HITACHI
Inspire the Next

DABroker for C++

手引・文法書

3020-6-032-80

■ 対象製品

P-F2463-23541 DABroker for C++ 02-12（適用 OS：Windows2000，Windows XP Professional，Windows XP Professional x64 Edition，Windows Server 2003，Windows Server 2003 x64 Edition，Windows Server 2003 R2，Windows Server 2003 R2 x64 Edition，Windows Server 2008，Windows Server 2008 R2，Windows Server 2012，Windows Server 2012 R2，Windows Vista，Windows 7，Windows 8，Windows 8.1）

P-F1B63-23511 DABroker for C++ 02-07（適用 OS：HP-UX 11.0，HP-UX 11i，HP-UX 11i V2(PA-RISC)，HP-UX 11i V3(PA-RISC)）

P-F1J63-24511 DABroker for C++ 02-08（適用 OS：HP-UX 11i V2.0(IPF)，HP-UX 11i V3.0(IPF)）

P-F1M63-22511 DABroker for C++ 02-07（適用 OS：AIX 5L V5.1，AIX 5L V5.2，AIX 5L V5.3，AIX V6.1，AIX V7.1）

P-F8163-23511 DABroker for C++ 02-10（適用 OS：Red Hat Enterprise Linux AS 4 (x86)，Red Hat Enterprise Linux ES 4(x86)，Red Hat Enterprise Linux AS 4 (AMD64 & Intel EM64T)，Red Hat Enterprise Linux ES 4(AMD64 & Intel EM64T)，Red Hat Enterprise Linux 5 Advanced Platform (x86)，Red Hat Enterprise Linux 5 (x86)，Red Hat Enterprise Linux 5 Advanced Platform (AMD/Intel 64)，Red Hat Enterprise Linux 5 (AMD/Intel 64)，Red Hat Enterprise Linux Server 6(32-Bit x86)，Red Hat Enterprise Linux Server 6(64-Bit x86_64)）

これらのプログラムプロダクトのほかにもこのマニュアルをご利用になれる場合があります。詳細は「リリースノート」でご確認ください。

■ 輸出時の注意

本製品を輸出される場合には、外国為替及び外国貿易法の規制並びに米国輸出管理規則など外国の輸出関連法規をご確認の上、必要な手続きをお取りください。

なお、不明な場合は、弊社担当営業にお問い合わせください。

■ 商標類

Adaptive Server は，Sybase, Inc.の商標です。

CORBA は，Object Management Group が提唱する分散処理環境アーキテクチャの名称です。

HP-UX は，Hewlett-Packard Development Company, L.P.のオペレーティングシステムの名称です。

IBM，AIX は，世界の多くの国で登録された International Business Machines Corporation の商標です。

IBM，VisualAge は，世界の多くの国で登録された International Business Machines Corporation の商標です。

Itanium は，アメリカ合衆国およびその他の国における Intel Corporation の商標です。

Linux は，Linus Torvalds 氏の日本およびその他の国における登録商標または商標です。

Microsoft 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。

ODBC は，米国 Microsoft Corporation が提唱するデータベースアクセス機構です。

Oracle と Java は，Oracle Corporation 及びその子会社，関連会社の米国及びその他の国における登録商標です。

Pentium は，アメリカ合衆国およびその他の国における Intel Corporation の商標です。

Red Hat は，米国およびその他の国で Red Hat, Inc. の登録商標もしくは商標です。

Sybase SQL Anywhere は，Sybase，Inc.の商標です。

Visual C++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。

Windows 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。

Windows Server 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。

X/Open は，The Open Group の英国ならびに他の国における登録商標です。

その他記載の会社名，製品名は，それぞれの会社の商標もしくは登録商標です。

■ 発行

2015 年 4 月　3020-6-032-80

■ 著作権

All Rights Reserved. Copyright (C) 1998, 2015, Hitachi, Ltd.

変更内容

変更内容（3020-6-032-80） DABroker for C++ 02-12

追加・変更内容	変更個所
HiRDB の Binary 属性をサポートしました。	4.4，5.8，5.9，7.1.3，7.1.6，付録 B

変更内容（3020-6-032-80） DABroker for C++ 02-10

追加・変更内容	変更個所
Oracle 12c をサポートしました。	1.1.2
Red Hat Enterprise Linux Server 6 をサポートしました。	3.2.3，6.1，7.1.5，付録 A.1
Windows 7，Windows 8，Windows Server 2012 をサポートしました。	3.2.4，3.2.5，7.1.5，付録 A.2

単なる誤字・脱字などはお断りなく訂正しました。

変更内容（3020-6-032-70） P-F2463-22541 DABroker for C++ 02-07，P-F1J63-24511 DABroker for C++ 02-07，P-F9D63-22511 DABroker for C++ 02-07，P-F9D63-23511 DABroker for C++ 02-07，P-F1M63-22511 DABroker for C++ 02-07

追加・変更内容
アクセスできる DBMS に Oracle10g を追加しました。
セットアップコマンド実行での注意を追加しました。

変更内容（3020-6-032-70） P-F1J63-24511 DABroker for C++ 02-07

追加・変更内容
HP-UX(IPF)の場合のコンパイルオプションとリンクオプションを追加しました。
DABroker for C++ が提供するファイルを追加しました。 • libdabcpp32.so(HP-UX(IPF)版)

変更内容（3020-6-032-60） P-F1M63-22511 DABroker for C++ 02-05，P-F1M63-22711 Cosminexus DABroker for C++ 02-05

追加・変更内容
シングルスレッドに対応する機能を追加しました。
SQL Server2000 へのアクセスをサポートしました。
OTS インタフェースを使ってトランザクション制御をする場合の記述を削除しました。

変更内容（3020-6-032-60） P-F1B63-23511 DABroker for C++ 02-04，P-F1B63-23711 Cosminexus DABroker for C++ 02-04

追加・変更内容
検索したレコードに対するロック方法で TYPE_EXCLUSIVE2 へのアクセスをサポートしました。

変更内容（3020-6-032-60） P-F1L63-23511 DABroker for C++ 02-04

追加・変更内容
P-F1B63-23511 02-04 と同等の機能を持つ P-F1L63-23511 を発行しました。

変更内容（3020-6-032-60） P-F2463-22541 DABroker for C++ 02-03，P-F2463-22741 Cosminexus DABroker for C++ 02-03

追加・変更内容
HiRDB クライアント環境定義の環境変数を指定できる機能を追加しました。
Oracle9i へのアクセスをサポートしました。

はじめに

このマニュアルは，次に示すプログラムプロダクトが提供するクラスライブラリを使ったアプリケーションの作成方法と，各クラスライブラリの文法について説明したものです。

- P-F2463-23541　DABroker for C++
- P-F1B63-23511　DABroker for C++
- P-F1J63-24511　DABroker for C++
- P-F1M63-22511　DABroker for C++
- P-F8163-23511　DABroker for C++

■ 対象読者

DABroker for C++を使って，データベースをアクセスするサーバアプリケーションを開発する方を対象としています。

また，C++言語，及び使用する DBMS についてある程度の知識をお持ちの方を対象としています。

■ マニュアルの構成

このマニュアルは，次に示す章と付録から構成されています。

第１章 DABroker for C++の概要

DABroker for C++の役割・位置付けや特長について説明しています。また，どのようなデータベースアクセスができるサーバアプリケーションが作成できるかについても説明しています。

第２章 データベースアクセス

簡易版，詳細版データベースアクセス方法について，コーディング例を使って説明しています。

第３章 アプリケーションの作成

DABroker for C++クラスライブラリを利用した，アプリケーションの作成方法について説明しています。

第４章 簡易版関数詳細

簡易版のクラスライブラリごとに，プロパティやメソッドを詳細に説明しています。

第５章 詳細版関数詳細

詳細版のクラスライブラリごとに，プロパティやメソッドを詳細に説明しています。

第６章 共通関数詳細

簡易版と詳細版で共通のクラスライブラリについて，プロパティやメソッドを詳細に説明しています。

第７章 データ型

C++クラスライブラリのインタフェースで使うデータ型について説明しています。

第８章 トラブルシューティング

エラー発生時のトレースログの採取方法や，出力されるメッセージの意味について説明しています。

付録 A セットアップ

DABroker for C++の環境設定について説明しています。

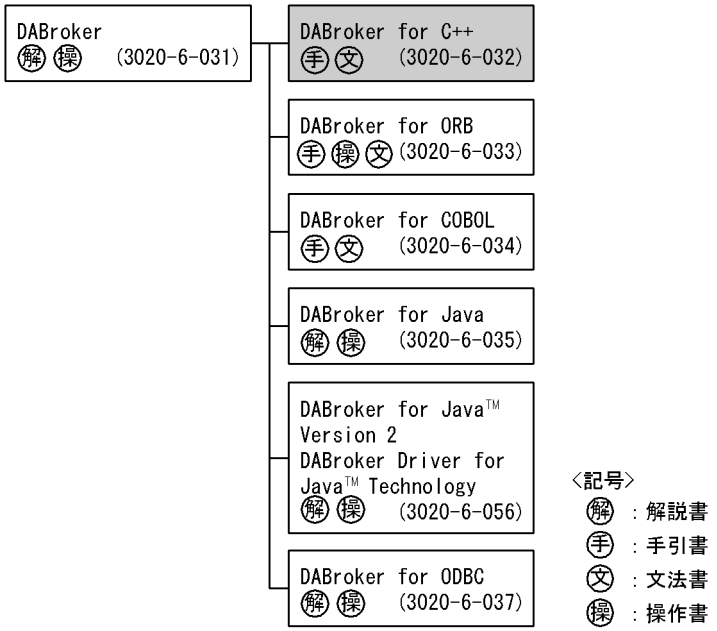
付録 B 用語解説

DABroker で使用する用語について説明しています。

■ 関連マニュアル

このマニュアルの関連マニュアルを次に示します。また，これ以外にも使用する DBMS に応じてその製品のマニュアルを参照してください。

DABroker のマニュアル



DBMS のマニュアル

- 使用する DBMS が HiRDB Version 7 の場合

スケーラブルデータベースサーバ HiRDB Version 7 システム運用ガイド（UNIX(R)用）(3000-6-274)
スケーラブルデータベースサーバ HiRDB Version 7 システム運用ガイド（Windows(R)用）(3020-6-274)
スケーラブルデータベースサーバ HiRDB Version 7 UAP 開発ガイド（UNIX(R)/Windows(R)用）(3000-6-276)
スケーラブルデータベースサーバ HiRDB Version 7 SQL リファレンス（UNIX(R)/Windows(R)用）(3000-6-277)

- 使用する DBMS が HiRDB Version 8 の場合

スケーラブルデータベースサーバ HiRDB Version 8 システム運用ガイド（UNIX(R)用）(3000-6-354)
スケーラブルデータベースサーバ HiRDB Version 8 システム運用ガイド（Windows(R)用）(3020-6-354)
スケーラブルデータベースサーバ HiRDB Version 8 UAP 開発ガイド（3020-6-356）
スケーラブルデータベースサーバ HiRDB Version 8 SQL リファレンス（3020-6-357）

- 使用する DBMS が HiRDB Version 9 の場合

スケーラブルデータベースサーバ HiRDB Version 9 システム運用ガイド（UNIX(R)用）(3000-6-454)
スケーラブルデータベースサーバ HiRDB Version 9 システム運用ガイド（Windows(R)用）(3020-6-454)
スケーラブルデータベースサーバ HiRDB Version 9 UAP 開発ガイド（3020-6-456）
スケーラブルデータベースサーバ HiRDB Version 8 SQL リファレンス（3020-6-457）

- 使用する DBMS が VOS3 XDM/RD E2 などの場合

データマネジメントシステム XDM E2 系 解説（6190-6-620）
データマネジメントシステム XDM E2 系 システム定義（6190-6-625）

Database Connection Server（6190-6-648）

- 使用する DBMS が SQL/K の場合

ファイル運用（650-3-151）
SQL/K（650-3-241）
Database Connection Server（650-3-243）

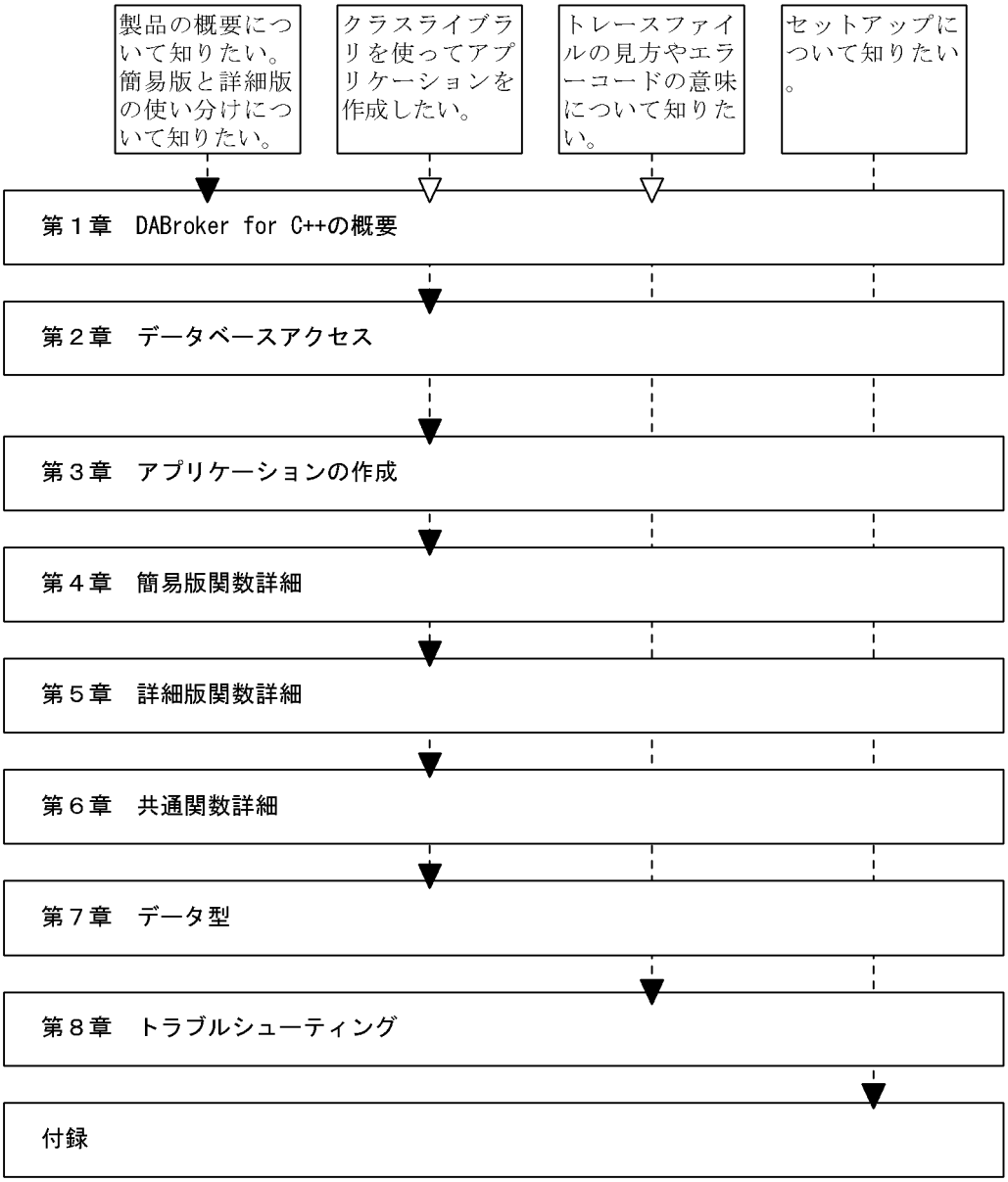
- 使用する DBMS が上記以外の場合

使用する DBMS のマニュアルを参照してください。

TPBroker のマニュアル（TPBroker の OTS 機能を利用する場合）

トランザクショナル分散オブジェクト基盤 TPBroker ユーザーズガイド(3000-3-660)
トランザクショナル分散オブジェクト基盤 TPBroker for Java OTS ガイド（3000-3-661）

■ 読書手順



■ このマニュアルでの表記

このマニュアルでは，製品名称，及び名称について次のように表記します。

製品名称又は名称	表記	
AIX 5L	AIX	UNIX
AIX V6.1		
AIX V7.1		

製品名称又は名称	表記		
HP-UX 11.0	HP-UX(PA-RISC)	HP-UX	UNIX
HP-UX 11i			
HP-UX 11i V2(PA-RISC)			
HP-UX 11i V3(PA-RISC)			
HP-UX 11i V2(IPF)	HP-UX(IPF)		
HP-UX 11i V3(IPF)			
Red Hat Enterprise Linux AS4.0	Red Hat Linux		
Red Hat Enterprise Linux ES4.0			
Red Hat Enterprise Linux 5			
Red Hat Enterprise Linux 5 Advanced Platform			
Red Hat Enterprise Linux Server 6			
Common Object Request Broker Architecture	CORBA		
Database Management System	DBMS		
HiRDB Version 7	HiRDB		
HiRDB Version 8			
HiRDB Version 9			
Itanium(R) Processor Family	IPF		
Microsoft(R) SQL Server	SQL Server		
Microsoft(R) Visual C++(R) 5.0	Visual C++		
Microsoft(R) Visual C++(R) 6.0			
Microsoft(R) Visual C++(R) .NET 2003			
Microsoft(R) Visual Studio(R) 2010	Visual Studio		
Microsoft(R) Visual Studio(R) 2013			
Microsoft(R) Windows(R) XP	Windows XP		Windows
Microsoft(R) Windows Server(R) 2003, Standard Edition	Windows Server 2003		
Microsoft(R) Windows Server(R) 2003, Enterprise Edition			
Microsoft(R) Windows Server(R) 2003, Datacenter Edition			
Microsoft(R) Windows Server(R) 2003 R2, Standard Edition			
Microsoft(R) Windows Server(R) 2003 R2, Enterprise Edition			
Microsoft(R) Windows Server(R) 2003 R2, Datacenter Edition			
Microsoft(R) Windows Vista(R) Business	Windows Vista		

製品名称又は名称	表記	
Microsoft(R) Windows Vista(R) Enterprise	Windows Vista	Windows
Microsoft(R) Windows Vista(R) Ultimate		
Microsoft(R) Windows 7(R) Professional	Windows 7	
Microsoft(R) Windows 7(R) Enterprise		
Microsoft(R) Windows 7(R) Ultimate		
Windows 8(R) Pro	Windows 8	
Windows 8(R) Enterprise		
Windows 8.1(R) Pro	Windows 8.1	
Windows 8.1(R) Enterprise		
Microsoft(R) Windows Server(R) 2008 Standard	Windows Server 2008	
Microsoft(R) Windows Server(R) 2008 Enterprise		
Microsoft(R) Windows Server(R) 2008 Datacenter		
Microsoft(R) Windows Server(R) R2 2008 Standard		
Microsoft(R) Windows Server(R) R2 2008 Enterprise		
Microsoft(R) Windows Server(R) R2 2008 Datacenter		
Microsoft(R) Windows Server(R) 2012 Standard	Windows Server 2012	
Microsoft(R) Windows Server(R) 2012 Datacenter		
Microsoft(R) Windows Server(R) 2012 R2 Standard		
Microsoft(R) Windows Server(R) 2012 R2 Datacenter		
Microsoft(R) Windows(R) 2000 Professional Operating System	Windows 2000	
Microsoft(R) Windows(R) 2000 Server Operating System		
Microsoft(R) Windows(R) 2000 Advanced Server Operating System		
Microsoft(R) Windows(R) 2000 Datacenter Server Operating System		
Open Database Connectivity	ODBC	
Operating System	OS	
Oracle 8	ORACLE	
Oracle 8i		
Oracle 9i		
Oracle 10g		
Oracle 10g R2		

製品名称又は名称	表記
Oracle 11g	ORACLE
Oracle 11g R2	
Oracle 12c	
Object Transaction Service	OTS
Sybase Adaptive Server Anywhere	Adaptive Server Anywhere
Sybase SQL Anywhere	SQL Anywhere
VOS3 ACE3 E3	ACE3
VOS3 XDM/RD E2	XDM/RD
VOS3 XDM/SD E2	XDM/SD
VOS K RDB 編成ファイル	SQL/K
HiRDB：BLOB 型	BLOB 型
HiRDB：BINARY 型	
ORACLE：LONGRAW 型	
SQL Anywhere：LONG BINARY 型	
SQL Anywhere：IMAGE 型	
Adaptive Server Anywhere：LONG BINARY 型	
Adaptive Server Anywhere：IMAGE 型	
Adaptive Server Anywhere：javaserialization,java.lang.Object	
SQL Server：IMAGE 型	

特に，それぞれの製品についての記述が必要な場合は，そのまま表記します。

■ KB（キロバイト）などの単位表記について

1KB （キロバイト），1MB （メガバイト），1GB （ギガバイト），1TB （テラバイト）はそれぞれ 1,024 バイト，$1{,}024^2$ バイト，$1{,}024^3$ バイト，$1{,}024^4$ バイトです。

目次

1 DABroker for C++の概要

この章ではDABroker for C++の役割・位置付けや特長について説明しています。また，提供するC++クラスライブラリがどのような分類で分けられるか，更にこれらのC++クラスを使って，どのようなアプリケーションを作成できるかについても説明しています。

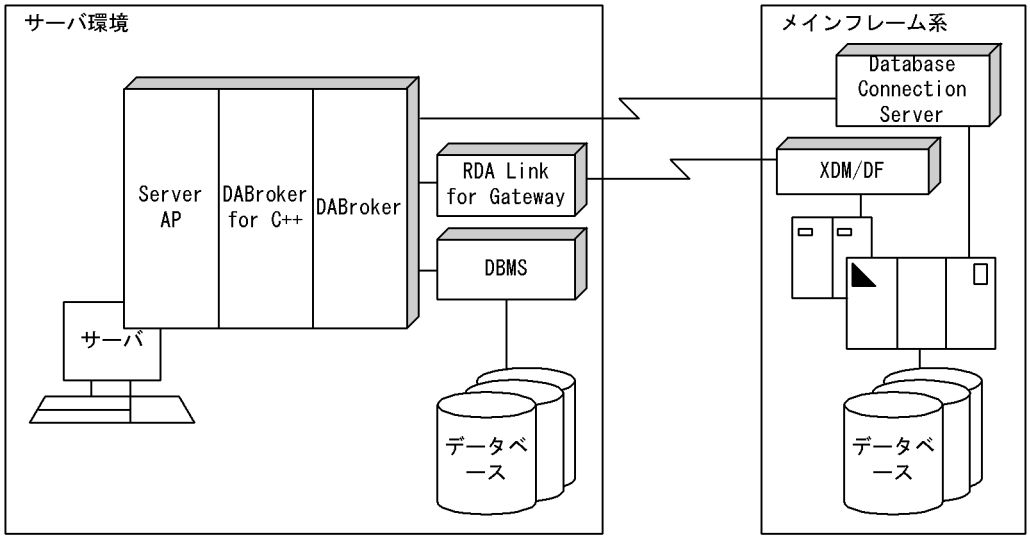
1.1　DABroker for C++の役割・位置付け

DABroker for C++は，DABroker の付加プログラムプロダクトであり，データベースにアクセスするサーバアプリケーションを作成するための C++クラスライブラリを提供します。

1.1.1　DABroker for C++の役割・位置付け

DABroker for C++の位置付けを図 1-1 に示します。

図 1-1　DABroker for C++の位置付け



DABroker for C++を利用して開発するアプリケーションは，主にサーバ上で稼働し，サーバ上のデータベース，及びメインフレームのデータベースにアクセスできます。クライアントで動作するアプリケーションでの利用も可能です。

なお，DABroker 全体の概要については，マニュアル「DABroker」を参照してください。

1.1.2　アクセスできる DBMS

アプリケーションからは，次の DBMS をアクセスできます。なお，ネットワーク経由で，ほかのマシンのデータベースも利用できます。

- HiRDB
- ORACLE

  Windows 版：Oracle 9i，Oracle 10g，Oracle10g R2，Oracle 11g，Oracle 11g R2，Oracle 12c にアクセスできます。

  Red Hat Linux 版：Oracle 9i，Oracle 10g，Oracle10g R2，Oracle 11g，Oracle 11g R2 にアクセスできます。

