能です。

以下の C 関数は、トランザクション・サーバで使用するために **TPMake** が生成したファイル transac.c で定義されます。各関数についてプロトタイプ、機能、パラメータ、戻り値を説明します。

これらの関数は、トランザクション・サーバのファンクショナリティを作成する際に、開発者が使用するものです。トランザクション関数は、SQL ステートメントをトランザクションにグループ化するときに、トランザクションのバウンダリを引くために使用します。デディケイテッド・サーバ関数は、デディケイテッド・トランザクション・サーバを使用するときに呼び出されます。

デディケイテッド・サーバ関数については、Nextra API の sql_connect_( ), sql_prepare_( ), sql_set_max_rows_( )を参照してください。

abort_work( )

```
int abort_work();
```

この関数はカレントトランザクションを停止し、トランザクションの起動前の状態にデータをロールバックします。トランザクションルーチンからの終了は実行されません。

パラメータ	in/out	説明
なし		

戻り値	意味
< 1	アボートできなかった RDBMS から返されるエラーコード。
DCPSUCCESS	正常終了。
DBNOTRANSAC	abort_work()呼び出しときに、トランザクションが起動されていなかった。

begin_work( )

```
int begin_work(char * TransName);
```

この関数は、トランザクションの起動を確立します。これは各トランザクションの最初の SQL ステートメント呼出関数より前に置きます。

パラメータ	in/out	説明
TransName	in	起動するトランザクション名。

戻り値	意味
< 1	RDBMS は起動できなかった。
DCPSUCCESS	正常終了。
DBCOMMITPEND	以前のトランザクションがコミット、またはアボートされていません。

commit_work( )

```
int commit_work();
```

この関数は、使用可能な各 DB で commit を実行します。トランザクションにアクセスされなかった DB をコミットできなかった場合、警告が通知されます。Oracle DB では、リソースファイルと SQL ファイルを適切にコーディングすれば、この関数は 2 フェーズ・コミット制御を行います。SQL Server DB では、デフォルトでこの関数は 2 フェーズ・コミット制御を行います。

パラメータ	in/out	説明
なし		

戻り値	意味
< 1	トランザクションで起動された RDBMS はコミットできなかった。
DCPSUCCESS	正常終了。
DBNOTRANSAC	起動されているトランザクションがありません。

ご注意

商標権に関する注意

Nextra 製品は、全て Inspire International Inc. の商標または登録商標です。その他記載のブランドおよび製品名は、該当する会社の商標または登録商標です。

著作権に関する注意

Nextra
トランザクション制御ガイド

2015 年 11 月 08 日　v6.5 1st Edition
2011 年 9 月 15 日　v6 1st Edition
2010 年 9 月 13 日　Reviewed
2008 年 8 月 12 日　v5 2nd Edition
2007 年 4 月 18 日　第 3 版発行
2004 年 7 月 19 日　第 2 版発行
2003 年 4 月 18 日　初版発行

著者　Inspire International Inc.