  AIX 版：Oracle 8i，Oracle 9i，Oracle 10g，Oracle10g R2，Oracle 11g，Oracle 11g R2 にアクセスできます。

  HP-UX（PA-RISC）版：Oracle 8i，Oracle 9i，Oracle 10g，Oracle10g R2，Oracle 11g，Oracle 11g R2 にアクセスできます。

HP-UX（IPF）版：Oracle 9i，Oracle 10g，Oracle10g R2，Oracle 11g，Oracle 11g R2 にアクセスできます。

- Sybase SQL Anywhere，Adaptive Server Anywhere

  Windows 版の DABroker から Sybase SQL Anywhere, 及び Adaptive Server Anywhere にアクセスできます。

- Microsoft SQL Server

  Windows 版の DABroker から Microsoft SQL Server にアクセスできます。

- メインフレーム系データベース

  HP-UX（PA-RISC），又は Windows 版の DABroker から RDA Link for Gateway を経由して，次のメインフレーム環境のデータベースにアクセスできます。

  - VOS3 XDM/RD E2

  また，DABroker から Database Connection Server を経由して，メインフレーム環境のデータベースにアクセスすることもできます。接続できるデータベースを次に示します。

  - VOS3 XDM/RD E2
  - VOS3 XDM/SD E2
  - VOS K SQL/K

1.2　特長

DABroker for C++の特長を次に示します。

(1)　コーディング量を少なくできるクラスライブラリ

データベースアクセス用の C++クラスを提供します。この C++クラスは，データベースへのアクセスをオブジェクト指向のインタフェースでカプセル化しています。このため，C++クラスを使えば，データベースへアクセスするためのアプリケーションを，「埋め込み SQL」や「CLI (Call Level Interface)」のような手続き型のインタフェースでコーディングするより，少ない手続きで作成することができます。このクラスを使うと，次のようなことができます。

- 検索データの参照と更新
- 任意の SQL 文の実行
- トランザクション制御
- テーブル情報の取得

どのようなクラスが提供されているかは，「1.3 提供 C++クラス」を参照してください。

(2)　マルチスレッド処理

アプリケーションは，データベースアクセスを同期で処理するか非同期で処理するかを選択できます。非同期処理では，時間の掛かる SQL 処理を要求してもすぐにアプリケーションに制御を戻すことができ，DBMS でデータの処理をしている間に，アプリケーションで別の処理を進めたりすることができます。アプリケーションからの非同期指定のデータベースアクセス要求は，マルチスレッドで処理されます。

(3)　複数データベースアクセス

複数のデータベースにアクセスするアプリケーションを作成できます。複数の同一種別の DBMS，複数種別の DBMS のどちらにもアクセスできます。

(4)　TPBroker の OTS 機能を使ったトランザクション制御

一つのトランザクションで複数のデータベースを更新するような CORBA アプリケーションを開発する場合に，データベース間の整合性を保つための OTS 機能を利用できます。

OTS 機能を使う場合は，別途 TPBroker が必要です。

(5)　データベースに依存しないエラー処理の実現

DBMS の種別を意識しないでデータベース接続時のユーザ ID 不正，ロックエラーなどが判断できます。

(6)　抽象データ型へのアクセス

HiRDB の抽象データ型は，オブジェクトを格納できる型であり，このオブジェクトデータを扱う専用のプログラムもユーザが任意に定義できます。

1.3　提供 C++クラス

DABroker が提供するクラスライブラリはデータベースアクセスクラスとエラー処理クラスの 2 種類に分かれます。

1.3.1　データベースアクセスクラス

データベースアクセスクラスには簡易版と詳細版の二つのタイプがあります。

簡易版クラスは,2 種類のオブジェクトでデータベースをアクセスするためのクラスです。

詳細版クラスは,データベースを木目細かくアクセスするためのクラスであり，簡易版と比較して，ストアドプロシジャを利用するためのクラスや，テーブル名，フィールド名などの定義情報を参照するクラスが追加されています。

また，DBMS が HiRDB で，繰り返し列へアクセスする場合にだけ利用できる四つのクラスもあります。これらのクラスは，簡易版と詳細版の両方で利用できるクラスです。

各クラスについて次に説明します。

(1)　簡易版クラス

簡易版クラスでは下記に示すデータベースアクセスができます。

- データの検索と更新・削除（?パラメタ使用可）
- 任意の SQL の実行
- トランザクション制御

(2)　詳細版クラス

詳細版クラスでは下記に示すデータベースアクセスができます。

- データの検索と更新・削除（?パラメタ使用可）
- 任意の SQL の実行（?パラメタ使用可）
- トランザクション制御
- ストアドプロシジャの利用
- テーブル,フィールド,プロシジャ,プライマリキー情報などの参照

(3)　簡易版と詳細版の共通のクラス

繰り返し列を扱うためのクラスで，下記に示すデータベースアクセスができます。

- 要素への値の設定
- 要素の値の取得

1.3.2　エラー処理クラス

データベースアクセス時にエラーが発生した場合, このエラー処理クラスを利用してエラーの情報を取得できます。このクラスは簡易版クラスと詳細版クラスの両方で使用できます。

1.4　データベースアクセスの基礎知識

この節では，DABroker が提供する C++クラスライブラリを利用する上で必要となる基礎知識について説明します。

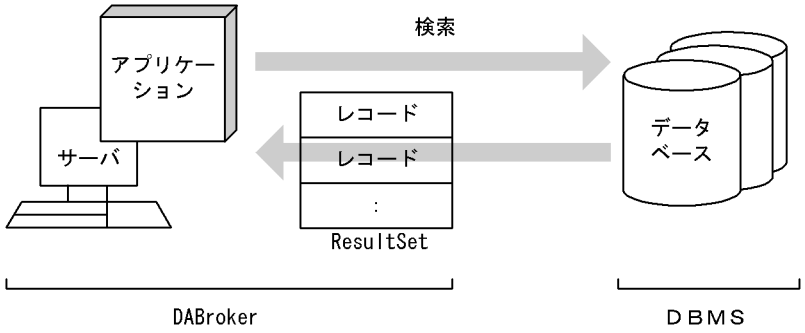
- ResultSet
- データベースへの接続と切断
- トランザクションと排他制御
- 同期・非同期処理
- 繰り返し列

1.4.1　ResultSet

ResultSet とは，レコードのアクセスに使うもので，検索したレコードが ResultSet へ格納されます。ResultSet は，論理的な仮想の表として見ることができ，アプリケーションからは，この ResultSet 中の任意のレコードにカーソルを位置付け，データ取得用のメソッドを呼び出すことで，フィールドの値を参照します。

図 1-2 に ResultSet のイメージ図を示します。

図 1-2　ResultSet のイメージ図



1.4.2　データベースへの接続と切断

データベースをアクセスするには，そのデータベースの種類や存在する場所などの情報が必要であり，この情報は，DABroker 本体の接続先データベース定義ファイルで定義します。

DABroker for C++を利用するアプリケーションでは，ここで定義されたデータベース種別名，データベース名を使用して接続するため，データベース種別の変更や稼働サーバ変更などの影響を受け難い形態となっています。

DABroker for C++では，データベースの接続先の情報を持つオブジェクトを生成し，Connect メソッドでデータベースと接続します。この Connect メソッドでは，ユーザ ID/パスワードなどを指定します。

データベースとの接続は，オブジェクトの削除や Close メソッドで切断します。

1.4.3　同期・非同期処理

DABroker によるデータベースアクセスでは，データベースアクセスを非同期に実行することも簡単にできるインタフェースになっています。

同期・非同期処理の選択は，DBMS との接続の際に呼び出す Connect メソッドの引数で指定します。

(1)　同期処理

同期処理が指定された場合，DABroker は SQL 文の処理要求を受け付けると，アプリケーションに制御を戻さないまま，DBMS に対して SQL 文の処理を要求します。アプリケーションに制御が戻るのは，SQL 文の処理が完了した時点です。

(2)　非同期処理

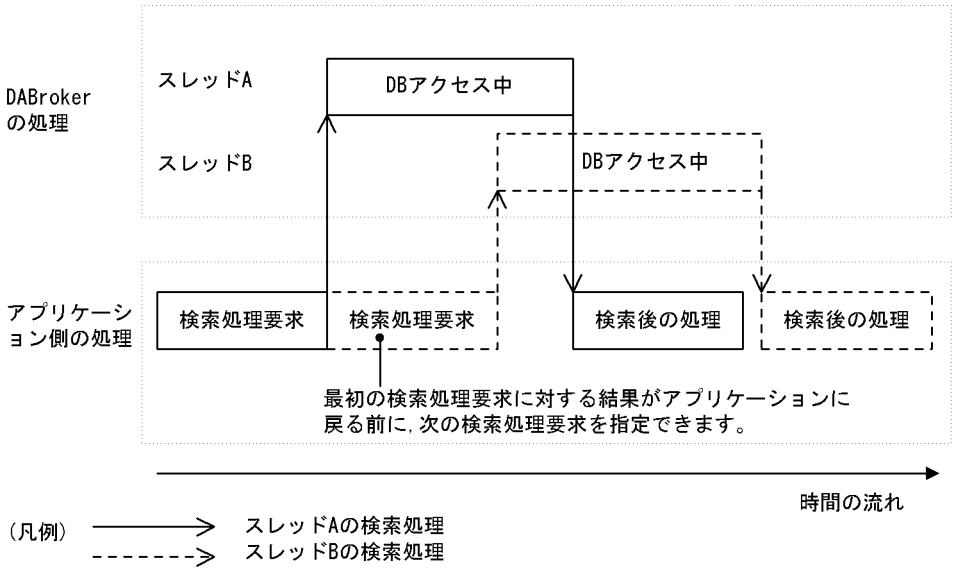
非同期処理が指定された場合，DABroker はアプリケーションからの SQL 文の処理要求を受け取ると，制御をいったんアプリケーションに戻し，同時にスレッドを生成して DBMS に対して SQL 文の処理を要求します。

したがって，アプリケーションでは，複数の SQL 文の処理を同時に行うことが可能であり，性能対策に効果があります。この複数のスレッドを生成して行う処理を，DABroker ではマルチスレッド処理と呼びます。

アプリケーションでは，非同期処理の完了待ちや実行中であるかなどの確認を，クラスの提供するメソッドを利用して行えます。

非同期アクセスの例を図 1-3 に示します。

図 1–3　非同期アクセスの例



1.4.4　トランザクションと排他制御

(1)　トランザクション制御

DABroker for C++を使ったトランザクション制御には二つの方法があります。一つ目は，DABroker for C++のクラスライブラリを使ってトランザクションを制御する方法であり，一つの DBMS を対象にトランザクション制御ができます。二つ目は，CORBA アプリケーションの場合に TPBroker の OTS 機能を使ってトランザクションを制御する方法であり，複数の DBMS を対象にトランザクション制御ができます。

データベースを更新するアプリケーションでは，DBMS ごとに必ずどちらかのトランザクション制御を設定してください。

(a)　一つの DBMS を対象にトランザクション制御を行う方法

簡易版クラスと詳細版クラスでは，トランザクションの制御方法が異なります。簡易版クラスは，Connect メソッドを実行するとそのオブジェクトはトランザクション開始状態になります。詳細版クラスでは，トランザクション制御の有無を選択可能できます。トランザクション制御するには，DBTransaction クラスの，BeginTrans メソッド，Commit メソッド，Rollback メソッドを使用して制御します。

(b)　複数の DBMS を対象にトランザクションの同期制御を行う方法

複数のデータベースに対して更新する場合，障害が発生したときでも複数のデータベース間で矛盾がないよう 2 相コミットと呼ばれる方法で，複数の DBMS に対して同期を取り，コミット又はロールバックすることが可能です。コミット処理中に障害が発生した場合でも，トランザクション内のすべての変更処理を矛盾なく自動的にロールバックできます。

DABroker と TPBroker の OTS 機能を組み合わせてこのトランザクション制御を備えたシステムが構築できます。

(2)　排他制御

データベースのテーブルは，複数のアプリケーションからアクセスされます。このような環境で動作するアプリケーションでは，レコードの排他制御を意識したデータベースアクセスを行う必要があります。レコードの排他を意識したアプリケーションを設計しないと，データベースシステム全体の性能が低下するなどの影響が出ます。

排他制御は，データベースの検索・更新方法にもよりますが，基本的に Resultset を使用したデータベースアクセスで意識する必要があります。

DABroker for C++では，下記の排他方法を提供しています。

- 更新目的の検索

  排他の種類として，TYPE_EXCLUSIVE を指定します。基本的に，SELECT 文には"FOR UPDATE"が付加され，検索したレコードはロックされます。

- 参照目的の検索

  TYPE_NONE を指定します。この指定は，SQL 実行時に排他オプションは付加されず，ロック方法は，対象となるデータベースのデフォルトとなります。

- ダーティリード

  TYPE_NOWAIT を指定します。この指定は，レコードにどの種類のロックが掛かっていても，レコードを検索する（ダーティリード）オプションです。このため，読み取りの一貫性は保証されません。

データベースごとに排他の機能や利用方法が異なるので，上記の方法がすべてのデータベースで利用できるわけではありません。排他制御については，正しく理解して利用することが必要です。

(3)　排他エラーとデッドロック

データベースをアクセスするアプリケーションでは，必ず，排他エラーとデッドロックを意識する必要があります。

この排他エラーとデッドロックのエラーコードは，DBMS によって異なりますが，DABroker では，このエラーコードの違いを吸収し，アプリケーションでは，DBMS を意識することなくエラー処理を記述できます。

(a)　排他エラーとは

排他エラーとは，アプリケーションがデータベースをアクセスしようとした時に，アクセス対象のレコードが，他のアプリケーションにロックされている状態を意味しています。この状態が発生した時に，待ちとするか，エラー報告するかを選択できる機能を提供しています。

(b)　排他エラー処理形態の設定

排他待ちが発生した時の処理方法を，データベース接続時の Connect メソッドで指定できます。

- LOCK_OPT_WAIT
  ロックが解除されるまで待つ
- LOCK_OPT_NOWAIT
  ロックされていた場合は，すぐに DB_ERROR_DRIVER_ERROR のエラーをスロー

これ以外に，ロックされていたときのエラースロー時に，ロールバックするかどうかを指定する，LOCK_OPT_WITH_ROLLBACK, LOCK_OPT_WITHOUT_ROLLBACK がありますが，使用する DBMS によって動作が異なるため，詳細については「4.簡易版関数詳細」又は「5.詳細版関数詳細」の Connect メソッドを参照してください。

(c)　デッドロックとは

二つ以上のトランザクションが二つ以上の資源の確保をめぐって互いに相手を待つ状態となり，そこから先へ処理が進まなくなることをデッドロックといいます。

デッドロックは，アプリケーションからアクセスするテーブルやレコードの処理する順番が統一されていない場合やロック範囲が広い場合など，さまざまなケースで発生します。

デッドロック状態となることを防ぐ目的で，DBMS は，一つのアプリケーションにデッドロックを通知します。このデッドロック状態を通知されたアプリケーションは，すみやかにロールバックし，ロックしているレコードを解放する必要があります。

データベースアクセス中にデッドロックが発生するとエラーがスローされ，DBSQLCA クラスの e_USERCODE に 20002，e_USERERROR にエラーメッセージが設定されます。

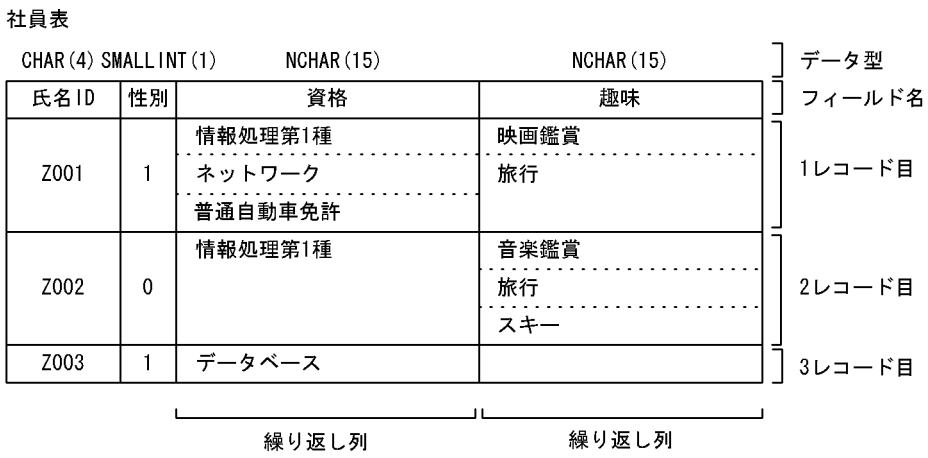
1.4.5　繰り返し列

ここでは，DABroker for C++から繰り返し列へアクセスするための基礎知識として，SQL 文を使った，繰り返し列のデータ操作について説明します。

繰り返し列は，図 1-4 に示すように，複数の**要素**（データ）を一つのフィールド値として持てるフィールドのことです。図の例では，資格及び趣味のフィールドを繰り返し列としています。1 レコード目の資格

フィールド（繰り返し列）には三つの要素があるといい，要素を特定するために，配置されている順に要素番号が 1 から付けられます。また，情報処理第 1 種，映画鑑賞，旅行などの各データを要素の値といいます。例えば，1 レコード目の資格フィールドで，要素番号 1 の要素の値は「情報処理第 1 種」となります。

図 1-4　繰り返し列を使ったテーブル



次に，上記の社員表テーブルを作成する場合の，テーブル定義及び複数列インデックス定義の SQL 文の例を示します。

```
CREATE TABLE 社員表
       (氏名ID CHAR(4) NOT NULL,           --NULL値不可
        性別   SMALLINT(1) NOT NULL,       --NULL値不可
        資格   NCHAR(15)  ARRAY[3],
                          --フィールド値として持てる最大要素数3
        趣味   NCHAR(15)  ARRAY[4]);
                          --フィールド値として持てる最大要素数4
CREATE INDEX 資格趣味 ON 社員表(資格, 趣味);
```

以下に，繰り返し列のあるテーブルへのデータアクセスについて説明します。

(1)　レコードの検索

繰り返し列の各要素の値を検索の条件として利用できます。SQL 文は以下のように指定します。

```
// 資格の「ネットワーク」，趣味の「旅行」の，両方の条件を満たす氏名IDを検索
SELECT 氏名ID FROM 社員表
       WHERE ARRAY(資格, 趣味) [ANY] ( 資格 =N'ネットワーク'
                                                AND 趣味 =N'旅行' );
```

検索の条件に繰り返し列の要素の値を使う場合は，そのフィールドに対して複数列インデクスが定義されている必要があります。

(2)　レコードの追加

繰り返し列を持つレコードを追加する際には，その要素の指定順序を考慮する必要があります。ARRAY で指定した順に要素は配置されます。

```
// 社員IDがZ004のレコードを追加
INSERT INTO 社員表
       VALUES ('Z004',
               1,
               ARRAY[N'英語検定',N'ネットワーク']
               ARRAY[N'パソコン',N'旅行',N'音楽鑑賞',N'ギター']);
```

(3)　レコードの更新

繰り返し列を更新する場合はSQLのUPDATE文を使いますが，繰り返し列の更新は，次の三つの場合に分けられます。

- 新しく要素を追加する。
- 既存の要素を更新する。
- 既存の要素を削除する。

(a)　要素の追加(ADD句)

フィールド値に要素を追加します。[*]を指定することによって，要素は最後に追加されます。

(例)

```
// Z002レコードの資格フィールドの最後の要素位置に，「データベース」を追加
UPDATE 社員表
        ADD 資格[*]  =ARRAY [N'データベース']
        WHERE 氏名ID =Z002 ;
```

(b)　要素の更新(SET句)

既存の要素を更新します。更新する要素の位置は，繰り返し列内の要素番号を使って指定します。

(例)

```
// z001の趣味を「映画鑑賞」から「インターネット」へ変更
UPDATE 社員表
        SET 趣味[1]  =ARRAY [N'インターネット']
        WHERE 氏名ID =Z001 ;
```

(c)　要素の削除(DELETE句)

要素を削除する場合は，削除する要素の要素番号を指定します。要素を削除した場合は，その要素以下の要素番号は繰り上がります。したがって，要素番号1の要素を削除した場合，要素番号2の要素が1になります。

(例)

```
// Z002の趣味「音楽鑑賞」を削除
UPDATE 社員表
        DELETE 趣味[1]
        WHERE 氏名ID =Z002 ;
```

(4)　レコードの削除

レコードを削除する場合，繰り返し列を持たない場合と同様にSQLのDELETE文を指定します。

(例)

```
// 資格の「ネットワーク」，趣味の「旅行」の，両方の条件を満たすレコードを削除
DELETE FROM 社員表
        WHERE ARRAY(資格，趣味)[ANY]( 資格 =N'ネットワーク'
                                                AND 趣味 =N'旅行' );
```

1.5　簡易版クラスのデータベースアクセス

この節では，簡易版クラスを利用したアプリケーションが，どのようなデータベースアクセスができるかについて説明します。

1.5.1　提供クラスとオブジェクト

簡易版クラスでは，下記クラスを提供しています。

- DBRDatabase クラス
- DBRResultSet クラス

このクラスでは，２種類のクラスから Database オブジェクトと ResultSet オブジェクトを生成します。Database オブジェクトは，データベースとの接続・切断を制御するオブジェクトであり，ResultSet オブジェクトはデータベースの検索・更新などを行うオブジェクトです。 データベースアクセスには，この二つのオブジェクトのメソッドを利用します。

また，繰り返し列を扱うための次のクラスも利用できます。

- DBRArrayDataFactory クラス
- DBRArrayData クラス
- DBRArrayDataPtr クラス
- DBRArrayDataConstPtr クラス

DBRArrayDataFactory オブジェクトは，Database オブジェクトから生成されます。これらは順に，繰り返し列を扱うための DBRArrayData クラスを管理するオブジェクトのクラス, 繰り返し列の要素の操作を管理するオブジェクトのクラス, 更新を目的として DBRArrayData クラスのインスタンスを管理するためのスマートポインタクラス，参照の場合に DBRArrayData クラスのインスタンスを管理するためのスマートポインタクラスです。

データベースアクセスで発生したエラー情報は, これらのオブジェクトに格納されません。エラー処理クラスライブラリのクラスを使用してエラー情報を取得できます。

(1)　Database オブジェクト

Database オブジェクトは，DBRDatabase クラスのコンストラクタ（データベース種別名を指定）を利用してオブジェクトを生成します。生成したオブジェクトは,オブジェクトを宣言した関数が終了すると自動的に削除されます。

一つの Database オブジェクトで接続するデータベースはひとつであり，複数のデータベースと接続する場合には，複数の Database オブジェクトが必要となります。

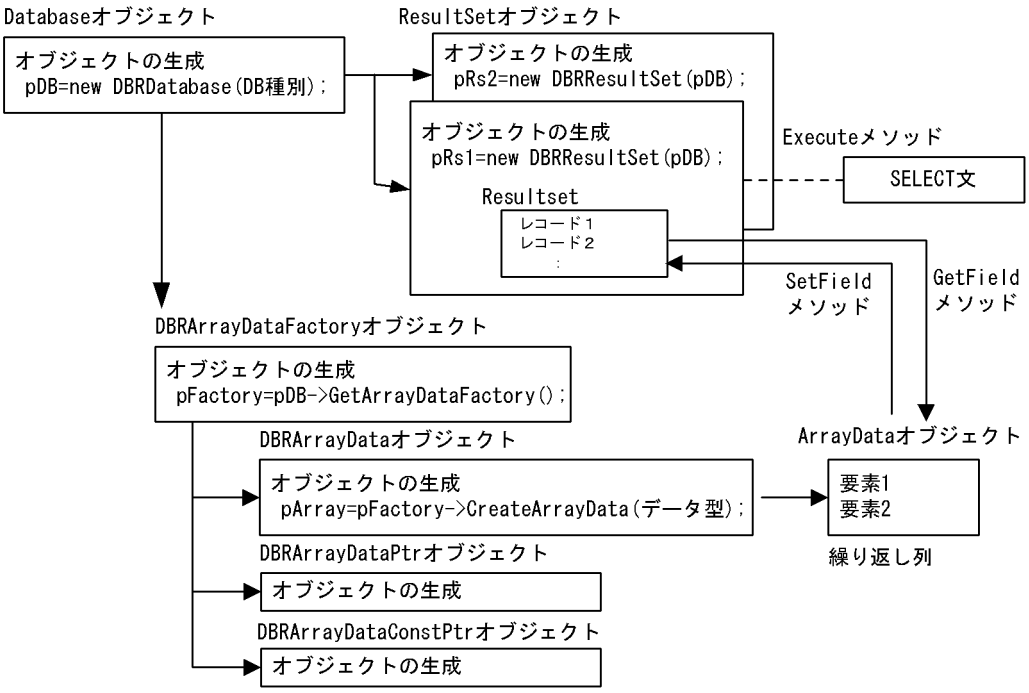
(2)　ResultSet オブジェクト

ResultSet オブジェクトは，データベースの検索・更新などを行うオブジェクトであり，Database オブジェクトを生成した後, DBRResultSet クラスのコンストラクタを利用して生成します。生成したオブジェクトは,オブジェクトを宣言した関数が終了するとき自動的に削除されますが, ResultSet オブジェクトの Close メソッドでも削除できます。この ResultSet オブジェクトは，ひとつの Database オブジェクトに対して最大６４個まで生成できます。

ひとつの ResultSet オブジェクトは，ひとつの SELECT 文に対応し，検索したレコードを格納します。

簡易版クラスで生成したオブジェクトのイメージを図 1-5 に示します。

図 1-5　簡易版クラスで生成したオブジェクトのイメージ



(3)　繰り返し列を扱うオブジェクト

繰り返し列を扱うオブジェクトの種類は次の４種類があります。

(a)　DBRArrayDataFactory オブジェクト

このオブジェクトは繰り返し列を扱う最上位オブジェクトです。繰り返し列の要素を更新する場合に接続するデータベースごとに必要です。このオブジェクトは DBRDatabase オブジェクトの GetArrayDataFactory メソッドで生成します。

(b)　DBRArrayData オブジェクト

このオブジェクトは繰り返し列のデータを持つオブジェクトであり，ResultSet オブジェクトの GetField メソッドでは参照用の DBRArrayData オブジェクトが生成され， DBRArrayDataFactory オブジェクトの CreateArrayData メソッドでは更新に使用するオブジェクトが生成されます。このオブジェクトは，一つのレコード内の一つの繰り返し列の要素を格納します。

要素に値を設定する場合は SetData メソッド，要素の値を参照する場合は GetData メソッドを使用します。

(c)　DBRArrayDataConstPtr オブジェクト

このオブジェクトは DBRArrayData オブジェクトをポイントするものです。DBRArrayData オブジェクトの要素を参照専用で使用する場合は，このオブジェクトのポインタを使用します。

(d)　DBRArrayDataPtr オブジェクト

このオブジェクトは DBRArrayData オブジェクトをポイントするものです。DBRArrayData オブジェクトの要素を更新する場合は，このオブジェクトのポインタを使用します。

1.5.2　データベースとの接続と切断

(1)　データベースとの接続

データベースと接続するには，DBRDatabase クラスのコンストラクタを利用して Database オブジェクトを生成し，Connect メソッドを呼び出してデータベースと接続します。

接続するデータベース種別名は，コンストラクタの引数で設定し，データベース名やユーザ ID，パスワードは Connect メソッドの引数で指定します。

また，下記オプションを設定することも可能です。

- 検索したときに該当するレコードがロックされていた場合の振る舞い
- SQL 文を同期処理，又は非同期処理のどちらで実行するか

(2)　データベースとの切断

データベースとの切断は，Database オブジェクトの Close メソッドを呼び出すことで行い，再接続には Connect メソッドを使用します。

Database オブジェクトの削除（オブジェクトを宣言した関数の終了）でもデータベースが切断されます。

1.5.3　レコードの検索

レコードの検索には, DBRResultSet クラスのコンストラクタを利用して生成する ResultSet オブジェクトのメソッドを使用します。

レコードの検索は，SELECT 文を Execute メソッドで通知し，Open メソッドで検索を実行します。検索したレコードは，ResultSet オブジェクトに格納されます。

レコードの検索には検索したレコードを参照だけする参照目的とレコードの更新又は削除をする更新目的の 2 種類があり，この目的により検索するレコード数が異なります。

また，SELECT の一部を実行時に置き換える，?パラメタの利用も可能です。?パラメタは ORACLE ではプレイスホルダと呼ばれています。

1.5.4　検索レコードの参照

検索したレコードは，ResultSet に格納されています。カーソルは最初レコードの先頭に位置付けられていて，必要に応じてカーソルを位置付け，GetField メソッドを使用してフィールドの内容を参照します。フィールドの内容は，GetField の引数で指定したデータ型に変換できます。

カーソル位置付けには，次へ移動（Next メソッド），前へ移動（Pervious メソッド），最後へ移動（Bottom メソッド），先頭へ移動（Top メソッド）などのメソッドがあります。また，現在のレコード位置を知るためのメソッド（GetCurrent，GetCurrentOfResultSet メソッド）もあります。

なお，更新目的で検索した場合には，1 レコードだけ ResultSet に格納されるため，Next メソッドだけが利用できます。

1.5.5　レコードの更新

レコードの更新方法には，検索したレコードを更新する方法（ResultSet を利用）と，UPDATE 文で直接更新する方法があります。

(1)　検索したレコードの更新

検索したレコードを更新するためには，次の順序で処理する必要があります。

1. 更新目的でレコードを検索
2. Edit メソッドの呼び出し
3. SetField メソッドでフィールドの値を更新
4. Update メソッドで更新を指示
5. Next メソッドで次のレコードに位置付け

(2)　UPDATE 文を利用した更新

UPDATE 文で更新するためには,ExecuteDirect メソッドを利用します。更新するレコードの条件や更新フィールドの値を指定した UPDATE 文を，ExecuteDirect メソッドの引数に設定することにより，更新できます。ExecuteDirect メソッドでは，?パラメタは利用できません。

1.5.6　レコードの削除

レコードの削除方法には，検索したレコードを削除する方法（ResultSet を利用）と，DELETE 文で直接削除する方法があります。

(1)　検索したレコードの削除

検索したレコードを削除するためには，次の順序で処理する必要があります。

1. 更新目的でレコードを検索
2. Edit メソッドの呼び出し
3. Delete メソッドで削除を指示
4. Next メソッドで次のレコードに位置付け

(2)　DELETE 文を利用した削除

DELETE 文で削除するためには,ExecuteDirect メソッドを利用します。削除するレコードの条件を指定した DELETE 文を，ExecuteDirect メソッドの引数に設定することにより，削除できます。
ExecuteDirect メソッドでは，?パラメタは利用できません。

1.5.7　レコードの追加

レコードを追加するためには,ExecuteDirect メソッドを利用します。追加するレコードの内容を指定した INSERT 文を ExecuteDirect メソッドの引数に設定することにより，追加できます。

1.5.8　繰り返し列へのアクセス

繰り返し列の要素を含むレコードを扱う場合は, 基本的には通常のレコードのデータを扱う場合と同じですが，繰り返し列のデータを扱う方法が異なります。

(1)　検索条件としての要素の利用

要素の値を検索，更新，削除，追加時の条件として利用できます。SQL 文の WHERE 句に Array 句を使って条件を設定します。SQL 文の指定方法の詳細については「1.4.5 繰り返し列」を参照してください。

(2)　ResultSet に取得した要素の参照

ResultSet に読込んだ繰り返し列の要素を参照する方法について説明します。

ResultSet に読込んだ繰り返し列は直接アクセスできないため，いったんアプリケーションがアクセスできる領域(DBRArrayData オブジェクト)に取り込む必要があります。このオブジェクトは DBRResultSet クラスの GetField メソッドで生成されます。

DBRArrayData オブジェクト中の要素は GetData メソッドで参照します。また，要素の値は，GetData メソッドの引数で指定したデータ型に変換できます。

(3)　ResultSet を利用した要素の更新

ResultSet に読込んだ繰り返し列を更新する方法には，次の 2 種類があります。

(a)　繰り返し列の要素の更新

繰り返し列は，ResultSet に読み込んだ繰り返し列から GetField メソッドで DBRArrayData オブジェクトに取込みますが, このオブジェクトは参照専用であるためこのオブジェクト内の要素を直接更新できません。このため，更新用の DBRArrayData オブジェクトを，参照用の DBRArrayData オブジェクトから CreateArrayData メソッドでコピーして生成します。

DBRArrayData オブジェクトの要素は，SetData メソッドで更新します。要素の更新後，SetField メソッドを呼び出して，繰り返し列を ResultSet に反映します。

(b)　繰り返し列の一括更新

ResultSet から繰り返し列のデータを取り込まずに，繰り返し列にデータを設定する方法です。

DBRArrayData オブジェクトを生成し，DBRArrayDataFactory オブジェクトの CreatArrayData メソッドで繰り返し列の属性を指定し，DBRArrayData オブジェクトを生成します。

次に，DBRArrayData オブジェクトの Create メソッドを呼び出し，要素数分の領域を確保します。要素の値は，SetData メソッドを呼び出して，確保した領域に設定します。要素を設定後，SetField メソッドを呼び出して，繰り返し列を ResultSet に反映します。

(4)　SQL 文を利用した要素の追加，更新，削除

DBRArrayData オブジェクトを利用しないで繰り返し列を更新する方法について説明します。この方法は，SQL の UPDATE 文に，ADD，SET，DELETE 句を使います。 ADD，SET，DELETE 句の指定方法については，「1.4.5 繰り返し列」を参照してください。

SQL の実行には，Database オブジェクトの ExecuteDirect メソッドを使います。

(5)　繰り返し列を含むレコードの追加

繰り返し列を含むレコードを追加するには, 繰り返し列を含まない場合と同様に SQL の INSERT 文を利用します。INSERT 文では，VALUES 句中の ARRAY 句に設定する要素の値を指定します。SQL の INSERT 文の指定方法については，「1.4.5 繰り返し列」を参照してください。

SQL の実行には，Database オブジェクトの ExecuteDirect メソッドを使います。

1.5.9　テーブルの定義と削除

TPBroker を利用したトランザクション環境では，定義系の SQL(CREATE 文など)は利用できないため注意してください。

(1)　テーブルの定義

テーブル（表）を定義するためには，ExecuteDirect メソッドを利用します。定義するテーブルの情報を指定した CREATE TABLE 文を，ExecuteDirect メソッドの引数に設定することで，テーブルの定義ができます。

(2)　テーブルの削除

テーブル（表）を削除するためには，ExecuteDirect メソッドを利用します。削除するテーブルを指定した，DROP TABLE 文を，ExecuteDirect メソッドの引数に設定することで，テーブルの削除ができます。

1.5.10　DBRResultSet 仮想関数の利用

DBRResultSet クラスでは，オーバライド可能な仮想関数を提供しています。

(1)　DBRResultSet クラスの仮想関数

DBRResultSet クラスでは，三つの仮想関数を利用できます。これらの仮想関数を定義した場合,DBRResultSet クラスの仮想関数にオーバライドされて実行されます。

- OnBeforeRefresh… Open メソッドで ResultSet に検索結果を読込む前に実行される関数
- OnMoveRecord … カーソルの移動時に呼び出される関数
- OnEndRecord … Next メソッドで IsEOF 条件の時に呼び出される関数

1.6　詳細版クラスのデータベースアクセス

この節では，詳細版クラスを使用したアプリケーションが，どのようなデータベースアクセスができるかについて説明します。

1.6.1　提供クラスとオブジェクト

詳細版クラスでは，下記クラスを提供しています。

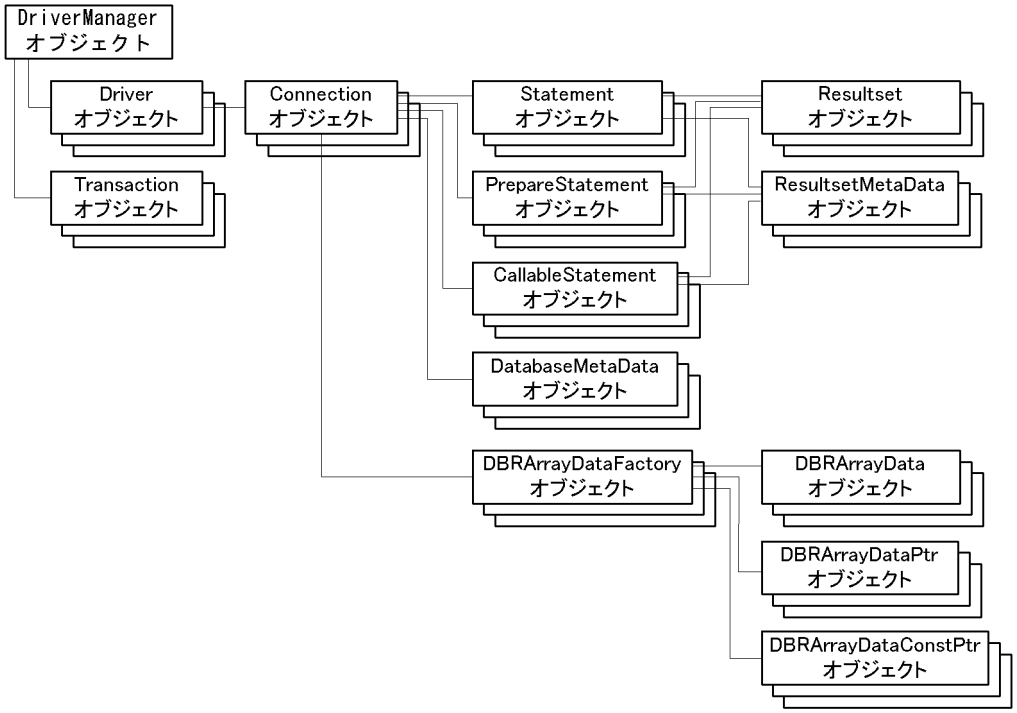
- DBDriverManager クラス
  DriverManager オブジェクト（トップオジェクト）のクラスです。
- DBDriver クラス
  DBMS の種別を管理する Driver オブジェクトのクラスです。 DBDriverManager の Driver メソッドから Driver オブジェクトを生成します。
- DBTransaction クラス
  トランザクションを管理するオブジェクトのクラスです。 DBDriverManager クラスの Transaction メソッドを使用して，Connection オブジェクト対応に生成します。
- DBConnection クラス
  DBMS とのコネクションを管理するオブジェクトのクラスです。DBDriver クラスの Connect メソッドから Connection オブジェクトを生成します。
- DBStatement クラス
  SQL 文の実行を管理するオブジェクトのクラスです。DBConnection クラスのメソッドを使用して生成します。一つのオブジェクトから複数の SQL 文を実行できます。
- DBPreparedStatement クラス
  ?パラメタ付き SQL 文の実行を管理するオブジェクトのクラスです。DBConnection クラスのメソッドに実行する SQL 文を渡して生成します。
- DBResultSet クラス
  検索結果が格納される ResultSet を管理するオブジェクトのクラスです。DBStatement クラス，DBPreparedStatement クラス又は DBCallableStatement クラスのメソッドを使用して生成します。
- DBResultSetMetaData クラス
  検索結果のフィールド名称などを管理するオブジェクトのクラスです。 DBStatement クラス，DBPreparedStatement クラス又は DBCallableStatement クラスのメソッドを使用して生成します。
- DBCallableStatement クラス
  ストアドプロシジャの実行を管理するオブジェクトのクラスです。DBConnection クラスのメソッドを使用して生成します。
- DBDatabaseMetaData クラス
  接続するデータベースのテーブル一覧などの情報を管理するオブジェクトのクラスです。アプリケーション実行時に，接続したデータベースのテーブル一覧，フィールド一覧などの最新情報を参照する場合に必要なオブジェクトです。 DBConnection クラスを使用して，接続するデータベース対応に生成します。
- DBRArrayDataFactory クラス
  繰り返し列を扱うための DBRArrayData クラスを管理するオブジェクトのクラスです。
  DBConnection クラスのメソッドを使用して生成します。
- DBRArrayData クラス

繰り返し列を持つレコードのデータ操作を管理するオブジェクトのクラスです。繰り返し列の要素に値を設定したり，要素の値を取得する場合に必要なオブジェクトです。DBArrayDataFactory クラスのメソッドを使用して生成します。

- DBRArrayDataPtr クラス
  DBRArrayData クラスのインスタンスを管理するためのスマートポインタクラスです。繰り返し列を更新する場合に利用するクラスです。
- DBRArrayDataConstPtr クラス
  DBRArrayData クラスのインスタンスを管理するためのスマートポインタクラスです。繰り返し列を参照する場合に利用するクラスです。

このクラスのオブジェクト構造を図 1-6 に示します。

図 1-6　詳細版クラスのオブジェクト構造



1.6.2　データベースとの接続と切断

(1)　データベースとの接続

データベースと接続するには，最初に DriverManager オブジェクトを生成する必要があります。次に，Driver オブジェクトを生成します。その後，Connect メソッドを呼び出してデータベースと接続します。データベ―スと接続すると Connection オブジェクトが生成されます。

接続するデータベース種別名は，DriverManager オブジェクトの Driver メソッドで設定し，データベース名やユーザ ID，パスワードは Connect メソッドの引数で指定します。

また，下記オプションを設定することも可能です。

- 検索したときに該当するレコードがロックされていた場合の振る舞い
- SQL 文を同期処理，又は非同期処理のどちらで実行するか

(2)　データベースとの切断

Connection オブジェクトを削除することで，データベースが切断されます。再接続には，再度 Connect メソッドを呼び出します。

1.6.3　レコードの検索

データベースの検索には, DBStatement クラスと DBPreparedStatement クラスを使用する方法があります。DBStatement クラスでは，アプリケーションで SELECT 文をすべて組み立てますが，DBPreparedStatement クラスでは, ?パラメタを使用したアクセスができます。この?パラメタとは, SQL 文の一部を可変の変数として，アプリケーション実行時に，変数部分のデータを設定し SQL 文を組み立てる方法です。ORACLE ではプレイスホルダと呼ばれています。

データベースを検索した結果は，ResultSet で受け取り，この ResultSet を参照することで，検索したレコードの内容を参照することができます。

(1)　DBStatement クラスを利用した検索

一般的な使用方法としては，DBConnection クラスの CreateStatement メソッドを呼び出して，Statement オブジェクトを生成します。次に，SELECT 文を Execute メソッドで渡して，データベースを検索し，検索結果である ResultSet を取得します。GetResultSet メソッドで検索結果を参照します。

(2)　DBPreparedStatement クラスを利用した検索

DBPreparedStatement クラスでのデータベースアクセスは，?パラメタを使用した SELECT 文を CreatePreparedStatement メソッドで渡し，SetParam メソッドで?パラメタへのデータを設定します。その後 Execute メソッドで検索を実行し, 検索結果である ResultSet を取得します。GetResultSet メソッドで検索結果を参照します。

1.6.4　検索レコードの参照

検索したレコードは,ResultSet に格納されているので, この ResultSet 中のレコードにカーソルを位置付け GetField メソッドを使用してフィールドの内容を参照します。フィールドの内容は，GetField の引数で指定したデータ型に変換できます。

カーソル位置付けには，次へ移動（Next メソッド），前へ移動（Pervious メソッド），最後へ移動（Bottom メソッド），先頭へ移動（Top メソッド）などのメソッドがあります。また，現在のレコード位置を知るためのメソッド（GetCurrent，GetCurrentOfResultSet メソッド）もあります。

1.6.5　レコードの更新

データの更新方法には，前節で説明した ResultSet を利用する方法と，SQL の UPDATE 文を利用する方法とが選択できます。

(1)　検索したレコードの更新

検索したレコードを更新するためには，次の順序で処理する必要があります。

1. 更新目的でレコードを検索
2. Edit メソッドの呼び出し
3. SetField メソッドでフィールドの値を更新

4. Update メソッドで更新を指示
5. Next メソッドで次のレコードに位置付け

(2)　UPDATE 文を利用した更新

SQL 文の UPDATE 文を使った更新にも二つの方法があり，DBStatement クラスと DBPreparedStatement クラスを利用できます。

- DBStatement クラスを利用した更新

  DBConnection クラスの CreateStatement で Statement オブジェクトを生成します。実行する UPDATE 文を Execute メソッドに渡すことでデータを更新できます。

  又は，DBConnection クラスも使えます。UPDATE 文を ExecuteDirect メソッドで指定して，DBStatement クラスよりも簡単にデータを更新を追加できます。

- DBPreparedStatement クラスを利用した更新

  DBPreparedStatement クラスを使用する場合，?パラメタを利用して検索条件や更新データを実行時に変更することができます。

  ?パラメタを指定した UPDATE 文を CreatePreparedStatement メソッドで指定し，Execute メソッドを呼び出します。SetParam メソッドで?パラメタへのデータを設定し，ExecuteUpdate メソッドで更新を実行します。

1.6.6　レコードの削除

レコードの削除方法には，前節で説明した ResultSet を利用する方法と，SQL の DELETE 文を利用する方法とが選択できます。

(1)　ResultSet を利用したレコード削除

検索したレコードを削除するためには，次の順序で処理する必要があります。

1. 更新目的でレコードを検索
2. Edit メソッドの呼び出し
3. Delete メソッドで削除を指示
4. Next メソッドで次のレコードに位置付け

(2)　DELETE 文を利用した削除

SQL 文の DELETE 文を使った削除にも二つの方法があり，DBStatement クラスと DBPreparedStatement クラスを利用できます。

- DBStatement クラスを利用した削除

  DBConnection クラスの CreateStatement で Statement オブジェクトを生成します。実行する DELETE 文を Execute メソッドに渡すことでレコードを削除できます。

  又は，DBConnection クラスも使えます。DELETE 文を ExecuteDirect メソッドで指定して，DBStatement クラスよりも簡単に行を削除できます。

- DBPreparedStatement クラスを利用した削除

  DBPreparedStatement クラスを使用する場合，?パラメタを利用した検索条件を実行時に変更することができます。

?パラメタを指定した DELETE 文を CreatePreparedStatement メソッドで指定し，Execute メソッドを呼び出します。SetParam メソッドで?パラメタへのデータを設定し，ExecuteUpdate メソッドで削除します。

1.6.7　レコードの追加

レコードを追加するときは，INSERT 文を使います。DBStatement クラスと DBPreparedStatement クラスを利用できます。

- DBStatement クラスを利用したレコードの追加

  DBConnection クラスの CreateStatement で Statement オブジェクトを生成します。実行する INSERT 文を Execute メソッドに渡すことでレコードを追加できます。

  DBConnection クラスも使えます。この場合，INSERT 文を ExecuteDirect メソッドで指定して，DBStatement クラスよりも簡単にレコードを追加できます。

- DBPreparedStatement クラスを利用した追加

  DBPreparedStatement クラスを使用する場合，?パラメタを利用して追加するレコードのデータを実行時に設定できます。

  ?パラメタを指定した INSERT 文を CreatePreparedStatement メソッドで指定し，Execute メソッドで実行します。SetParam メソッドで?パラメタへのデータを設定し，ExecuteUpdate メソッドで追加します。

  なお，一度に複数のレコードをまとめて追加できます。

1.6.8　繰り返し列へのアクセス

繰り返し列の要素を含むレコードを扱う場合は，基本的には通常のレコードのデータを扱う場合と同じですが，繰り返し列のデータを扱う方法が異なります。

(1)　検索条件としての要素の利用

要素の値を検索，更新，削除，追加時の条件として利用できます。この方法は，簡易版と同じです。

C++クラスライブラリでの各 SQL 文の指定方法は，通常のレコードを扱う場合と同様です。要素を条件にした検索，更新，削除，追加はパラメタを含む場合 DBPreparedStatememt クラス，含まない場合 DBStatement 又は DBConnection クラスを使います。

(2)　ResultSet に取得した要素の参照

ResultSet に読込んだ繰り返し列の要素を参照する方法は，簡易版と同じです。

(3)　ResultSet を利用した要素の更新

ResultSet を利用した繰り返し列の要素を更新する方法は，簡易版と同じです。

(4)　?パラメタを利用した要素の一括更新

SQL 文では，通常のレコードを更新する場合と同様に繰り返し列は指定しないで設定し，後で DBRArrayData クラスを利用して要素を設定し，更新する方法です。この方法では，繰り返し列全体を?パラメタに設定するため，要素ごとの更新ではなく，要素全体の更新になります。

?パラメタを指定した SQL の UPDATE 文の実行は，DBPreparedStatement クラスを使います。

次に，DBRArrayData オブジェクトの Create メソッドを呼び出し，要素数分の領域を確保します。要素の値は，SetData メソッドを呼び出して，確保した領域に設定します。要素を設定後，SetParam メソッドを呼び出して，繰り返し列を反映します。

(5)　SQL 文を利用した要素の追加，更新，削除

DBRArrayData オブジェクトを利用しないで繰り返し列を更新する方法について説明します。この方法は，簡易版と同じですが，関係するクラスが異なります。

SQL の実行には，DBConnection 又は DBStatement クラスを，?パラメタを使う場合は DBPreparedStatement クラスを使います。

(6)　繰り返し列を含むレコードの追加

繰り返し列を含むレコードを追加する方法は簡易版と同じですが，関係するクラスが異なります。

SQL の実行には，DBConnection 又は DBStatement クラスを，パラメタを使う場合は DBPreparedStatement クラスを使います。

1.6.9　テーブルの定義と削除

TPBroker を利用したトランザクション環境では，定義系の SQL(CREATE 文など)は利用できないため注意してください。

(1)　テーブルの定義

DABroker を利用して，テーブル（表）を定義できます。テーブルの定義は，SQL の CREATE TABLE 文を使用します。アプリケーションからは DBStatement クラスを使い，SQL 文を CreateStatement メソッドで指定し，Execute メソッドで実行します。

又は，DBConnection クラスを使用して，SQL 文を実行することもできます。この場合，SQL 文を ExecuteDirect メソッドで指定して，SQL 文を実行します。

(2)　テーブルの削除

DABroker を利用して，テーブル（表）を削除できます。テーブルの削除は，SQL の DROP TABLE 文を使用します。アプリケーションからは DBStatement クラスを使い, SQL 文を CreateStatement メソッドで指定し，Execute メソッドで実行します。

又は，DBConnection クラスを使用して，SQL 文を実行することもできます。この場合，SQL 文を ExecuteDirect メソッドで指定して，SQL 文を実行します。

1.6.10　テーブル定義情報の参照

下記クラスを使用することにより，テーブル一覧，フィールド一覧，プロシジャ一覧などの情報を取得できます。

- classListTables クラス
- classListColumns クラス
- classListProcedures クラス
- classListProcedureColumns クラス
- classListPrimaryKeys クラス

1.6.11　ストアドプロシジャの利用

この項では，ストアドプロシジャの利用方法について説明します。

なお，メインフレーム系の DBMS ではストアドプロシジャを利用したアクセスはできません。

また，繰り返し列はストアドプロシジャでは利用できません。

(1)　ストアドプロシジャの概要

ストアドプロシジャとは，SQL でデータベースアクセス処理を記述し，この処理を CREATE PROCEDURE でデータベースに登録したものです。

ストアドプロシジャでは，実行に必要な値を引数としてプロシジャに渡したり(IN)，プロシジャから実行結果を戻したり(OUT)，値をプロシジャに渡してその実行結果を戻したり(INOUT)できます。

ストアドプロシジャは次のような形式で作成します。これは，引数を使って値をプロシジャに渡すための例で，Sample1 という名称のプロシジャを作成しています。f1=入力値の条件でレコードを table1 から検索し，そのレコードの f2 フィールドの値を受け取ります。

```
CREATE PROCEDURE Sample1 (IN @f1 INT,OUT @f2 char(20))
BEGIN
        SELECT f2 into @f2 FROM table1  where f1=@f1;
END
```

なお，SQL Anywhere, Adaptive Server Anywhere の場合は，実行結果を引数としてだけでなく，ResultSet として返すこともできます。

ストアドプロシジャ内で使用できる SQL 文や文法は DBMS によって異なるので，プロシジャの作成やコンパイル方法,及び DBMS への登録方法は，使用する DBMS のドキュメントを参照してください。

(2)　ストアドプロシジャの実行と結果の受け取り

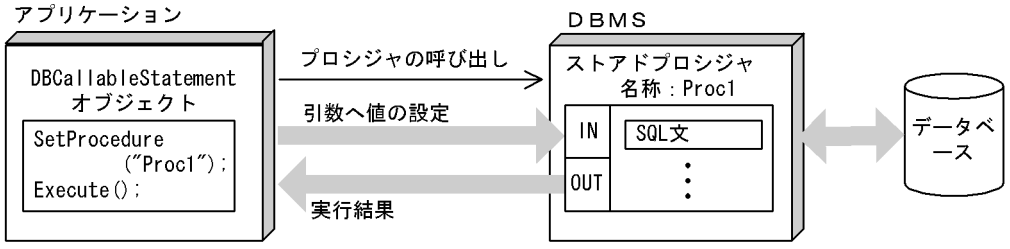
ストアドプロシジャは SQL の CALL 文を使って呼び出せますが，詳細版クラスにはストアドプロシジャを扱うための DBCallableStatement クラスがあります。このクラスには，実行するストアドプロシジャの準備，実行，引数の設定，更に，ストアドプロシジャの実行結果を取得するためのメソッドがあります。

(a)　ストアドプロシジャの実行

アプリケーションでは，実行するストアドプロシジャ名を SetProcedure メソッドを使って DBMS に通知します。引数を持つ場合，プロシジャ名の後に引数の数だけ"?"を指定し，SetParam メソッドで引数の値を設定します。次に，Execute メソッドでプロシジャを実行します。

引数を持つプロシジャの実行を図 1-7 に示します。

図 1-7　引数を持つプロシジャの実行



(b)　戻り値の受取り

戻り値を引数で受け取る場合，DBCallableStatement クラス GetParam メソッドを呼び出して取得します。

(c)　戻り値として複数のレコードを取得する場合

下記 DBMS の場合，ストアドプロシジャでの SELECT 文の実行結果が複数のレコードでもアプリケーションで受け取ることができます。

- SQL Anywhere
- Adaptive Server Anywhere

この場合，ストアドプロシジャでは Result で戻り値の型を定義し，アプリケーションでは，DBResultSet クラスの ResultSet として受取り，GetField メソッドで参照します。

```
CREATE PROCEDURE sample ()
RESULT (”Value” INT,”Shop” CHAR(30))
BEGIN
    SELECT  CAST( sum( sales_order_items.quantity * product.unit_price)
                AS INTEGER ) AS value
        shop_name,
    FROM customer
        INNER JOIN sales_order
        INNER JOIN sales_order_items
        INNER JOIN product
    GROUP BY shop_name
    ORDER BY value desc;
END
```

この例では，value 及び shop_name の検索結果を ResultSet として扱うために，RESULT 句で Value 及び Shop フィールドとそのデータ型を新しく定義しています。

また，SELECT 文中に演算処理がある場合は，その結果を考慮してストアドプロシジャの作成時に，結果に合ったデータ型を RESULT 句で用意しておきます。

2 データベースアクセス

DABroker では，C++によるアプリケーション開発向けのプログラミングインタフェースとして，簡易版と詳細版の二つの C++クラスライブラリを提供します。この章では，クラスライブラリを使用した，データベースアクセス方法について説明します。

2.1　データベースへの接続と切断

この節では，データベースへの接続と切断の方法を説明します。

各メソッドの引数などについては，「4. 簡易版関数詳細」及び「5. 詳細版関数詳細」を参照してください。

2.1.1　データベースへの接続

アプリケーションでは，接続先データベース定義ファイルのデータベース種別名及びデータベース名を使用してデータベースに接続します。接続先データベース定義ファイルを使うことによって，接続条件に変更が発生した場合などでも，アプリケーションを変更することなく，接続するデータベースの変更ができます。

(1)　接続先データベース定義ファイルの作成

データベースに接続するために，接続先データベース定義ファイルを作成しておく必要があります。接続先データベース定義ファイルの作成方法については，マニュアル「DABroker」を参照してください。

次に，接続先データベース定義ファイルの定義形式の例を示します。

(定義形式の例)

```
DBDEF_HIR1{                          # データベース種別名
    DBTYPE=hirdb                     # データベース種別
    db01{                            # データベース名
        DBHOST=host01                # データベースのホスト名
        DBNAME=22200                 # データベースのポート番号
        USRID="uid"                  # ユーザID
        PASSWD="passwd"              # パスワード
    }
}
```

TPBroker for C++の OTS 機能を使用してトランザクション制御を行う場合，接続先定義ファイルは使用できません。

(2)　データベースとの接続

(a)　簡易版クラス

アプリケーションからデータベースアクセスを行うためには，DBRDatabase クラスのコンストラクタで，接続先データベース定義ファイルのデータベース種別名を指定し，Database オブジェクトを生成します。次に，Database オブジェクトの Connect メソッドの引数にデータベース名やユーザ ID，パスワードを指定しデータベースと接続します。

```
////////////////////////////////////////////////////////
///  単一データベースとの接続と切断の例
DBRDatabase *pDB1;
     pDB1 = new DBRDatabase(         // Databaseオブジェクトを生成
                      "DBDEF_HIR1",  // データベース種別名称
                       NULL,0);
// データベースとの接続
     pDB1->Connect("UID",                      // ユーザID
                   "PASS",                     // パスワード
                    NULL,NULL,
                   "db01",                     // データベース名
                    LOCK_OPT_WAIT,0,STMT_SYNC);
//   データベースとの接続を切断
     pDB1->close();                        // データベースとの接続を切断
```

(b)　詳細版クラス

アプリケーションからデータベースと接続を行うには，DBDriverManager クラス， DBDriver クラス，DBConnection クラスを使用します。

まず, DBDriverManager クラスからトップオブジェクトである DriverManager オブジェクトを生成する必要があり，これは，new 演算子，又は自動変数として生成します。※

次に，DriverManager オブジェクトの Driver メソッドを呼び出して Driver オブジェクトを生成します。この Driver メソッドの引数には，接続先データベース定義ファイルで定義してある，データベース種別名を指定します。

Driver オブジェクトを生成後，接続先のデータベース名やユーザ ID，パスワードを指定した Connect メソッドを呼び出してデータベースと接続し，Connection オブジェクトを生成します。

注※　DriverManager オブジェクト以外のオブジェクトは必ず，上位オブジェクトの生成メソッドを使って，生成してください。また，必ず削除メソッドを使って，削除してください。

```
////////////////////////////////////////////////////////////
///  単一データベースとの接続と切断の例
DBDriverManager*  pDrvMan;
DBDriver * pDriver = NULL;
DBConnection * pConnect = NULL;

//データベース接続
      pDrvMan = new DBDriverManager;
                              //DriverManagerをnew演算子で生成
      pDrvMan->InitializeMessage();
      pDriver = pDrvMan->Driver("DBDEF_HIR1");
                                    // データベース種別名の設定
      pConnect = pDriver->Connect("Con1",           // オブジェクト名
                                    "UserID",    // ユーザID
                                    "Password",  // パスワード
                                    NULL,NULL,
                                    "db01",      // データベース名
                                    LOCK_OPT_WAIT,0,STMT_SYNC);
//データベース切断
      pDriver->RemoveConnection("Con1");      // データベース切断
      pConnect = NULL;
      delete pDrvMan;                         // トップオブジェクトの削除
      pDrvMan = NULL;
```

(3)　複数データベースへの接続

一つのアプリケーションから，複数のデータベースをアクセスできます。例えば，HiRDB で作成した支店データベースと全社データベースというような，複数データベースにアクセスできます。また，HiRDB と ORACLE といった，複数種別のデータベースにもアクセスできます。複数のデータベース間の整合性を保ちながらアクセスする場合は，TPBroker for C++の OTS 機能を使用したトランザクション制御が必要です。

(a)　データベースとオブジェクトの関係

簡易版クラスでは，データベースごとに DBRDatabase クラスで Database オブジェクトを生成し，Database オブジェクトの Connect メソッドを使用してデータベースと接続します。

詳細版クラスでは，データベース種別ごとに DBDriver オブジェクトを生成し，データベースごとに，Connect メソッドでデータベースと接続し，DBConnection オブジェクトを生成します。

(b)　接続できるデータベースの数

DABroker では同時に接続できる数に制限はありませんが，接続できる数は，データベースの同時ユーザ数(そのデータベースに同時にログインできるユーザ数の制限)に依存します。

```
//////////////////////////////////////////////////////////
//   複数のデータベースにアクセスする例
//   オブジェクトの生成
     DBRDatabase db1("DBDEF_HIR1",NULL,0);
     DBRDatabase db2("DBDEF_RDALINK",NULL,0);
     DBRDatabase db3("DBDEF_RDALINK",NULL,0);
//   データベースとの接続
     db1.Connect("UserID","password",         // ユーザID，パスワード
                  NULL,NULL,
                 "db01",                          // データベース名
                  LOCK_OPT_WAIT,0,STMT_SYNC);
     db2.Connect("UserID","password",         // ユーザID，パスワード
                  NULL,NULL,
                 "db02",                          // データベース名
                  LOCK_OPT_WAIT,0,STMT_SYNC);
     db3.Connect("UserID","password",         // ユーザID，パスワード
                  NULL,NULL,
                 "db03",                          // データベース名
                  LOCK_OPT_WAIT,0,STMT_SYNC);

//   データベースとの接続を切断
     db1.close();                          // データベースとの接続を切断
     db2.close();
     db3.close();
```

2.1.2　データベースとの切断

簡易版クラスでは，Database オブジェクトの Close メソッドにより，データベースが切断されます。再接続するには再度 Connect メソッドを実行してください。Database オブジェクトは，Close メソッドを呼んでも削除されず，アプリケーションが終了するとき自動的に削除されます。

詳細版では，該当する DBConnection オブジェクトを削除することでデータベースが切断されます。一時的にデータベースとの接続を切り離したい場合は，DBConnection クラスの Close メソッドを使います。再接続には Connect メソッドを使ってください。

DBConnection オブジェクトの削除には，DBDriver クラスの RemoveConnection メソッド又は DBConnection クラスの Remove メソッドを使います。最後に，トップオブジェクトである DBDriverManager オブジェクトを削除します。トップオブジェクトを new 演算子で生成した場合には，Delete 演算子で削除します。

具体的なコーディング例については，「2.1.1 データベースへの接続」を参照してください。

2.2　同期・非同期処理

データベースアクセスを同期処理で行うか，非同期処理で行うかをアプリケーションで選択できます。

2.2.1　同期・非同期処理を選択するメソッド

SQL の実行を同期・非同期処理を行えるメソッドについて説明します。

なお， TPBroker の OTS 機能を使用してトランザクションを制御する場合は，SQL 文の非同期処理が実行できないため，注意してください。

(1)　簡易クラス

データベースとの接続時に，SQL の実行を同期・非同期処理のどちらで行うかを，下記メソッドの引数で選択します。

- DBRDatabase クラス：Connect（データベースとの接続）

SQL の実行を同期処理で実行する場合は，Connect メソッドの引数に「STMT_SYNC」を指定します。非同期処理で実行する場合は，「STMT_ASYNC」を指定します。

（例）　**pCon1->Connect(…,STMT_ASYNC)**

同期・非同期処理の設定に従って，同期・非同期処理で実行されるメソッドを次に示します。

- DBRDatabase クラス：Connect，ExecuteDirect
- DBRResultSet クラス：Delete，Execute，Next，Open，PageNext，Refresh，Update

(2)　詳細クラス

データベースとの接続時に，SQL の実行を同期・非同期処理のどちらで行うかを，下記メソッドの引数で選択します。

- DBDriver クラス：Connect（DBConnection オブジェクトの生成）
- DBConnection クラス：Connect（Close 後の再接続）

SQL の実行を同期処理で実行する場合は，Connect メソッドの引数に「STMT_SYNC」を指定します。非同期処理で実行する場合は，「STMT_ASYNC」を指定します。

ここで設定した同期・非同期処理は，データベースと接続している間有効です。

（例）　**pConnect=pDriver->Connect(…,STMT_ASYNC)**

同期・非同期処理の設定に従って，同期・非同期処理で実行されるメソッドを次に示します。

- DBDriver クラス
  Connect
- DBConnection クラス
  Connect，ExecuteDirect
- DBStatement クラス
  Execute，GetResultSet
- DBPreparedStatement クラス

ExecuteUpdate，GetResultSet

- DBCallableStatement クラス
  Execute
- DBResultSet クラス
  PageNext，Refresh，Update，Delete
- DBDatabaseMetaData クラス
  GetTables，GetColumns，GetProcedures，GetProcedureColumns，GetPrimaryKeys

2.2.2　非同期処理の完了確認

(1)　簡易クラスでの非同期処理完了確認

非同期処理の完了を待つには，DBRDatabase クラスと DBRResultSet クラスの WaitForDataSource メソッドを使用します。このメソッドが呼び出されると，メソッドが呼び出されたオブジェクトで実行中の非同期処理すべてが終わるまで待ちます。

（例） WaitForDataSource メソッドによる完了待ち

```
if(pDB1->WaitForDataSource())   // 非同期処理が終わるまで待つ
{
printf("非同期で要求した処理が終了しました");
}
```

非同期処理の完了を確認するには，InWaitForDataSource メソッドを使用します。

DBRDatabase クラスの場合，Database オブジェクトに関係するすべての ResultSet オブジェクトのアクセス完了を意味し，DBRResultSet クラスの場合，単独の ResultSet オブジェクトのアクセス完了を意味します。

また，DBRDatabase クラスの InExecute メソッドでも確認することができます。

例えば，Database オブジェクトの ExecuteDirect メソッドの処理は終了していても，ResultSet オブジェクトの Execute メソッドが実行中ならば，InWaitForDataSource メソッドは TRUE を返します。

（例） InWaitForDataSource メソッドによる完了確認

```
if(pDB1->InWaitForDataSource())
{
  printf("非同期処理要求が残っています");
  if(pDB1->WaitForDataSource(10000))            // 10秒待つ
  {
     printf("非同期処理がすべて終了しました");
  }
  else
  {
    printf("10秒経ちましたが，非同期処理継続中です。
                                        終了までお待ちください。");
    pDB1->WaitForDataSource();    // 非同期処理が終わるまで待つ
  }
}
```

(2)　詳細クラスでの非同期処理完了確認

非同期処理の完了を待つには，各クラスの WaitForDataSource メソッドを使用します。このメソッドが呼び出されると，メソッドが呼び出されたオブジェクトで実行中の非同期処理すべてが終わるまで待ちます。

（例） WaitForDataSource メソッドによる完了待ち

```
if(pStatement->WaitForDataSource())
                                        // 非同期処理が終わるまで待つ
{
 printf("非同期で要求した処理が終了しました");
}
```

非同期処理の完了を確認するには，DBConnection オブジェクトの InWaitForDataSource メソッドや各クラスの InExecute メソッドを使用します。これは， DBConnection クラスのメソッドに限らず，どのクラスのメソッドであっても，その DBConnection オブジェクトから生成されたオブジェクトのメソッドでデータベースアクセス中であれば，TRUE（非同期処理継続中）が返ります。

例えば，DBConnection オブジェクトの ExecuteDirect メソッドの処理は終了していても，DBStatement オブジェクトの Execute メソッドが実行中ならば，InWaitForDataSource メソッドは TRUE を返します。

（例） InWaitForDataSource メソッドによる完了確認

```
if(pConnect->InWaitForDataSource())
{
  printf("非同期処理要求が残っています");
  if(pConnect->WaitForDataSource(10000)) {      // 10秒待つ
    printf("非同期処理がすべて終了しました");
  } else {
    printf("10秒経ちましたが，非同期処理継続中です。
                                        終了までお待ちください。");
    pConnect->WaitForDataSource()
                              //非同期処理が終わるまで待つ
  }
}
```

2.2.3　非同期処理実行中にエラーとなるメソッド

非同期処理中は，基本的にデータベースアクセス処理が完了するまで，次のメソッドの呼び出しを除き，データベースアクセス処理を要求しているオブジェクトの操作はできません。

- 非同期処理中かどうかを確認するメソッド
- 非同期処理の完了を待つメソッド

他のメソッドを要求すると，エラーがスローされます。

2.3　トランザクションと排他制御

トランザクションの制御は，次の三つの方式に分けられます。

- 一つの DBMS を対象にトランザクション制御を行う方法
  DABroker for C++のクラスライブラリを使ってトランザクションを制御します。
- 複数の DBMS を対象にトランザクション制御を行う方法（TPBroker の OTS 機能を使用する場合）
  TPBroker の OTS 機能を使ってトランザクションを制御します。
- 複数の DBMS を対象にトランザクション制御を行う方法（OpenTP1 を使用する場合）
  OpenTP1 を使ってトランザクションを制御します。

2.3.1　一つの DBMS を対象にトランザクション制御を行う方法

(1)　トランザクションの開始

データベースを更新する場合，クライアントからの一つの要求に対しては，必ず一つのトランザクションにすべきです。また，注意すべきことは，トランザクションはデータベース単位となる点です。複数のデータベースに対して同時に Commit や Rollback することはできません。

簡易版クラスを使用した場合のトランザクションは，Connect メソッドを実行するとそのオブジェクトはトランザクション開始状態になります。したがって，自動コミットは利用できません。

詳細版クラスでは，明示的にトランザクションを制御する方法と自動コミットの 2 通りの利用方法が選択できます。

トランザクションを制御するには，DBTransaction クラスの，BeginTrans メソッド，Commit メソッド，Rollback メソッドを使用して制御します。まず，DBDriverManager オブジェクトの Transaction メソッドを呼び出して，DBTransaction オブジェクトを生成します。この DBTransaction オブジェクトは，DBConnection クラスの RegisterTransactions メソッドを利用して DBConnection オブジェクトと関連付けます。次に，DBTransaction オブジェクトの BeginTrans メソッドを呼び出すか，SetAutoCommit メソッドを FALSE で呼び出すことでトランザクションを開始します。

トランザクションを使用しないでデータベースアクセスを行った場合，SQL 文を一文実行するごとにコミットします（自動コミット）。これはデータベースを参照する場合にだけ利用し，データベースの更新がある場合は，必ずトランザクション制御を行ってください。

(2)　コミットとロールバック

簡易版の場合，DBRDatabase クラスの Commit メソッド，Rollback メソッドを呼び出すことによってトランザクションが終了します。

詳細版の場合，DBTransaction オブジェクトの Commit メソッド，Rollback メソッドを呼び出すことによりトランザクションが終了します。

トランザクションを終了させないで， データベースとの接続を切断した場合は，アプリケーションの処理が異常であると認識し，DABroker がロールバック要求しますので注意が必要です。

(3)　サンプルコーディング

コミット，及びロールバックを使用したトランザクションのサンプルコーディングを次に示します。

（例１） 簡易版クラスのトランザクション

```
////////////////////////////////////////////////////////
///  クラスライブラリによるトランザクション制御（簡易版）
DBRDatabase *pDB1;
try
 {
pDB1 = new DBRDatabase("DBDEF_HIR1", NULL,0);
                                        // オブジェクトの生成
  pDB1->Connect("UID","PASS",
                                        // データベースと接続
                NULL,NULL, "db01",    // データベース名
                LOCK_OPT_WAIT,0,STMT_SYNC);
                          // 自動的にトランザクションの開始
    :   // データ操作など，その他の処理
  pDB1->Commit();          // 処理が正常に終了した場合コミット
    :
 }
 catch(DBSQLCA e)
 {
    cout << "エラーが発生しました。¥n" << e.ErrorMessage << "¥n"
                                                    << flush;
    if(e.e_SQLERROR)
    {
      cout << e.e_SQLERROR << "¥n" << flush;
                                      // エラーメッセージの取得
    pDB1->Rollback();                 // 処理中ならばロールバック
 }
  pDB1->close();                      // コネクションの解放
 }
```

（例２） 詳細版クラスのトランザクション

```
////////////////////////////////////////////////////////
///  クラスライブラリによるトランザクション制御（詳細版）

DBDriverManager*  pDrvMan;
DBDriver*  pDriver;
DBConnection* pCnct;
DBTransaction* pTrns;
  try
  {
    pDrvMan = new DBDriverManager;
                        //DBDriverManagerをnew演算子で生成
    pDrvMan->InitializeMessage();
    pDriver = pDrvMan->Driver("DBDEF_HIR1");
                                  // データベース種別名の設定
    pTrns = pDrvMan->Transaction();
                           // DBTransactionオブジェクト生成
    pCnct = pDriver->Connect("Con1",      //データベース接続
                             "UserID","Password",NULL,NULL,
                             "db01",LOCK_OPT_WAIT,0,STMT_SYNC);
    pCnct->RegisterTransactions(pTrns);
                                  //オブジェクトの関連付け
    pTrns->BeginTrans();          // トランザクションの開始
    :  // データ操作など，その他の処理
    pTrns->Commit();     // 処理が正常に終了した場合コミット
    :
  }
  catch(DBSQLCA e)
  {
    cout << "エラーが発生しました。¥n" << e.ErrorMessage << "¥n"
                                                    << flush;
    if(e.e_SQLERROR)
      cout << e.e_SQLERROR << "¥n" << flush;
                                      // エラーメッセージの取得
  {
     pTrns->Rollback();            // 処理中ならばロールバック
  }
  if(pCnct)
  {
    pDriver->RemoveConnection("Con1");    // コネクションの解放
    pCnct = NULL;
}
```

2.3.2　複数の DBMS を対象にトランザクション制御を行う方法（TPBroker の OTS 機能を使用する場合）

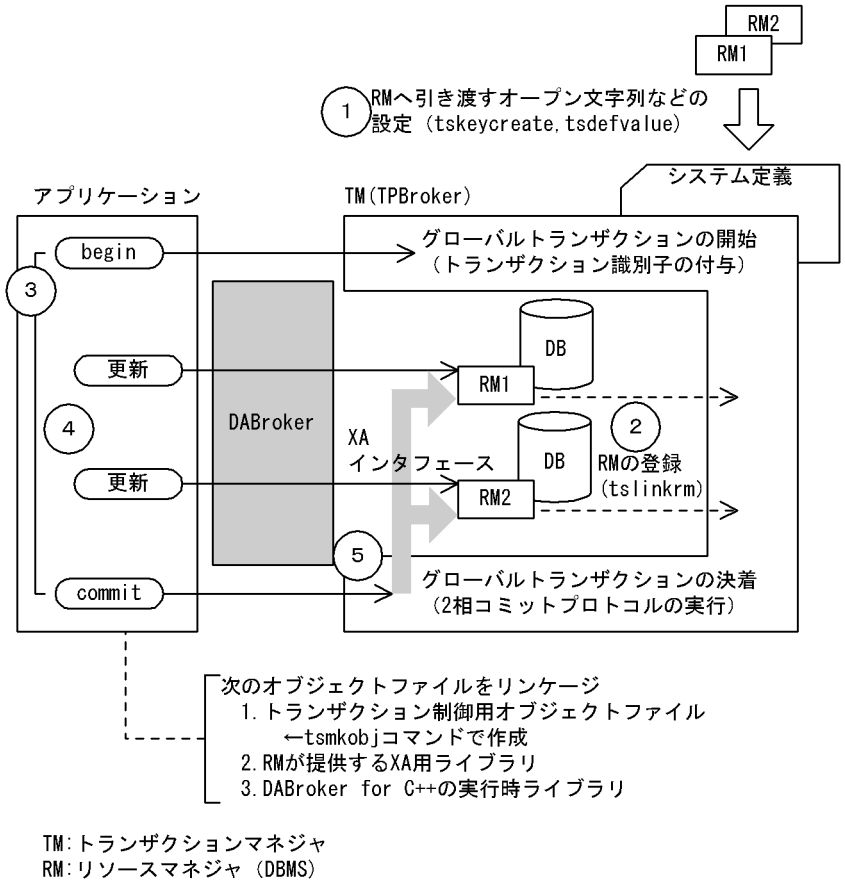
この方法では，DBMS へのアクセスは一つの DBMS を対象にトランザクション制御を行う方法と同じで，トランザクション制御の部分で TPBroker for C++が提供する OTS 機能を使用します。

OTS 機能は Red Hat Linux 版では使用できません。

(1)　動作概要

複数の DBMS を対象にトランザクション制御を行う場合の動作概要を図 2-1 に示します。TPBroker のトランザクション制御では， X/Open で規定された DTP モデルに準拠し，ユーザアプリケーションをアプリケーションプログラム（AP），DBMS をリソースマネジャ（RM），トランザクション制御を行う TPBroker をトランザクションマネジャ（TM）と呼びます。

図 2-1　複数の DBMS を対象にトランザクション制御を行う場合の動作概要



TM:トランザクションマネジャ
RM:リソースマネジャ（DBMS）

複数 DBMS に対してトランザクション制御する場合（③）は，TPBroker のメソッドを使ってトランザクションの開始・決着を記述し，その間の DBMS へのアクセス（④）は DABroker for C++のメソッドを使います。このとき DABroker for C++のメソッドの延長で DABroker がトランザクション識別子を取得し，SQL へ付加して DBMS へアクセスします。つまり，DABroker for C++のメソッドを使用する CORBA オブジェクトは別々であっても，トランザクション識別子が引き継がれるような仕組みになっています。

次に図 2-1 の丸付き数字の順に，必要になる準備，アプリケーションの処理手続きなどを説明します。TPBroker のコマンドの詳細については TPBroker のマニュアル「トランザクショナル分散オブジェクト基盤 TPBroker ユーザーズガイド」を参照してください。

(2)　リソースマネジャの動作環境の設定（①）

TPBroker の機能（コマンドなど）を使用してリソースマネジャ（DBMS）の XA インタフェースを利用するために必要な情報を TPBroker のシステム定義へ設定します。

TPBroker のシステム定義には，システム環境定義やプロセス監視定義があります。DBMS ごとに固有な情報なので，使用する DBMS のドキュメントを参照して登録してください。

例えば，システム環境定義の中のリソースマネジャ定義では，X/Open の規格に従った，オープン文字列，クローズ文字列，環境変数など DBMS で解釈される情報を設定します。設定には TPBroker の tskeycreate，tsdefvalue コマンドを使います。

(3)　リソースマネジャの登録（②）

アプリケーションで使用するリソースマネジャ（DBMS）をトランザクションマネジャへ TPBroker の tslnkrm コマンドを使って登録します。登録方法には，動的登録と静的登録があります。

(4)　アプリケーションでの処理手続き（③，④）

(a)　グローバルトランザクションの開始・決着

アプリケーションでは，TPBroker が提供するメソッドを使ってトランザクションの開始と決着を記述します。このトランザクションのことを TPBroker ではグローバルトランザクションと呼んでいます。

グローバルトランザクションの開始と決着は TPBroker が提供する次のメソッドを使います。

#	処理	使用する TPBroker 提供のメソッド
1	開始	CosTransactions::Current::begin()メソッド
2	決着(コミット)	CosTransactions::Current::commit()メソッド
3	決着(ロールバック)	CosTransactions::Current::rollback()メソッド

(b)　DBMS へのアクセス手続き

DBMS へのアクセスは，一つの DBMS を対象にトランザクション制御を行う方法と同様ですが，オブジェクトの生成方法が一部変わります。詳細は次の個所を参照してください。

- 簡易版クラスを使用する場合
  DBRDatabase クラスのコンストラクタ
  DBRDatabase クラスの Connect メソッド
- 詳細版クラスを使用する場合
  DBDriverManager クラスの Driver メソッド
  DBDriver クラスの Connect メソッド

この指定によって，そのオブジェクト，すなわち DBMS へのアクセスはグローバルトランザクションの制御対象になるわけです。指定しない場合は対象からはずれるので，DABroker が提供しているトランザクション用のメソッド（DBTransaction）を使って制御します。

(c)　アプリケーションのリンケージ

作成したアプリケーションのリンケージの際に，次の三つのオブジェクトファイルを付け加えてください。

- トランザクション制御用オブジェクトファイル

  TPBroker の tsmkobj コマンドでアプリケーションが使用している DBMS を対象にトランザクション制御で必要になる XA スイッチリストを含むオブジェクトファイルを作成します。

  例)　リソースマネジャ名が Oracle_XA の場合

  tsmkobj -o oracle_xa -r Oracle_XA
- DBMS が提供する XA 用ライブラリ

  DBMS のドキュメントを参照して指定してください。
- DABroker for C++が提供するライブラリ

(5)　グローバルトランザクション（⑤）

グローバルトランザクションの開始によってトランザクションの識別子が付与され，アプリケーションと DBMS の間でのアクセスは同じトランザクションの識別子で管理されます。コミットによって該当する識別子を対象に XA インタフェースを使ってトランザクションマネジャと DBMS の間で 2 相コミットプロトコルが実行されます。

(6)　サンプルコーディング

（例 1）簡易版クラスのトランザクション

```
////////////////////////////////////////////////////////
///  TPBrokerによるトランザクション制御（簡易版）
#include "tsport_c.hh"
#include "tpcosots_c.hh"
#include "dbbroker.h"

DBRDatabase *pDB1
DBRDatabase *pDB2

int main(int argc,char* argv[])
{
  pDB1 = NULL;
  pDB2 = NULL;

  try
  {
    CORBA::ORB_var orb=CORBA::ORB_init(argc,argv);
    CORBA::BOA_var boa=orb->BOA_init(argc,argv);
    CORBA::Object_var obj=orb->resolve_initial_referrences
                                  ("TransactionCurrent");
    CosTransactions::Current_var current=
                  CosTransactions::Current::_narrow(obj);

current->begin();      // グローバルトランザクションの開始

    try
    {
      pDB1 = new DBRDatabase(DRV_TYPE_ORACLE7,"XA");
      pDB1->Connect("orauser", "orapw1", "ORA1");
      pDB1->ExecuteDirect("INSERT INTO SAMPLETABLE
                                    VALUES(1,'ABC')");

      pDB2 = new DBRDatabase(DRV_TYPE_ORACLE7,"XA");
      pDB2->Connect("orauser2", "orapw2", "ORA2");
      pDB2->ExecuteDirect("INSERT INTO SAMPLETABLE2
                                    VALUES(2,'DEF')");

current->commit(1);
            // グローバルトランザクションのコミット
```

```
      pDB1->Close();
      pDB2->Close();
    }
    catch (DBSQLCA& ca)
    {
      current->rollback();
             // グローバルトランザクションのコミット
      // エラー処理
    }
  }
  catch(CORBA::SystemExecution se)
  {
    //  エラー処理
  }
  if(pDB1)
    delete pDB1;
  if(pDB2)
    delete pDB2;
  return();
}
```

（例２）詳細版クラスのトランザクション

```
////////////////////////////////////////////////////////
///  TPBrokerによるトランザクション制御（詳細版）
#include "tsport_c.hh"
#include "tpcosots_c.hh"
#include "dbbroker.h"

DBDriverManager * pDrvMan;

int main(int argc, char* argv[])
{
  pDrvMan = NULL;
  try
  {
    CORBA::ORB_var  orb = CORBA::ORB_init(argc, argv);
    CORBA::BOA_var  boa = orb->BOA_init(argc, argv);
    CORBA::Object_var  obj = orb->resolve_initial_references
                                    ("TransactionCurrent");;
    CosTransactions::Current_var  current =
                     CosTransactions::Current::_narrow(obj);

    current->begin();    // グローバルトランザクションの開始

    try
    {
      pDrvMan = new DBDriverManager;
      pDrvMan->InitializeMessage();

// OTS機能を使用したトランザクション制御の使用
      DBDriver*  pDriver =
                    pDrvMan->Driver(DRV_TYPE_ORACLE7,"XA");

      DBConnection*  pConnect1 = pDriver->Connect
                ("Sample1", "orauser1", "orapw1", "ORA1");
      DBConnection*  pConnect2 = pDriver->Connect
                ("Sample2", "orauser2" , "orapw2","ORA2");
      pConnect1->ExecuteDirect
              ("INSERT INTO SAMPLETABLE VALUES (1,'ABC')");
      pConnect2->ExecuteDirect
              ("INSERT INTO SAMPLETABLE VALUES (2,'DEF')");

      current->commit(1);     // トランザクションのコミット

      pConnect1->Close();
      pConnect2->Close();
    }
    catch (DBSQLCA& ca)
    {
      current->rollback();
                        // トランザクションのロールバック
      //エラー処理
    }
  }
```

```
    catch (CORBA::SystemException se)
    {
      //エラー処理
    }
    if(pDrvMan)
      delete pDrvMan;
    return 0;
  }
```

(7)　DBStatement オブジェクトの使用可能範囲

DBStatement オブジェクト，DBPreparedStatement オブジェクト，DBCallableStatement オブジェクトについては，複数のサーバメソッド間で共有しないようにしてください。同じＲＭに対する操作であっても，異なるスレッドで起動されたメソッドでは正常に動作しません。あるメソッドで生成した DBStatement オブジェクト（DBPreparedStatement オブジェクト，DBCallableStatement オブジェクト）はそのメソッド内で削除するようにしてください。

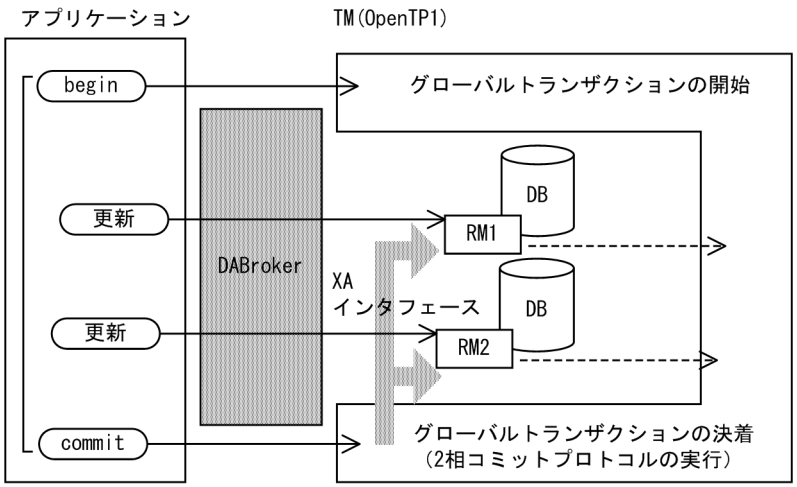
2.3.3　複数の DBMS を対象にトランザクション制御を行う方法（OpenTP1 を使用する場合）

この方法では，DBMS へのアクセスは一つの DBMS を対象にトランザクション制御を行う方法と同じで，トランザクション制御の部分で TPBroker for C++が提供する OpenTP1 を使用します。

(1)　動作概要

OpenTP1 を使用してトランザクション制御を行う場合の動作概要を図 2-2 に示します。OpenTP1 のトランザクション制御は，X/Open で規定された DTP モデルに準拠しており，ユーザアプリケーションをアプリケーションプログラム（AP），HiRDB をリソースマネジャ（RM），トランザクション制御を行う OpenTP1 をトランザクションマネジャ（TM）と呼びます。

図 2-2　OpenTP1 を使用してトランザクション制御を行う場合の動作概要



TM:トランザクションマネジャ
RM:リソースマネジャ（HiRDB）

グローバルトランザクションを制御する場合，AP は OpenTP1 に対してトランザクションの開始・決着の要求を出します。このとき DABroker for C++の DBTransaction クラスは使用せず，OpenTP1 が提供するインタフェースを使用します。グローバルトランザクションの制御対象となっているデータベースに対して検索・更新処理をする場合は DABroker for C++のメソッドを使います。

HiRDB を使用する分散トランザクション環境の構築やグローバルトランザクションの制御方法については，OpenTP1 のマニュアル，及び HiRDB のマニュアルを参照してください。

(2)　OpenTP1 を使用する場合の DABroker for C++の操作

HiRDB を使用した分散トランザクション環境で, SQL の処理をグローバルトランザクションの制御対象とするためには，DABroker for C++のメソッドを次のように使用する必要があります。

- データベースへのアクセスをグローバルトランザクションの制御対象とした場合
  トランザクション制御は OpenTP1 が提供するインタフェースを使うため, DABroker for C++が提供している DBTransaction クラスは使用しません。
- DBRDatabase クラスを使用している場合
  Commit メソッドと Rollback メソッドは使用できません。これらのメソッドを使用しても，トランザクションの制御は行いません。
- グローバルトランザクションの制御対象としない場合
  DBTransaction クラスを使用してトランザクションを制御します。
- グローバルトランザクションの制御対象とするためにデータベースに接続する場合
  接続先データベース定義ファイルによる接続はできません。
- OpenTP1 のトランザクション制御を使用する場合
  次の関数の使用方法に注意してください。
  - DBRDatabase クラスのコンストラクタ（「4.3　DBRDatabase コンストラクタ」参照）
  - DBRDatabase クラスの Connect メソッド（「4.3　Connect メソッド」参照）
  - DBDriverManager クラスの Driver メソッド（「5.3　Driver メソッド」参照）
  - DBDriver クラスの Connect メソッド（5.4　Connect メソッド）

上記以外の動作についてはグローバルトランザクションの制御対象としない場合と変更ありません。

(3)　注意事項

(a)　排他制御に関する注意

複数のデータベースにアクセスする場合, 排他制御の範囲はデータベース単位です。複数のデータベースにわたる排他制御はできません。つまり，一つのデータベース内でのデッドロックは検出されますが，複数のデータベース間にわたる待ち状態は検出されないことになります。このため, 複数のデータベース間でデッドロックが発生するようなアプリケーションは設計しないでください。

(b)　HiRDB のライブラリについて

アプリケーションが使用する HiRDB のライブラリは，静的登録・動的登録どちらの場合も各 OS の複数接続機能に対応したシングルスレッド用のライブラリを使用してください。

(c)　スレッドについて

OpenTP1 上で動作するプログラムではスレッドを生成してはいけません。

DABroker for C++と HiRDB でもシングルスレッドに対応したライブラリを使用する必要があります。DABroker for C++が提供するシングルスレッドライブラリを使用する場合には，次の条件を満たす必要があります。

- DABroker for C++を使用するアプリケーション（プロセス）が，pthread ライブラリを使用しないこと。
- DABroker for C++のマルチスレッドを前提とする次の機能は使用しないこと。
  - 非同期処理機能
    詳細については「1.4.3　同期・非同期処理」を参照してください。
  - 2 バッファ方式
    詳細については「3.3.3　検索性能の向上策」を参照してください。

(d)　グローバルトランザクションの制御対象としないアプリケーションの作成について

- トランザクション制御
  グローバルトランザクションの制御対象としないアプリケーションでは，トランザクション制御は DBTransaction クラスを使用してください。
- HiRDB のライブラリについて
  アプリケーションが使用する HiRDB のライブラリは，各 OS に合ったライブラリを使用してください。
- シングルスレッドライブラリ

OpenTP1 上で動作するプログラムではスレッドを生成しないでください。DABroker for C++と HiRDB でもシングルスレッドに対応したライブラリを使用する必要があります。詳細については「3.3.4(3)スレッドについて」を参照してください。

2.3.4　排他制御

排他制御では，レコードやテーブルのロック方法を設定したり，データの入力方法までも設定します。この排他制御は，データの整合性やアプリケーションの実行性能に影響するものであり，正しく理解して利用することが必要です。特に，性能を重視する場合，ロックする対象や範囲を必要最小限とする必要があります。

また，利用するデータベースによって，この排他制御の機能や動作が異なりますので，注意が必要です。

(1)　レコード排他の種類

(a)　排他の設定方法

簡易クラスでは，ResultSet に検索結果を得る時，DBRResultSet クラスの Execute メソッドの引数で排他の種類を設定します。

詳細版クラスでは，ResultSet に検索結果を得る時，SetResultSetType メソッドの引数で排他の種類を設定します。

(b)　排他の種類

Execute メソッドや SetResultSetType メソッドで設定する排他の種類と目的について説明します。

- 更新目的の検索
  排他の種類として，TYPE_EXCLUSIVE を指定します。基本的に，SELECT 文には"FOR UPDATE"が付加され，検索したレコードはロックされます。ResultSet から更新する場合は，この排他を選択してください。TYPE_EXCLUSIVE は，すべてのデータベースに有効です。
- 参照目的の検索

TYPE_NONE を指定します。この指定は，SQL 実行時に排他オプションは付加されず，ロック方法は，対象となるデータベースのデフォルトとなります。

- ダーティリード

  TYPE_NOWAIT を指定します。この指定は，レコードにどの種類のロックが掛かっていても，レコードを検索する（ダーティリード）オプションです。このため，読み取りの一貫性は保証されません。また，HiRDB，XDM/RD，SQL Server に有効です。

(c)　ロックの有効範囲

簡易クラスでは，Execute メソッドでロックの方法を設定した場合に，ロック制御されるのは，Open メソッド実行時点から，コミット，又はロールバック実行終了までの間です。詳細版クラスでは，GetResultSet メソッド実行時点から，コミット又はロールバック実行終了までロック制御されます。

(2)　HiRDB，XDM/RD の排他

HiRDB，XDM/RD では，排他の種類ごとに下記オプションが付加されます。

- TYPE_EXCLUSIVE

  SELECT 文に"FOR UPDATE"が付加され，レコード排他は，WITH EXCLUSIVE LOCK が仮定されます。

- TYPE_NONE

  SELECT 文にオプションは付加されません。

- TYPE_WAIT

  "WITHOUT LOCK WAIT"が付加されます。このオプションでは，検索時に共有ロックされますが，参照が済んだレコードからロックが解除されます。

  TYPE_WAIT を指定した検索では，トランザクションの終了を待たせないで排他が解除されるので，同時実行性が向上します。

- TYPE_NOWAIT

  "WITHOUT LOCK NOWAIT"が付加されます。レコードにどの種類のロックが掛かっていても,又コミットされていないデータも検索します。このため,読み取りの一貫性は保証されません。

- TYPE_SHARED

  "WITH SHARE LOCK"が付加されます。このオプションでは，検索時に共有ロックされます。共有ロックの掛かったレコードは，ほかのトランザクションから参照できますが，更新はできません。

(3)　ORACLE の排他

ORACLE の場合，排他の種類ごとに下記オプションが付加されます。

- TYPE_EXCLUSIVE　… "FOR UPDATE"が付加されます。
- TYPE_NONE, TYPE_SHARED　… SELECT 文にオプションは付加されません。
- TYPE_WAIT　… 利用できません。
- TYPE_NOWAIT　… 利用できません。

ORACLE の場合，TYPE EXCLUSIVE だけですが，SQL の LOCK 文を実行（ExecuteDirect メソッド）することによっても，レコードのロック方法やテーブルのロック方法を設定できます。

下記に LOCK 文で設定できる排他オプションの概略について説明します。詳細については，ORACLE のドキュメントを参照してください。

- ROW SHARE／SHARE UPDATE
  レコードの共用モードです。参照ロックはできません。更新・追加・削除レコードはロックされます。
- ROW EXCLUSIVE
  レコードの排他モードです。参照ロックができ，更新・追加・削除レコードはロックされます。
- SHARE ROW EXCLUSIVE
  レコードの共用排他モードです。
- SHARE
  テーブルの共用モードです。参照ロックはできません。更新・追加・削除できません。
- EXCLUSIVE
  テーブル単位にロックを掛けます。

(4)　SQL Anywhere，Adaptive Server Anywhere の排他

このデータベースの場合，排他の種類ごとに下記オプションが付加されます。

- TYPE_EXCLUSIVE　… "FOR UPDATE"が付加されます。
- TYPE_NONE, TYPE_SHARED　… SELECT 文にオプションは付加されません。
- TYPE_WAIT　… 利用できません。
- TYPE_NOWAIT　… 利用できません。

SQL Anywhere，Adaptive Server Anywhere の場合，SQL の SET OPTION 文を実行（ExecuteDirect メソッド）することによっても，レコードのロック方法を設定できます。

下記に SET OPTION 文で設定できる排他オプションの概略について説明します。詳細については，データベースのドキュメントを参照してください。また，ロックモードを切り替える場合には，再度 SET OPTION 文を実行する必要があります。

- SET OPTION ISOLATION_LEVEL=n
  - n=0：参照ロックを使用しないため，ほかのトランザクションで排他ロックされているレコードも検索します。検索したレコードは参照ロックしません。
    排他ロックされているレコードの更新は待ちとなります。
  - n=1：参照ロックされていても，レコードを検索します。
    検索したレコードは，カーソルがあるレコードに対して参照ロックをかけますが，参照が済んだレコードから参照ロックを解除します。排他ロックされているレコードのアクセスでは，待ちとなります。
  - n=2：参照ロック，排他ロックされているレコードのアクセスで待ちとなります。
    検索したレコードには参照ロックをかけ，更新したレコードには排他ロックをかけます。
  - n=3：ISOLATION_LEVEL=2 の制御に加え，検索した条件を満たすレコードの追加，更新を抑止するためファントムロックをかけます。

オプションの詳細については，SQL Anywhere のドキュメントを参照してください。

(5)　SQL Server の排他

SQL Server の場合，排他の種類ごとに下記オプションが付加されます。

- TYPE_EXCLUSIVE　… "UPDLOCK"が付加されます。

Open メソッド実行時点からレコードに共有ロックを掛け, SQL で更新する時点で更新ロックを掛けます。共有ロックを掛けられたレコードは，ほかのトランザクションから参照することは可能ですが，更新できません。更新ロックが掛かるとほかのトランザクションから更新も参照もできません。

- TYPE_NONE, TYPE_SHARED　… SELECT 文にオプションは付加されません。
- TYPE_WAIT　… "HOLDLOCK"が付加されます。
  Open メソッド実行時点からトランザクションの終了まで共有ロックを掛けます。共有ロックの掛かったレコードは，ほかのトランザクションから参照できますが，更新はできません。
- TYPE_NOWAIT　… "NOLOCK"が付加されます。

(6)　SQL/K の排他

SQL/K の場合，排他の種類ごとに下記オプションが付加されます。

- TYPE_EXCLUSIVE　… "FOR UPDATE"が付加されます。
  FOR UPDATE 句の詳細については，マニュアル「SQL/K」を参照してください。
- TYPE_NONE，TYPE_WAIT，TYPE_NOWAIT，TYPE_SHARED　… SELECT 文にオプションは付加されません。

SQL/K の場合，SQL の SELECT 文に FOR UPDATE 句を指定することによっても，レコードのロック方法を設定できます。この場合，資源の競合時に共用しない設定にしていると，競合が解除されるまで待ち状態になります。

(7)　UPDATE，DELETE，INSERT 文によるロック

簡易クラスでは, DBRDatabase クラスの ExecuteDirect メソッドに UPDATE 文, DELETE 文, INSERT 文を使って更新・削除・追加することができます。

詳細クラスでは，DBStatement クラスの Execute メソッドなどに UPDATE 文，DELETE 文，INSERT 文を使って更新・削除・追加をすることができます。

UPDATE 文，DELETE 文，INSERT 文の場合には， ExecuteDirect や Execute メソッド実行開始からコミット，又はロールバック実行時点までの間，更新・削除・追加されたレコードがロックされます。

この UPDATE 文，DELETE 文で更新・削除対象のレコードは，いったん検索してから更新・削除されますので検索時はロックによる待ちが発生します。

SQL Server の場合は付属のドキュメントを参照してください。

(8)　LOCK 文によるテーブルのロック

簡易クラスの場合，DBRDatabase クラスの ExecuteDirect メソッドで LOCK 文を実行することにより，テーブルをロックすることができます。

詳細クラスの場合，DBConnection オブジェクトの ExecuteDirect メソッド，DBStatement オブジェクトの Execute メソッド，DBPreparedStatement オブジェクトの Execute メソッドから LOCK 文を実行することにより，テーブルをロックすることができます。

テーブルのロック範囲は，LOCK 文を実行した時点から，コミット，又はロールバック実行終了までの間です。

LOCK 文でテーブルをロックすると，レコード単位のロックをしないため，オーバヘッドが大幅に削減できますが，同時に動作できるアプリケーションの多重度が下がります。

テーブル単位にロックを掛けるためには，LOCK 文でテーブルを EXCLUSIVE モードとします。

（例）hirdb.ExecuteDirect("LOCK TABLE TABLE1 IN EXCLUSIVE");

SQL Server では，SELECT 文の排他オプションを使ってテーブルをロックし，詳細版クラスの場合は，SetResultSetType メソッドの設定でテーブルをロックすることもできます。「2.3.3(5)SQL Server の排他」を参照してください。

SQL Anywhere,Adaptive Server Anywhere の場合はテーブルのロックは使用できません。

(9)　排他エラーとデッドロック

(a)　排他エラー時の動作

ロックを掛けようとしたときに，ほかのトランザクションによって該当するレコードやテーブルがロックされていた場合の動作として，ロックが解除されるまで待つか，エラーを報告するかのどちらかの動作を選択できます。この選択は，データベース接続時に Connect メソッドの引数で次の指定をします。

- LOCK_OPT_WAIT
  ロックが解除されるまで待つ
- LOCK_OPT_NOWAIT
  ロックされていた場合は，すぐに DB_ERROR_DRIVER_ERROR のエラーをスロー

これ以外に，ロックされていたときのエラースロー時に，ロールバックするかどうかを指定する，LOCK_OPT_WITH_ROLLBACK, LOCK_OPT_WITHOUT_ROLLBACK がありますが，使用する DBMS によって動作が異なるため，詳細については「4.簡易版関数詳細」又は「5.詳細版関数詳細」の Connect メソッドを参照してください。

簡易版クラスでは，DBRDatabse クラスの Connect メソッドで指定した値は，そのコネクションでのデフォルト値となります。また，ロックされていたときの動作は，DBRResultSet クラスの Execute メソッドの実行時にも指定できます。実行時に特に指定しない場合は，DBRDatabase クラスの Connect メソッドで指定した値が有効になります。

詳細版クラスでは，DBDriver クラスの Connect メソッドで指定した値がデフォルト値となります。また，ロックされていたときの動作は，DBStatement クラスの Execute メソッド，又は DBPreparedStatement クラスの Execute メソッドの実行時にも指定できます。実行時に特に指定しない場合は，Connect メソッドで指定した値が有効になります。

一般的なアプリケーションでは，LOCK_OPT_WAIT と設定し，ロックエラーを意識しないで，デッドロックだけを意識したコーディングにします。LOCK_OPT_NOWAIT を使用するアプリケーションでは，個々のデータベースアクセスごとに一定時間を空けてリトライをするなどのコーディングが余計に必要となります。

SQL Server の場合，引数の指定は無効です。ロックが解除されるまで待ちます。

(b)　排他エラーとデッドロックの判定

排他エラー時の動作で，LOCK_OPT_WAIT を指定した場合，ほかのトランザクションとの間でデッドロックが発生することがあります。この場合，DBMS からエラーが返され，DABroker for C++では DB_ERROR_DRIVER_ERROR をスローします。

トランザクション処理中にエラーがスローされたとき，DBSQLCA クラスの e_USERCODE，又は e_USERERROR メソッドに次に示す値が設定されます。

- 排他エラーが発生した場合

  (ロックしようとしたときに対象となるリソースが, 既にほかのトランザクションによって排他されていた場合)

  e_USERCODE に-20001，e_USERERROR に排他されていることを示すエラーメッセージが設定されます。

- デッドロックが発生した場合

  e_USERCODE に-20002，e_USERERROR にデッドロックが発生したことを示すエラーメッセージが設定されます。

ただし，RDA Link for Gateway を経由してメインフレーム系のデータベースをアクセスする場合には，e_USERCODE にエラーが返らないので注意が必要です。

(c)　デッドロック時の対応

アプリケーションでは，デッドロックが発生した場合に，どのように対処するかを決めておく必要があります。

基本的には，デッドロックが発生したときには，デッドロックが通知されたトランザクションは，ロールバックしてそのロックを解除することで，デッドロックを回避します。

その後の処理方法は，アプリケーションの形態によって異なりますが，特にバッチ処理，又は Web から起動されるアプリケーションの場合は，次のような処理をするのが一般的です。

1. デッドロックを検知したため，ロールバックを実行してロックを解除します
2. 一定時間を置いて，再度実行します
3. 一定の回数，繰り返して同様のエラーが起こるようであれば，アプリケーションの処理を終了させます

デッドロックが発生した場合のコーディング例については，「2.4 エラー処理」を参照してください。

(d)　効率を考えた参照と更新

複数のサーバアプリケーションから DABroker 及びデータベースにアクセスする場合は，使用するデータベースの排他範囲を最小限に押さえたり, データベースとの連絡回数を削減することを考慮しておく必要があります。

参照してから更新するような場合，次のような方法があります。

1. 更新したいデータを，まず参照専用の ResultSet で検索し，排他制御は掛けません。
2. 検索したデータの中から更新に必要なレコードを確認します。
3. 更新対象のレコードだけを更新可能な ResultSet で検索し，ロックを掛けます。
4. レコードを更新します

必要なレコードだけにロックが掛かるので, ほかのトランザクションからのアクセスへの排他を最小限に押さえられます。

(10)　TPBroker の OTS 機能を利用するケースの排他

TPBroker の OTS 機能を利用し，複数の DBMS をアクセスする場合，排他制御の範囲は DBMS 単位です。複数の DBMS にまたがる排他制御は行われません。つまり，一つの DBMS 内でのデッドロックは検出されますが，複数の DBMS 間にまたがる待状態は検出されないことになります。このため，複数の DBMS 間でデッドロックが発生するようなアプリケーションは設計してはなりません。

2.4　エラー処理

DABroker では，アプリケーションからのデータベースアクセス要求に対して，アプリケーションの処理続行が不可能なエラーが発生した場合，エラーをスローします。

エラーは，DBSQLCA クラスを利用して解析することができます。

特に，DABroker では，DBMS ごとに異なるエラー発生時のエラーコードの違いを吸収し，アプリケーションでは，DBMS を意識することなくエラー処理を記述できます。ただし，すべてのエラーコードを対象としている訳ではなく，アプリケーションのエラー処理の判断に必要な，次のコードだけを対象としています。

エラーの種類	e_USERCODE	e_USEREEROR
ロックエラー	-20001	更新またはロックしようとしたレコードは他のトランザクションによってすでにロックされています。
デッドロック	-20002	デッドロックが発生しました。
UserID/Password 不正※	-20003	ユーザ ID またはパスワードの指定が不正です。
テーブルのアクセス権限不足	-20004	テーブルに対するアクセス権限が不足しています。

注※

SQL/K の場合，接続時以外(例えば不正な構文の SQL 文を指定して DBStatement::Execute 関数を実行したとき)でもこのエラーコードとメッセージが返される場合があります。接続時以外でこのエラーコードとメッセージが返された場合は無視してください。

2.4.1　基本的なアプリケーションのエラー処理

データベースアクセス中のエラーは，C++の例外処理に通知されます。このため，例外処理の中でエラーコードを判断し，エラー後の処理を決定します。

アプリケーションの構造は，さまざまな構造が考えられますが，ここでは，メインの処理，データベースとの接続処理，データベースアクセス処理が分離されているという前提で，エラー処理を説明します。

デッドロックエラーの場合

まず，仕掛り中のトランザクションをロールバックします。その後，時間をおいてリトライして，同様のエラーが何度も起こる場合はアプリケーションの処理を中止します。

パスワード不正，アクセス権限不足の場合

アプリケーションからエラーメッセージを出力して終了します。

(1)　メイン処理

メイン処理では，外部から入力したユーザ ID とパスワードをデータベース接続関数へ渡し，次にデータベースアクセス関数を呼び出すものとします。

```
#define LockError -20001
#define DeadLock  -20002
#define UIDPwdInv -20003
////////////////////////////////////////////////////////
/// メイン処理
main(int argc, char** argv)
```

```
{
   retry=10;
        .....
 ユーザIDとパスワード入力
 データベース接続関数呼び出し
 while(retry > 0){
     try{
     データベースアクセス関数呼び出し
     }
     catch(DBSQLCA e) {
       if (e.e_USERCODE == DeadLock) {
         // デッドロックエラー
         pDB1->Rollback();
         cout << e.e_USERERROR << endl;
         cout << "3秒待ってリトライします" << endl;
         SLEEP(3);
         retry--;
       } else {
         retry = 0;
       }
     }
   }
}
```

(2)　データベース接続処理

メイン関数から渡されたユーザ ID とパスワードが正しくなかった場合，アプリケーションからエラーメッセージを出力し，接続を中止します。

```
////////////////////////////////////////////////////////
/// データベース接続処理
.....
   try{
   //データベース接続処理
   }
   catch(DBSQLCA e)
   {
     if (e.e_USERCODE == UIDPwdInv )
     {
       // USERID/Password不正
       cout << e.e_USERERROR << endl;
       cout << "接続に失敗しました。
          ユーザID又はパスワードが正しくありません。"<< endl;
     } else {
       .....
     }
   }
```

(3)　データベースアクセス処理

データベースアクセス処理中のエラーは，catch にスローされるので，e_USERCODE に設定されたエラーコードを判定します。データベースアクセスで判断するエラーで，デッドロックが発生した場合は，ロールバックしてロックを解除し，デッドロックを回避します。また，その後，複数回リトライしても同様のエラーが起こる場合は，アプリケーションを終了させます。

```
////////////////////////////////////////////////////////
// データベースアクセス処理
try{
      .....
      // 一つ目のテーブルをロック
      pRs1->Execute("SELECT IN,F_NAME FROM TEST",LOCK_OPT_WAIT);
      pRs1->open();
      // 二つ目のテーブルをロック
      pRs2->Execute("SELECT IN,F_COUNTRY FROM TEST2",LOCK_OPT_WAIT);
      pRs2->open();
   }
   catch(DBSQLCA e) {
     switch (e.e_USERCODE) {
       case DeadLock:    // デッドロックエラー
           throw e;
```

```
            case .....   :    // その他のエラー処理
                throw ....;
        }
    }
}
```

2.5　簡易版クラスのデータベースアクセス

ここでは，簡易版クラスを利用したデータベースアクセスの方法について説明します。

2.5.1　レコードの検索

レコードの検索には，DBRResultSet クラスのコンストラクタを利用して生成する ResultSet オブジェクトのメソッドを使用します。

レコードの検索は，SELECT 文を Execute メソッドで通知し，Open メソッドで検索を実行します。検索したレコードは，この ResultSet オブジェクトに格納されます。

レコードの検索には検索したレコードを参照だけする参照目的とレコードの更新または削除をする更新目的の２種類があり，この目的により検索するレコード数が異なります。

また，SELECT 文の組み立てでは，SELECT の一部を実行時に置き換える，?パラメタの利用も可能です。

(1)　参照目的の検索

参照を目的とした検索では，Execute メソッドの引数 swType に TYPE_NONE を指定する必要があります。レコードの検索は，Open メソッドを呼び出すことにより実行され，検索したレコードが ResultSet に読込まれます。ResultSet に読込むレコード数は，SetMaxRows メソッドで設定します。

検索条件に一致したレコードすべてを処理する場合は Next メソッドでカーソルを移動します。この場合，SetMaxRows メソッドで設定したレコード数を越えて Next メソッドが要求されると，次のレコードが自動的に ResultSet に読込まれます。

参照専用の ResultSet に検索結果を読込む例を示します。

```
////////////////////////////////////////////////////////
/// SQLの実行（検索）
DBRResultSet *pRs1;
  pRs1 = new DBRResultSet(pDB1);                  // オブジェクトの生成

  pRs1->SetMaxRows(10);                           // レコード数
  pRs1->Execute("SELECT F1, F2 FROM TABLE1",      // SELECT文の設定
                TYPE_NONE);                       // 参照目的
  pRs1->Open();                                   // レコードの検索
  while(!pRs1->IsEOF())
  {
    .....
    pRs1->Next();
  }
```

(2)　更新目的の検索

更新を目的とした検索では，Execute メソッドの引数 swType に TYPE_EXCLUSIVE を指定する必要があります。この場合，ResultSet には 1 レコードだけが読込まれます。

TYPE_EXCLUSIVE を指定した場合は，検索時の SELECT 文で FOR UPDATE オプションは指定する必要はありません。

注意事項

更新目的の検索の場合，下記例に示す SELECT 文を指定するとデータベースによりエラーとなります（Execute メソッド実行時にエラーがスローされる）。

- 集合関数(avg, count など)や組み込み関数を使用する場合

- DISTINCT を使用する場合
- GROUP BY 句や UNION 句を使用する場合
- 演算子を含む場合(UNION, INTERSECT, MINUS)
- JOIN を含む場合
- 上記に該当する VIEW を利用する場合

(3)　?パラメタを使用した検索

SELECT 文の一部を実行時に置き換える?パラメタを利用する場合，Execute メソッドの引数で?パラメタ付きの SELECT 文とします。?パラメタの値は，SetParam メソッドで指定します。レコードの検索は，Open メソッドを呼び出すことにより実行され，検索したレコードが ResultSet に読込まれます。

?パラメタを使用した例を示します。

```
////////////////////////////////////////////////////////
/// ?パラメタを使用した検索
DBRResultSet *pRs1;
  pRs1 = new DBRResultSet(pDB1);              // オブジェクトの生成

  pRs1->Execute("SELECT F1, F2 FROM TABLE1 ID > ? AND ID < ?",
                TYPE_NONE);
  pRs1->SetParam(1,100);        // １番目の?パラメタに対する値の設定
  pRs1->SetParam(2,300);        // ２番目の?パラメタに対する値の設定
  pRs1->Open();                 // レコードの検索
```

2.5.2　検索レコードの参照

ここでは，ResultSet に読込まれたレコードを参照する方法について説明します。参照目的で検索したレコードをアクセスする時には，カーソルを制御してカレントレコードを決定する必要がありますが，更新目的で検索した場合には，１レコードだけが読込まれますのでカーソルの制御は不要です。

(1)　参照目的レコードのカーソル制御

カーソルの移動には，次のメソッドを利用します。

- カーソルを次のレコードへ移動（Next メソッド）
- カーソルを前のレコードへ移動（Pervious メソッド）
- カーソルを n 番目のレコードへ移動（Absolute メソッド）
- カーソルを現在のレコードから n 個先のレコードへ移動（Relative メソッド）
- カーソルを最後のレコードへ移動（Bottom メソッド）
- カーソルを先頭のレコードへ移動（Top メソッド）
- カーソルを次の ResultSet の先頭レコードへ移動（PageNext メソッド）

ResultSet には，SetMaxRows メソッドで設定したレコード数だけが格納され，これより多くのレコードが検索されている場合は，Next メソッドや PageNext メソッドを呼び出し，次のレコードを ResultSet に読込むことが必要です。

レコードを先頭から順に参照する場合には，Next メソッドだけを利用することにより，次のレコードが自動的に ResultSet に読込まれます。

ResultSet に読込まれたレコードのカーソルを自由に移動するようなケースでは，次のレコードを ResultSet に読込むために，一度 Bottom メソッドで最後のレコードに位置付け，Next で読込む方法もあ

りますが，PageNext メソッドを使用することにより，カーソルの位置を意識せず，次のレコードを ResultSet に読込むことができます。

カーソルの移動は，ResultSet に格納されたレコードだけに対して有効であり，前に読込まれたレコードに位置づけることはできません。

カーソルが ResultSet のレコードの範囲を越えて指定された場合には，エラーがスローされます。アプリケーションでは，カーソルを移動する前後で IsEOF メソッドの戻り値を確認することで ResultSet の範囲を確認できます。

(2)　フィールド値の参照・データ変換

ResultSet のレコードのフィールド値は，DBRResultSet クラスの GetField メソッドを使って参照します。データベースのデータ型と GetField メソッドの引数に指定したデータ型とが異なる場合は，引数のデータ型に合わせて変換されます。データ型の変換規則については，「7. データ型」を参照してください。

```
////////////////////////////////////////////////////////
// 値の参照の例
LPTSTR          pField;
INT32           nField;
   //データベースのデータの検索処理
                    :
   // 検索結果をResultSetに取得
   // レコードの読み込み
   while(!pRs1->IsEOF())
   {
       // フィールドのデータの文字列変換と「ポインタ」の受け取り
           pRs1->GetField(1, &nField);
           cout << ”  Data=” << nField;
           pRs1->GetField(2, &pField);
           cout << ”  Data=” << pField;
       cout << endl;
       // 次のレコードの読み込み
       pRs1->Next();
   }
```

注意事項

- Null が格納されているフィールドに対して GetField メソッドを呼出すと取得データは意味のない値が戻されますので,Null が格納される可能性があるフィールドに対しては IsFieldNull メソッドでフィールドの内容が Null でないことを確認してください。
- 文字列から数値データ型への変換に失敗した場合には，0 が返されます。指定したフィールドがなかった場合や値が変換できない場合は，エラーをスローします。
- Execute メソッドで，TYPE_BLOB_FILE を指定した場合は，BLOB 型のフィールドに対する GetField メソッドの引数にファイル名称が戻されます。この場合, GetField メソッドの引数の型には，文字列と同様に，LPTSTR を指定してください。

2.5.3　レコードの更新

レコードを更新するには，検索したレコードを更新する方法（ResultSet を利用）と，UPDATE 文で直接更新する方法とがあります。

データの更新が完了した後で Commit メソッドを発行してデータベースの更新情報を確定させる必要があります。

(1)　検索したレコードの更新

レコードを検索し更新するためには,更新目的でレコードを検索する必要があります。検索されたレコードは ResultSet に 1 レコードだけ格納され，このレコードを更新するには，Edit メソッドで更新準備を指示し，SetField メソッドでフィールドの内容を変更し，Update メソッドで更新を指示します。複数のレコードを処理する場合には，Next メソッドで次のレコードを読込み，この処理を繰り返す必要があります。

SetField メソッドの引数に指定したデータ型とデータベースのデータ型とが異なる場合は，データベースのデータ型に合わせて変換されます。データ型の変換規則は「7. データ型」を参照してください。

```
////////////////////////////////////////////////////////
/// SQLの実行（更新）
DBRResultSet *pRs1;
  pRs1 = new DBRResultSet(pDB1);              // オブジェクトの生成

   // 更新目的で検索
  pRs1->Execute("SELECT F1, F2 FROM TABLE1", TYPE_EXCLUSIVE);

  pRs1->Open();                       // レコードの検索
  while(!pRs1->IsEOF())
  {
    // フィールド"F2"のデータを更新。uFunc()というユーザ関数を使用
    pRs1->Edit();
    pRs1->SetField("F2", uFunc());
    pRs1->Update();

    pRs1->Next();                     // 次のレコードの読み込み
  }
  pDB1->Commit();                     // 更新内容の確定
   .....
```

注意事項

接続する DBMS が SQL Server の場合，Close メソッドで ResultSet を削除してから Commit メソッドを呼出してください。Close メソッドを呼ばないと排他ロックが解除されません。

BLOB 型データの渡し方

Execute メソッドで, TYPE_BLOB_FILE を指定した場合は, BLOB 型のフィールドに対する GetField メソッドの引数にファイル名称が戻されます。この場合，GetField メソッドの引数の型には，文字列と同様に，LPTSTR を指定してください。

XDM/RD をご使用の場合の注意

XDM/RD をご使用の場合，LONG VARCHAR 型， LONG NVARCHAR 型，及び LONG MVARCHAR 型のフィールドのデータの更新はできません。SetField メソッドで値を設定しても，Update メソッドの実行時にエラーになります。

(2)　UPDATE 文を利用した更新

UPDATE 文を使ってレコードを更新できます。これは，既に更新するデータや内容が特定できる場合に，UPDATE 文で直接，更新個所及び値を指定する方法です。UPDATE 文を利用してデータを更新するには DBRDatabase クラスの ExecuteDirect メソッドを使用します。

```
////////////////////////////////////////////////////////
/// SQLの実行(更新) - DBRDatabaseクラスでUPDATE文を指定
    pDB1->ExecuteDirect("UPDATE TABLE1 SET F1=100 WHERE F2=300");
    pDB1->Commit();
```

2.5.4　レコードの削除

レコードの削除方法には，検索したレコードを削除する方法（ResultSet を利用）と，DELETE 文で直接削除する方法があります。

レコードの削除が完了した後で Commit メソッドを発行してデータベースの更新情報を確定させる必要があります。

(1)　検索したレコードの削除

レコードを検索し削除するためには,更新目的でレコードを検索する必要があります。検索されたレコードは ResultSet に 1 レコードだけ格納され，このレコードを削除するには，Edit メソッドで削除準備を指示し，Delete メソッドで更新を指示します。複数のレコードを処理する場合には，Next メソッドで次のレコードを読込み，この処理を繰り返す必要があります。

```
////////////////////////////////////////////////////////
/// SQLの実行(削除)
  DBRResultSet *pRs1(pDB1);       // オブジェクトの生成

   // 更新目的で検索
   pRs1->Execute("SELECT F1, F2 FROM TABLE1", TYPE_EXCLUSIVE);

   pRs1->Open();                  // レコードの検索
   while(!pRs1->IsEOF())
   {
      pRs1->Delete();             // 削除のためのDeleteメソッドの呼び出し
      pRs1->Next();       // Nextメソッドにより次のレコードの読み込み
   }
   pDB1->Commit();                // トランザクションのコミット
    .....
```

注意事項

接続する DBMS が SQL Server の場合，Close メソッドで ResultSet を削除してから Commit メソッドを呼出してください。Close メソッドを呼ばないと排他ロックが解除されません。

(2)　DELETE 文を利用した削除

DELETE 文を使ってレコードを削除できます。これは，既に削除するレコードが特定できる場合に，DELETE 文で削除するレコードを指定する方法です。DELETE 文を利用してレコードを削除するには DBRDatabase クラスの ExecuteDirect メソッドを使用します。

```
////////////////////////////////////////////////////////
/// SQLの実行(削除)
     pDB1->ExecuteDirect("DELETE FROM TABLE1 WHERE F1=100");
     pDB1->Commit();
```

2.5.5　レコードの追加

レコードを追加するためには,ExecuteDirect メソッドを利用します。追加するレコードの内容を指定した INSERT 文を ExecuteDirect メソッドの引数に設定することにより，追加できます。

INSERT 文には?パラメタが指定できません。

レコードの追加が完了した後で Commit メソッドを発行してデータベースの更新情報を確定させる必要があます。

```
////////////////////////////////////////////////////////
/// SQLの実行(追加)
     pDB1->ExecuteDirct("INSERT INTO TABLE1(F1,F2)
                                          VALUES(1,100)");
     pDB1->Commit();
```

2.5.6　DBRResultSet 仮想関数の利用

DBRResultSet クラスでは, オーバライド可能な仮想関数を提供しています。この仮想関数を利用するためには，派生クラスを作成する必要があります。

ここでは，この関数仕様とその利用方法について説明します。

(1)　OnBeforeRefresh メソッドの仕様

■機能

Open，Refresh メソッドで ResultSet に検索結果を読込む前に呼び出される。

■形式

```
virtual void OnBeforeRefresh(BOOLEAN bRefesh);
```

■引数

bRefesh … 呼び出し元が Refresh メソッドの場合は TRUE，Open は FALSE

(2)　OnMoveRecord メソッドの仕様

■機能

Absolute，Bottom，Next，PageNext，Previous，Refresh，Relative，Top メソッドでカーソルの移動時に呼び出される。また，Update メソッドで更新直前に呼び出される。

■形式

```
virtual void OnMoveRecord(BOOLEAN bSaveAndValidate);
```

■引数

bSaveAndValidate … 呼び出し元が Update メソッドの場合は TRUE，他は FALSE

(3)　OnEndRecord メソッド

■機能

カレントレコードを移動した結果，IsEOF が TRUE になった時に呼び出される。

■形式

```
virtual void OnEndRecord(BOOLEAN bEndRecord);
```

■引数

bEndRecord … TRUE 固定

(4)　仮想関数の利用例

DBRResultSet クラスで提供する仮想関数を利用する例を説明します。この仮想関数を利用するためには，派生クラスを作成する必要があります。

(a)　派生クラスの外部仕様

図 2-3 に示すテーブルをアクセスする Custom クラスとし, OnMoveRecord のカーソル移動では, フィールド値をメンバ変数に格納し，Update の時には，フィールド値をチェックするものとします。

図 2-3　業務で使用するテーブルの例

テーブル名：custom

USERID ※1	TEL ※2	ZIP ※3	※4

このテーブルでは顧客情報を管理します。

注※1 得意先コードを格納するフィールド名
注※2 電話番号を格納するフィールド名
注※3 郵便番号を格納するフィールド名
注※4 業務に関係しないフィールド

(b)　派生クラスの作成

■派生クラスの仕様

- Custom コンストラクタ

  機能

  DBRResultSet クラスの機能を継承した Custom オブジェクトを生成します。テーブル custom に対応した変数 tokuisakiCD，TELNO，ZIPCD を持ちます。

  形式

  ```
  Custom(DBRDatabase *pDatabase)
  ```

  引数

  pDatabase：DBRDatabase オブジェクトのポインタを指定します。

- Next などのカーソル操作メソッド

  機能

  カーソルを移動すると，テーブル Custom に対応した変数 tokuisakiCD，TELNO，ZIPCD にフィールド値を格納します。

- Update メソッド

  機能

  変数 tokuisakiCD，TELNO，ZIPCD にフィールド値を設定後に呼び出すとデータのチェックを行った後，Update 処理を行う。

■派生クラスの定義

データの型を LPTSTR としてメンバ変数 tokuisakiCD，TELNO，ZIPCD を定義します。これらのメンバ変数はアプリケーションに参照させるため public 属性にします。

OnMoveRecord メソッドのプロトタイプだけを定義します。

```
//////////////////////////////////////////////////////////
class Custom : public DBRResultSet
{
Public:
// Customのコンストラクタ
  Custom(DBRDatabase* pDatabase) ;
  // OnMoveRecord関数の定義
  virtual void OnMoveRecord(BOOLEAN bSaveAndValidate);
  // メンバ変数の定義
  LPTSTR tokuisakiCD;
  LPTSTR TELNO;
  LPTSTR ZIPCD;
};
```

■コンストラクタの定義

コンストラクタでは，フィールド値を格納するメンバ変数を初期化します。

```
////////////////////////////////////////////////////////
Custom::Custom(DBRDatabase* pDatabase) :
                                        DBRResultSet(pDatabase)
{
  tokuisakiCD = NULL;
  TELNO = NULL;
  ZIPCD = NULL;
}
```

■OnMoveRecord 仮想関数の再定義

Update メソッドの場合，メンバ変数 ZIPCD のデータの妥当性をチェックし，不当なデータの場合は０００を設定します。

他のカーソル移動メソッドの場合，カレントレコードの各フィールド値をそれぞれ tokuisakiCD,TELNO,ZIPCD メンバ変数に設定します。

```
////////////////////////////////////////////////////////
void Custom::OnMoveRecord(BOOLEAN bSaveAndValidate)
{
   if (bSaveAndValidate){        // レコードの更新要求か
     // 更新データのチェック
     if(ZIPCD[0]!='0') {         // 先頭文字は０か
        ZIPCD="000";             // ０００を設定
     }
     SetField(1, tokuisakiCD);
     SetField(2, TELNO);
     SetField(3, ZIPCD);
   } else {                      // いいえ
     GetField(1, &tokuisakiCD);
                      // USERIDフィールドをtokuisakiCD変数に設定
     GetField(2, &TELNO);        // TELフィールドをTELNO変数に設定
     GetField(3, &ZIPCD);        // ZIPフィールドをZIPCD変数に設定
   }
};
```

2.6　詳細版クラスのデータベースアクセス

ここでは，詳細版クラスを利用したデータベースアクセスの方法について説明します。

2.6.1　レコードの検索

(1)　DBStatement クラスを利用した検索

レコードの検索には SQL 文の SELECT 文を使います。DBStatement クラスを使用する場合， SELECT 文中に?パラメタを指定できません。

アプリケーションでは， DBConnection オブジェクトの CreateStatement メソッドを呼び出して，DBStatement オブジェクトを生成し，Execute メソッドの引数として SELECT 文を指定します。

ロック方法は，SetResultSetType メソッドで設定し，一度に ResultSet に読み込むレコードの最大数は，SetMaxRows メソッドを使って設定します。

レコードの検索は，DBStatement オブジェクトの GetResultSet メソッドを呼び出して，検索結果を ResultSet に得ます。

```
////////////////////////////////////////////////////////////
/// SQLの実行（検索）　?パラメタを使用しない

DBStatement*                pStatement;
DBResultSet*                pResultSet;

                        // DBStatementオブジェクトの生成
pStatement = pConnect->CreateStatement();
                        // 排他オプションの設定
pStatement->SetResultSetType(TYPE_NONE);
                        // SELECT文の設定
pStatement->Execute("SELECT F1, F2 FROM TABLE1");
                        // レコードの検索
pResultSet = pStatement->GetResultSet();
.....
```

(2)　DBPreparedStatement クラスを利用した検索

SQL 文の SELECT 文中に?パラメタを指定した検索には，DBPreparedStatement クラスを使用します。

アプリケーションでは， DBConnection クラスの CreatePreparedStatement メソッドの引数として SQL 文を指定して，DBPreparedStatement オブジェクトを生成し，Execute メソッドでデータベースへ通知（SQL 文の解析）します。その後，?パラメタへの値を， SetParam メソッドで設定します。

ロック方法は，SetResultSetType メソッドで設定し，一度に ResultSet に読み込むレコード数の最大値は，SetMaxRows メソッドを使って設定します。

レコードの検索は，GetResultSet メソッドを呼び出して，検索結果を ResultSet に得ます。

```
////////////////////////////////////////////////////////////
/// SQLの実行（検索）　?パラメタを使用

DBPreparedStatement*        pStatement;
DBResultSet*                pResultSet;
                    :
                    :
// DBPreparedStatementオブジェクトを生成。
// このとき，SQL文に?パラメタを設定
pStatement=pConnect->CreatePreparedStatement
("SELECT * FROM TABLE1 ID > ? AND ID < ? ");
 pStatement->Execute();                          // SQL文の実行
 pStatement->SetParam(1,100);
```

```
                          // SetParamで?パラメタに対する値の設定
pStatement->SetParam(2,300);
pResultSet = pStatement->GetResultSet();        // レコードの検索
```

(3)　検索結果の確認

レコードの検索結果は，DBResultSet オブジェクトの IsEOF メソッドを使って確認します。判定が FALSE であれば，ResultSet にデータが読み込まれています。

又は，DBResultSet オブジェクトの GetRowCount メソッドでレコード数を調べられます。

（例 1）IsEOF を使った場合

```
pResultSet = pStatement->GetResultSet();        // レコードの検索
if (pResultSet->IsEOF())                        // 検索結果の確認
```

（例 2） GetRowCount を使った場合

```
pResultSet = pStatement->GetResultSet();        // レコードの検索
RCount=pResultSet->GetRowCount();               // 検索結果の確認
```

2.6.2　検索レコードの参照

ここでは，ResultSet に読込まれたレコードを参照する方法について説明します。

(1)　カーソルの移動

ResultSet に複数の検索結果を得た場合，一つ一つのレコードを特定するために，カーソルを使用します。

ResultSet の中でカーソルを移動させるには，次のメソッドを呼び出します。

- カーソルを次のレコードへ移動（Next メソッド）
- カーソルを前のレコードへ移動（Pervious メソッド）
- カーソルを n 番目のレコードへ移動（Absolute メソッド）
- カーソルを現在のレコードから n 個先のレコードへ移動（Relative メソッド）
- カーソルを最後のレコードへ移動（Bottom メソッド）
- カーソルを先頭のレコードへ移動（Top メソッド）

なお，1 度に ResultSet へ格納できるレコード数は SetMaxRows メソッドで指定でき，これより多くのレコードを検索した場合は，複数回に分けてレコードを取得することになります。SetMaxRows メソッドの値を超えたレコード数を参照するには，次に示す PageNext メソッドを利用します。

- カーソルを次の ResultSet の先頭レコードへ移動（PageNext メソッド）

この他に，現在のレコード位置を知るためのメソッドがあります。

- 先頭レコードから数えた現在のレコード位置：GetCurrent メソッド
- 現在読み込んでいる ResultSet の先頭レコードから数えた現在のレコード位置：GetCurrentOfResultSet メソッド

カーソルが，ResultSet のレコードの範囲を越えて指定された場合には，エラーがスローされます。アプリケーションでは，カーソルを移動する前後で IsEOF メソッドの戻り値を確認することで，ResultSet の範囲を確認できます。

(2)　フィールド値の参照・データ変換

ResultSet のレコードのフィールド値は，DBResultSet クラスの GetField メソッドを使って参照します。データベースのデータ型と GetField メソッドの引数に指定したデータ型とが異なる場合は，引数のデータ型に合わせて変換されます。

文字列から数値データ型への変換に失敗した場合には，0 が返されます。指定したフィールドがなかった場合や値が変換できない場合は，エラーをスローします。データ型の変換規則については，「7. データ型」を参照してください。

SetResultSetType メソッドで，TYPE_BLOB_FILE を指定した場合は，BLOB 型のフィールドに対する GetField メソッドの引数にファイル名称が戻されます。この場合，GetField メソッドの引数の型には，文字列と同様に，LPTSTR を指定してください。

```
////////////////////////////////////////////////////////
// 値の参照の例
LPTSTR          pField;
INT32           nField;
       //データベースのデータの検索処理
                    :
                    :
       // 検索結果をResultSetに取得
       // レコードの読み込み
   while(!pResultSet->IsEOF())
   {
       // フィールドのデータの文字列変換と「ポインタ」の受け取り
           pResultSet->GetField(1, &nField);
           cout << ”  Data=” << nField;
           pResultSet->GetField(2, &pField);
           cout << ”  Data=” << pField;
       cout << endl;
       // 次のレコードの読み込み
       pResultSet->Next();
   }
```

(3)　更新のための参照をする場合

レコードの更新を目的として，データベースのレコードを参照するときには，SetResultSetType メソッドで ResultSet の排他タイプを TYPE_EXCLUSIVE とします。これにより，GetResultSet で得る ResultSet が更新可能になります。

更新可能な ResultSet には，一度に一つのレコードしか読み込めません。次のレコードを読み込むには，Next メソッドを使います。

なお，更新可能な DBResultSet オブジェクトを生成している場合は，検索時に SELECT 文で FOR UPDATE オプションを指定する必要はありません。

2.6.3　レコードの更新

データを更新するには，前節で説明した ResultSet を利用する方法と，SQL の UPDATE 文を利用する方法とがあります。

(1)　ResultSet を利用した更新

ResultSet に検索されたレコードを更新するためには，カーソルをレコードに位置付け，Edit メソッドで更新準備を指示します。この後，SetField メソッドでフィールドの内容を変更し，最後に Update メソッドで更新を指示します。複数のレコードを更新する場合には，この処理を繰り返す必要があります。

GetField メソッドの引数に指定したデータ型とデータベースのデータ型とが異なる場合は，データベースのデータ型に合わせて変換されます。データ型の変換規則は「7. データ型」を参照してください。

```
////////////////////////////////////////////////////////
/// SQLの実行（更新）

DBStatement*                pStatement;
DBResultSet*                pResultSet;

      // DBStatementクラスの生成
pStatement = pConnect->CreateStatement();
      // 更新可能なDBResultSetクラスを生成するために
      // DBResultSetクラスのタイプを指定
pStatement->SetResultSetType(TYPE_EXCLUSIVE);
      // SELECT文の設定
pStatement->Execute("SELECT F1, F2 FROM TABLE1");
      // レコードの検索
    pResultSet = pStatement->GetResultSet();
      // レコードの読み込み
    while(!pResultSet->IsEOF())
   {
      // フィールド"F2"のデータを更新
      // ここでは，uFunc()というユーザ関数を使用
      // LPCTSTR uFunc(void);
      // 更新のためのEdit-SetField-Updateメソッド呼び出し
   pResultSet->Edit();
   pResultSet->SetField("F2", uFunc());
   pResultSet->Update();
      // Nextメソッドで次のレコードの読み込み
   pResultSet->Next();
   }
// その他の処理
.....
```

BLOB 型データの渡し方

SetResultSetType メソッドで，TYPE_BLOB_FILE を指定した場合は，BLOB 型のフィールドに対する GetField メソッドの引数にファイル名称が戻されます。この場合，GetField メソッドの引数の型には，文字列と同様に，LPTSTR を指定してください。

XDM/RD をご使用の場合の注意

XDM/RD をご使用の場合，LONG VARCHAR 型， LONG NVARCHAR 型，及び LONG MVARCHAR 型のフィールドのデータの更新はできません。SetField メソッドで値を設定しても，Update メソッドの実行時にエラーになります。

(2)　UPDATE 文を利用した更新

SQL 文の UPDATE 文を使って，データを更新できます。これは，既に更新するデータや内容が特定できる場合に，SQL 文で直接，更新個所及び値を指定する方法です。この方法では，現在のデータを確認してから更新することはできません。

次に，DBStatement クラス及び DBConnection クラスを使用して，?パラメタを使用しない更新をする場合と， DBPreparedStatement クラスを使用して，?パラメタを使用した更新をする場合の三通りについて説明します。

(a)　DBStatement クラスを使用した場合

DBConnection クラスの CreateStatement メソッドで DBStatement オブジェクトを生成します。次に DBStatement クラスの Execute メソッドに SQL 文を指定してメソッドを呼び出し，データを更新します。

```
////////////////////////////////////////////////////////
/// SQLの実行(更新) - DBStatementクラスでUPDATE文を指定

   // DBStatementオブジェクトの生成
```

```
pStatement = pConnect->CreateStatement();
   // SQL文の実行
pStatement->Execute("UPDATE TABLE1 SET F1=100 WHERE F2=300");
```

(b)　DBConnection クラスを使用した場合

DBConnection クラスの ExecuteDirect メソッドで SQL 文を実行してデータを追加します。

この方法は手順が簡単ですが，データをいったん検索してから更新したい場合は，DBStatement クラスを使用してください。DBConnection クラスでは検索結果を受け取ることができないため，検索ができません。

```
////////////////////////////////////////////////////////
/// SQLの実行(更新) - DBConnectionクラスでUPDATE文を指定

   // SQL文の実行
pConnect->ExecuteDirect("UPDATE TABLE1 SET F1=100
                                              WHERE F2=300");
```

(c)　DBPreparedStatement クラスを使用した場合

?パラメタを記述した UPDATE 文を指定して DBConnection クラスの CreatePreparedStatement メソッドを呼び出し，DBPreparedStatement オブジェクトを生成します。次に，Execute メソッドで SQL 文をデータベースへ通知（解析）し，SetParam メソッドで?パラメタのデータを設定します。そして，ExecuteUpdate メソッドで更新を実行します。

```
////////////////////////////////////////////////////////
/// SQLの実行(更新)
///- DBPreparedStatementクラスで?パラメタのあるUPDATE文を指定

   // DBPreparedStatementオブジェクトの生成
pStatement = pConnect->CreatePreparedStatement(
                        "UPDATE TABLE1 SET F1=? WHERE F2=?");
   // SQL文の解析
pStatement -> Execute();
   // ?パラメタへの値の設定
pStatement->SetParam(1,100);
pStatement->SetParam(2,300);
   // 更新の実行
pStatement->ExecuteUpdate();
```

(3)　更新可能な ResultSet の取得時の注意

SetResultSetType メソッドで引数に TYPE_EXCLUSIVE を指定した場合，GetResultSet メソッドで更新できる ResultSet が取得できます。ただし，次の場合には，SetResultSetType メソッドの引数に TYPE_EXCLUSIVE を指定すると Execute メソッド実行時にエラーをスローします。

- 集合関数(avg, count など)や組み込み関数を使用する場合
- DISTINCT を使用する場合
- GROUP BY 句や UNION 句を使用する場合
- 演算子を含む場合(UNION, INTERSECT, MINUS)
- JOIN を含む場合
- 上記に該当する VIEW を利用する場合

また，XDM/RD を使用している場合，検索結果に long 列(long mvarchar 型，long nvarchar 型, long varchar 型)を含んでいると，そのフィールドに対しては，Update メソッドを実行できません。

2.6.4　レコードの削除

データを削除するには，ResultSet を利用する方法と，SQL の DELETE 文を利用する方法とがあります。

(1)　ResultSet を利用した削除

ResultSet に検索されたレコードを削除するためには，カーソルをレコードに位置づけ，Edit メソッドで削除準備を指示し，Delete メソッドで削除を指示します。複数のレコードを削除する場合には，この処理を繰り返す必要があります。

```
////////////////////////////////////////////////////////
/// SQLの実行(削除)

DBStatement*                pStatement;
DBResultSet*                pResultSet;

      // SQL文実行のため，DBStatementクラスの生成
   pStatement = pConnect->CreateStatement();
      // 更新可能なDBResultSetオブジェクトを生成するため，
      // DBStatementオブジェクトでTYPE_EXCLUSIVEを設定
      // このオプションはデフォルト値なので省略可
   pStatement->SetResultSetType(TYPE_EXCLUSIVE);
      // SELECT文を指定し，SQL文の解析
      // ここでは，SELECT文に"FOR UPDATE"不要
      // このオプションは自動的に付加される。
   pStatement->Execute("SELECT F1, F2 FROM TABLE1");
      // DBResultSetを生成し，検索結果を取得
      // 削除できるレコードは１レコードだけ
   pResultSet = pStatement->GetResultSet();
   while(!pResultSet->IsEOF())          // すべてのレコードの読み込み
   {
      // 削除のためのDeleteメソッドの呼び出し
      pResultSet->Delete();
      pResultSet->Next();
                        // Nextメソッドにより次のレコードの読み込み
   }
      // その他の処理
.....
```

(2)　DELETE 文を利用した削除

SQL 文の DELETE 文を使って，データを削除できます。次に，DBStatement クラス及び DBConnection クラスを使用して，?パラメタを使用しない更新をする場合と，DBPreparedStatement クラスを使用して，?パラメタを使用した更新をする場合の三通りについて説明します。

(a)　DBStatement クラスを使用した場合

DBConnection クラスの CreateStatement メソッドで DBStatement オブジェクトを生成します。生成された DBStatement クラスの Execute メソッドで SQL 文を実行してデータを削除します。

```
////////////////////////////////////////////////////////
/// SQLの実行(削除) - DBStatementクラスでDELETE文を指定

   // DBStatementオブジェクトの生成
pStatement = pConnect->CreateStatement();
   // SQL文の実行
pStatement->Execute("DELETE FROM TABLE1 WHERE F1=100");
```

(b)　DBConnection クラスを使用した場合

DBConnection クラスの ExecuteDirect メソッドで SQL 文を実行してデータを削除します。

この方法は手順が簡単ですが，データをいったん検索してから削除したい場合は，DBStatement クラスを使用してください。DBConnection クラスでは検索結果を受け取ることができないため，検索ができません。

```
////////////////////////////////////////////////////////
/// SQLの実行(削除) - DBConnectionクラスでDELETE文を指定

   // SQL文の実行
pConnect->ExecuteDirect("DELETE FROM TABLE1 WHERE F1=100");
```

(c)　DBPreparedStatement クラスを使用した場合

?パラメタを記述した DELETE 文を指定して,DBConnection クラスの CreatePreparedStatement メソッドを呼び出し，DBPreparedStatement オブジェクトを生成します。次に，Execute メソッドで SQL 文をデータベースに通知（解析）し，SetParam メソッドで?パラメタのデータを設定します。そして，ExecuteUpdate メソッドで削除を実行します。

```
////////////////////////////////////////////////////////
/// SQLの実行(削除)
/// - DBPreparedStatementクラスで?パラメタのあるDELETE文を指定

   // DBPreparedStatementオブジェクトの生成
pStatement = pConnect->CreatePreparedStatement(
                                "DELETE FROM TABLE1 WHERE F1 < ? AND F2 < ?");
   // SQL文の解析
pStatement->Execute();
   // ?パラメタへの値の設定
pStatement->SetParam(1,100);
pStatement->SetParam(2,300);
   // 削除の実行
pStatement->ExecuteUpdate();
```

2.6.5　レコードの追加

データを追加するには，SQL の INSERT 文を利用します。

次に，DBStatement クラス及び DBConnection クラスを使用して，?パラメタを使用しない更新をする場合と，DBPreparedStatement クラスを使用して，?パラメタを使用した更新をする場合の三通りについて説明します。

(1)　DBStatement クラスを使用した場合

DBConnection クラスの CreateStatement メソッドで DBStatement オブジェクトを生成します。生成された DBStatement クラスの Execute メソッドで SQL 文を実行してデータを追加します。

```
////////////////////////////////////////////////////////
/// SQLの実行(追加) - DBStatementクラスでINSERT文を指定

   // DBStatementオブジェクトの生成
pStatement = pConnect->CreateStatement();
   // SQL文の実行
pStatement->Execute("INSERT INTO TABLE1(F1,F2)VALUES(1,100)");
```

(2)　DBConnection クラスを使用した場合

DBConnection クラスの ExecuteDirect メソッドで SQL 文を実行してデータを追加します。

この方法は手順が簡単ですが，データをいったん検索してから追加したい場合は，DBStatement クラスを使用してください。DBConnection クラスでは検索結果を受け取ることができないため，検索ができません。

```
////////////////////////////////////////////////////////
/// SQLの実行(追加) - DBConnectionクラスでINSERT文を指定

   // SQL文の実行
pConnect->ExecuteDirect("INSERT INTO TABLE1(F1,F2)
                                          VALUES(1,100)");
```

(3)　DBPreparedStatement クラスを使用した場合

?パラメタを指定した INSERT 文を指定し,DBConnection クラスの CreatePreparedStatement メソッドを呼び出し，DBPreparedStatement オブジェクトを生成します。次に， Execute メソッドで SQL 文をデータベースに通知（解析）し，SetParam メソッドで?パラメタのデータを設定します。そして，ExecuteUpdate メソッドで追加を実行します。

なお，レコードは SetInsertRows メソッドで設定する数（一度に追加するレコード数）に従って，追加されます。一度に複数レコードを追加する場合，SetParam メソッドの引数には，レコード番号とフィールド番号及びデータ値の，計 3 個を指定する必要があります。

```
////////////////////////////////////////////////////////
/// SQLの実行(追加)
/// - DBPreparedStatementクラスで?パラメタのあるINSERT文を指定

   // DBPreparedStatementオブジェクトの生成
pStatement = pConnect->CreatePreparedStatement
("INSERT INTO TABLE1(F1,F2)VALUES(?,?)");
   // SQL文の解析
pStatement->Execute();
   // ?パラメタへの値の設定
pStatement->SetParam(1,1);
pStatement->SetParam(2,100);
   // 追加の実行
pStatement->ExecuteUpdate();
```

2.6.6　ストアドプロシジャの利用

ストアドプロシジャを利用するには，データベースの接続時に生成した DBConnection オブジェクトの CreateCallableStatement メソッドを呼び出し，DBCallableStatement オブジェクトを生成します。次に，SetProcedure メソッドで実行するストアドプロシジャ名を指定し，引数を SetParam メソッドで設定し，Execute メソッドで実行します。

戻り値のあるストアドプロシジャの実行結果は， GetParam メソッドの引数で受け取る方法と，ResultSet で受け取る方法があります。また，これらを併用することもできます。

ストアドプロシジャは，サーバ系の下記 DBMS で利用可能でであり，ストアドプロシジャを利用するには，DBCallableStatement クラスを使います。

- HiRDB
- ORACLE
- Sybase SQL Anywhere，Adaptive Server Anywhere
- SQL Server

(1)　ストアドプロシジャの実行

ストアドプロシジャを利用するには，データベースの接続時に生成した DBConnection オブジェクトの CreateCallableStatement メソッドを呼び出し，DBCallableStatement オブジェクトを生成します。次に，SetProcedure メソッドで実行するストアドプロシジャ名を指定し，引数を SetParam メソッドで設定し，Execute メソッドで実行します。

```
DBConnection*         pConnect;
  DBCallableStatement*  pCall;
  {
    // DBCallableStatementオブジェクトの生成
    pCall = pConnect->CreateCallableStatement("sample1");
    pCall->SetProcedure("Proc100");              // プロシジャ名設定
    pCall->Execute();                            // プロシジャ実行
  }
```

(a)　引数を持つストアドプロシジャの実行

ストアドプロシジャが引数を持つ場合，SetProcedure メソッドでは，プロシジャ名に続けて"?"を引数分指定します。?パラメタに設定する値は，DBCallableStatement オブジェクトの SetParam メソッドで指定します。SetParam メソッドでは，?パラメタの位置と設定するデータを指定します。

SetParam メソッドで指定する?パラメタの位置は，IN，INOUT パラメタの位置です。OUT は含みません。

```
  DBConnection*         pConnect;
  DBCallableStatement* pCall;
 {
    // DBCallableStatementオブジェクト生成
    pCall = pConnect->CreateCallableStatement("sample1");
    pCall->SetProcedure("Proc_100(?, ?)");
                                      // ストアドプロシジャ名設定
    pCall->SetParam(1,  pszCity) ;    // パラメタ値の設定
    pCall->Execute();                 // ストアドプロシジャ実行
 }
```

(b)　同期・非同期処理

ストアドプロシジャも通常のデータベースアクセスと同様，DBConnection オブジェクトの設定により同期・非同期処理が行われます。

```
  DBConnection*         pConnect;
  DBCallableStatement* pCall;
  {
    // DBCallableStatementオブジェクトの生成
    pCall = pConnect->CreateCallableStatement("sample1");
    pCall->SetProcedure("Proc_100");         // プロシジャ名設定
    pCall->Execute();                        // プロシジャ実行
    pCall->WaitForDataSource(DBR_INFINITE);
                                    // プロシジャ実行完了待ち
}
```

また，ストアドプロシジャの実行完了確認には，DBCallableStatement オブジェクトの IsCompleted メソッドも使えます。

```
    if (!pCall->IsCompleted())            // プロシジャ実行の完了確認
    pResultSet = pCall ->GetResultSet();  // 検索結果の受け取り
```

(2)　ストアドプロシジャの戻り値の受け取り(引数)

この項では，戻り値のあるストアドプロシジャの実行結果を，引数で受け取る方法について説明します。

実行結果は DBCallableStatement オブジェクトの GetParam メソッドの引数で受け取ります。データは Getparam メソッドの引数で指定したデータ型で取得します。

```
void class1::func1(char* pszCity,INT32 dwCount)
 {
  DBConnection*          pConnect;
  DBCallableStatement*   pCall;

           // DBCallableStatementオブジェクトの生成
    pCall = pConnect->CreateCallableStatement("sample1");
    pCall->SetProcedure("Proc_100(?, ?)");
                                    // ストアドプロシジャ名設定
```

```
    pCall->SetParam(1,  pszCity) ;    // パラメタ値の設定
    pCall->Execute();                 // ストアドプロシジャ実行
    pCall->WaitForDataSource(DBR_INFINITE);
                                      // プロシジャ実行完了待ち
    pCall->GetParam(1,  &dwCount);    // 戻り値の受け取り
}
```

なお，IN パラメタと OUT パラメタの位置はどちらも 1 になります。

(3)　ストアドプロシジャの戻り値の受け取り(ResultSet)

下記 DBMS の場合だけ，ストアドプロシジャでの SELECT 文の実行結果が複数のレコードでもアプリケーションで受け取ることができます。

- SQL Anywhere
- Adaptive Server Anywhere

この場合，ストアドプロシジャでは RESULT 句で戻り値の型を定義し，アプリケーションでは，DBResultSet クラスの ResultSet として受取り，GetField メソッドで参照します。

(a)　ストアドプロシジャの形式

ストアドプロシジャでは，SELECT 文の結果を呼び出し元に渡すために，RESULT 句で戻り値の型を定義します。

```
CREATE PROCEDURE sample ()
RESULT ("Value" INT,"Shop" CHAR(30))
BEGIN
  SELECT  CAST( sum( sales_order_items.quantity *
                product.unit_price)
                AS INTEGER ) AS value
                shop_name,
  FROM customer
           INNER JOIN sales_order
           INNER JOIN sales_order_items
           INNER JOIN product
  GROUP BY shop_name
  ORDER BY value desc;
END
```

この例では，value 及び shop_name の検索結果を ResultSet として扱うために，RESULT 句で Value 及び Shop フィールドとそのデータ型を新しく定義しています。

また，SELECT 文中に演算処理がある場合は，その結果を考慮してストアドプロシジャの作成時に，結果に合ったデータ型を RESULT 句で用意しておきます。

なお，RESULT 句に指定するデータ型は，元テーブルと同じデータ型，又は自動的に変換できるデータ型だけが指定できます。自動変換できるデータ型については,使用する DBMS のドキュメントを参照してください。

(b)　ResultSet の参照

ストアドプロシジャの検索結果を ResultSet で受け取る場合，DBCallableStatement オブジェクトから GetResultSet メソッドを呼び出して，DBResultSet オブジェクトを生成し，検索結果を ResultSet に得ます。ResultSet に読込まれたレコードを参照する方法については，「2.5.2 検索レコードの参照」と同様です。

データの型

ストアドプロシジャ内の RESULT 句で定義したデータ型と GetField メソッドの引数に指定したデータ型とが異なる場合は，引数のデータ型に合わせて変換されます。データ変換については，「7.1 クラス

ライブラリで扱うデータ型と変換規則」を，変換時の注意事項については「2.5.2 検索レコードの参照」を参照してください。

データ変換

ストアドプロシジャ内の SELECT 文の場合も単純な SELECT 文の場合も，検索結果を DBResultSet クラスの GetField メソッドで参照します。異なる点は，単純な SELECT 文の場合データベースでのフィールドの型を GetField メソッドの引数に指定したデータ型に変換して参照しますが，ストアドプロシジャの場合 RESULT 句で定義したフィールドの型を GetField メソッドの引数に指定したデータ型に変換して参照します。

その他

ストアドプロシジャを利用した検索でのレコードのロック方法は，SetResultSetType メソッドの引数で設定します。

また，検索結果が複数レコードの場合，n 件ずつ分割して取得することもできます。このとき一度に取得するレコードの最大数は，SetMaxRows メソッドで指定します。

(c)　複数 ResultSet の参照

ストアドプロシジャ中に，SELECT 文が複数記述されているストアドプロシジャでは，SELECT 文ごとに ResultSet を扱います。

次に，複数の SELECT 文を記述したストアドプロシジャの例を示します。

```
CREATE PROCEDURE ListPeople()
RESULT ( lname CHAR(36), fname CHAR(36) )
BEGIN
    SELECT emp_lname, emp_fname FROM employee;
    SELECT lname, fname FROM customer;
    SELECT last_name, first_name FROM contact;
END
```

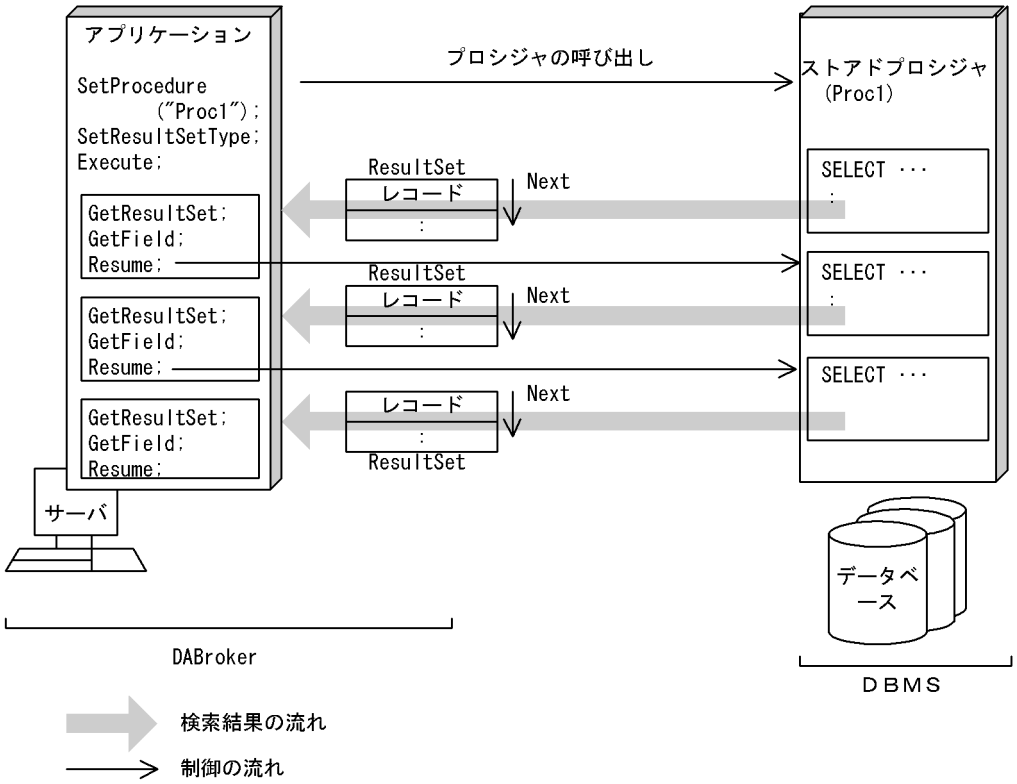
SELECT 文ごとに ResultSet を返すストアドプロシジャも，RESULT 句を指定します。ただし，複数の SELECT 文があっても，プロシジャ中に RESULT 句は一つだけしか指定できません。このため，指定する SELECT 文には次のような制限があります。

- SELECT 文に指定するフィールド数と，RESULT 句で指定するフィールド数は同じにする。
- SELECT 文に指定するフィールドの型は，RESULT 句で指定する型と同じ,又は自動的に変換できるものにする。

アプリケーションでは，これらの制限によって，RESULT 句だけを意識すればよくなります。つまり，GetField メソッドを使ってアプリケーションから参照する時は，RESULT 句で定義したフィールドのデータ型だけを意識します。

プロシジャ内の次の SELECT 文に対する ResultSet を取得する概念図を図 2-4 に示します。

図 2-4　プロシジャ内の次の SELECT 文に対する ResultSet を取得する概念図



検索結果の有無確認

複数の SELECT 文がある場合，ひとつの ResultSet の参照が終わっても，次の ResultSet の処理が必要となります。このような場合，IsEOF メソッドと IsCompleted メソッドを利用して判断します。

下記例は，ResultSet の有無を IsCompleted メソッドで確認し，ResultSet 中のレコードの確認を IsEOF メソッドで確認しています。

```
pCall->Execute();                    // ストアドプロシジャ実行
if (!pCall->IsCompleted())           // プロシジャ実行の完了確認
{
    pResultSet = pCall ->GetResultSet(); // 検索結果の受け取り
    while(!pResultSet->IsEOF())
    {
        pResultSet->GetField(1, &nField);      // データの取得
        cout << "  Data=" << nField;
        pResultSet->GetField(2, &pField);
        cout << "  Data=" << pField;
        cout << endl;
        pResultSet->Next();          // 次のレコードの読み込み
    }
    pCall->Resume();                 // 処理の継続
}
```

GetResultSet メソッドを実行すると，前に作成した DBResultSet オブジェクトは削除されることに注意してください。

カーソルの移動

複数の SELECT 文がある場合，ひとつの ResultSet に対する処理が終了した後で，Resume メソッドを使って，カーソルを以降の ResultSet の先頭に位置付ける必要があります。

2.7　繰り返し列へのアクセス

この節では，繰り返し列へのアクセス方法について説明します。繰り返し列へのアクセスは，簡易版クラスと詳細版クラスで共通です。

この節で使用している例は，HiRDB を使用した場合のものです。

- 検索条件としての要素の利用
  繰り返し列の要素を，レコードを検索したり，削除したりするときの検索条件として利用できます。
- ResultSet に検索した要素の参照
  ResultSet に読み込んだ繰り返し列の要素を，DBRArrayData オブジェクトに取込み，GetData メソッドで参照する方法について説明します。
- ResultSet を利用した要素の更新
  要素を更新する手段として ResultSet に読み込んだ繰り返し列の要素を更新する方法と新たに要素を設定する（一括更新）について説明します。
- パラメタを利用した要素の一括更新
  SQL 文の繰り返し列を?パラメタとし，繰り返し列の要素を設定する方法について説明します。
- SQL 文を利用した要素の更新
  SQL 文で直接要素を更新する方法について説明します。

2.7.1　検索条件としての要素の利用

次に示す SQL の ARRAY 句を利用すれば，レコードを検索したり更新するときの検索条件に繰り返し列の要素が指定できます。検索条件の要素に?パラメタを利用することもできます。

ARRAY(繰り返し列名〔，繰り返し列名〕)…[ANY](検索条件)

検索条件とする場合，ARRAY 句を用い繰り返し列であることを宣言します。

(1)　繰り返し列の要素を使った条件設定

検索条件として繰り返し列を利用する場合も，基本的な設定方法は通常のレコードを扱う場合と同じです。通常のレコードでの条件設定と異なるのは，SQL 文の条件として繰り返し列を設定するときに ARRAY 句で指定する点です。基本的には，SQL 文の WHERE 句に要素の値を条件として設定し，レコードを特定します。

(例)繰り返し列 F1 の値として 1 を持ち，F2 に 100 以上の値を持つレコードを検索する。

```
////////////////////////////////////////////////////////////
/// SQLの実行(検索) - DBStatementクラスでSELECT文を指定

pStatement = pConnect->CreateStatement();
pStatement->SetResultSetType(TYPE_NONE);    // 排他オプションの設定
pStatement->Execute(                        // SELECT文の設定
          "SELECT * FROM TABLE1 WHERE ARRAY(F1,F2) [ANY]
                                                (F1=1 AND F2>=100)");
```

(2)　繰り返し列の要素に?パラメタを使った条件設定

繰り返し列の要素に?パラメタを指定した場合，?パラメタの値は SetParam メソッドで設定します。

(例) 繰り返し列 F1 に 1 から 100 の値を持つレコードを削除する。

```
////////////////////////////////////////////////////////
/// SQLの実行(削除)
/// - DBPreparedStatementクラスで?パラメタのあるDELETE文を指定

   // DBPreparedStatementオブジェクトの生成
pStatement = pConnect->CreatePreparedStatement(
             "DELETE FROM TABLE1 WHERE ARRAY (F1)[ANY]
                                       (F1 > ? AND F1 < ? )");

pStatement->Execute();              // SQL文の解析
pStatement->SetParam(1,1);          // 1番目の?パラメタへ値を設定
pStatement->SetParam(2,100);        // 2番目の?パラメタへ値を設定
pStatement->ExecuteUpdate();        // 実行
```

2.7.2 繰り返し列の参照

ここでは， ResultSet に取得した繰り返し列の要素を参照する方法について説明します。

ResultSet に取得した繰り返し列の要素をアプリケーションから直接アクセスできません。そのため，GetField メソッドの引数に DBRArrayDataConstPtr オブジェクトへのポインタを指定し DBArrayData オブジェクトに ResultSet の要素を取込みます。

要素の値は，DBArrayData オブジェクトの GetData メソッドで参照します。すべての要素を参照するには，GetData メソッドを要素数分繰り返し指定する必要があります。要素数は，DBRArrayData オブジェクトの GetArrayCount メソッドで調べられます。

また，要素の値は，GetData メソッドの引数で指定したデータ型に変換できます。

```
////////////////////////////////////////////////////////////////////
// ResultSetを利用した要素の参照例
INT32 i32Loop;
LPTSTR lptData;
//データベースのデータの検索処理
                :
// 検索結果をResultSetに取得
// 参照時はConst付きポインタを使用
DBRArrayDataConstPtr cpArray;               // DBRArrayDataConstPtrの宣言
pResultSet->GetField(1,&cpArray);           // データの取得

// 要素数分ループして参照
for(i32Loop = 1;i32Loop < cpArray->GetArrayCount()+1;i32Loop++)
{
if(cpArray->GetData(i32Loop, &lptData))
  cout << "ArrayData(" << i32Loop << ")=" << lptData << endl;
else
  cout << "ArrayData(" << i32Loop << ")=欠損値" << endl;
}
cout << endl;
```

2.7.3 ResultSet を利用した要素の更新

ResultSet を利用した繰り返し列の要素を更新する方法について説明します。

- 繰り返し列の要素の更新

  ResultSet に読み込んだ繰り返し列のデータを更新する方法です。

- 繰り返し列の要素の一括更新

  ResultSet から繰り返し列のデータを読み込まずに，繰り返し列のデータを設定する方法です。

(1) 繰り返し列の要素の更新

簡易版クラスを使用している場合，DBRDatabase オブジェクトの GetArrayDataFactory メソッドで DBRDatabase オブジェクトごとに DBRArrayDataFactory オブジェクトを生成します。詳細版クラス

を使用している場合は，DBConnection オブジェクトの GetArrayDataFactory メソッドで DBConnection オブジェクトごとに生成します。

1. レコードを ResultSet に探索し，要素を GetField メソッドで参照用 DBRArrayData オブジェクトに取込みます。
2. 参照用の DBRArrayData オブジェクトから CreateArrayData メソッドで更新用 DBRArrayData オブジェクトを生成します。
3. DBRArrayData オブジェクトの要素を SetData メソッドで更新します。要素に NULL を設定する場合は，SetNull メソッドを呼び出します。要素に NULL を設定しても，その要素は UPDATE 文の DELETE 句で要素を削除した場合のように削除されません。
4. 繰り返し列は，SetField メソッドで ResultSet に反映します。

（例）社員表テーブルの資格フィールドを検索して，要素の値が Database であれば DSP に変更する例です。

```
////////////////////////////////////////////////////////
/// SQLの実行(要素の更新)- 検索したレコードを更新
DBRDatabase *pDB1;
DBRResultSet *pResultSet;
DBRArrayDataFactory *pFactory = NULL;
DBRArrayDataPtr pArray2;
DBRArrayDataConstPtr pArray1;
int   nArray;
LPTSTR youso;
    :
    :
// DBRArrayDataFactoryオブジェクトの生成
  pFactory = pDB1->GetArrayDataFactory();
  pResultSet = new DBRResultSet(pDB1);
                                  // ResultSetオブジェクトの生成
// レコードの検索
  pResultSet-> Execute("SELECT 資格 from 社員表",TYPE_EXCLUSIVE);
  pResultSet->Open();                      // ResultSetへの読込み
  if(pResultSet->GetArraySize(1) > 0)      // 繰り返し列かどうかの判定
  {
    while(!pResultSet->IsEOF())            // 検索結果の終わりの判定
    {
// 繰り返し列の要素を参照用DBRArrayDataオブジェクトに読み出す
      pResultSet->GetField(1,&pArray1);
// 参照用のDBRArrayDataオブジェクトをコピーして
// 更新用のDBRArrayDataオブジェクトを生成
      pArray2=pFactory->CreateArrayData(pArray1);
// 繰り返し列の要素数分処理
      for (nArray=1;nArray<=pArray1->GetArrayCount();nArray++)
      {
// 要素を読み出し，Nullかどうかを判定
        if(pArray1->GetData(nArray,&youso))
         {                                 // Nullでない場合の処理
          if(strcomp(youso,"Database")==0)      // 要素の値を確認
            pArray2->SetData(nArray,"DSP");     // 要素の値をDSPに更新
         }
      }
      pResultSet->Edit();                   // レコードの更新準備
// 更新用DBRArrayオブジェクトの繰り返し列をResultSetに反映する
      pResultSet->SetField(1,pArray2);
      pResultSet->Update();                 // データベースを更新
// Updateメソッド終了後更新用のDBRArrayDataオブジェクトを削除
      delete pArray2;
      pResultSet->Next();                   // カレントレコードを更新
    }
  }
  else                                      // 繰り返し列でなければ
   cout << "Not Array Field";               // Not Array Fieldを表示
  pDB1->Commit();                       // データベースの更新をコミット
  :
```

(2)　繰り返し列の要素の一括更新

次の順序で繰り返し列に設定する要素の値を準備します。

DBRArrayDataFactory オブジェクトの CreatArrayData メソッドに繰り返し列の型を指定し DBRArrayData オブジェクトを生成します。次に DBRArrayData オブジェクトの Create メソッドで要素数分の領域を確保します。 DBRArrayData オブジェクトには SetData メソッドで要素の値を設定します。 DBRArrayData オブジェクトの値は SetField メソッドで ResultSet に反映します。

(例) 社員表テーブルの資格フィールドに要素 1syu,Database,Network を新規に追加する例

```
////////////////////////////////////////////////////////
DBRDatabase *pDB1;
DBRResultSet *pResultSet;
DBRArrayDataFactory *pFactory = NULL;
DBRArrayDataPtr pArray1;
    :
    :
// DBRArrayDataFactoryオブジェクトの生成
  pFactory = pDB1->GetArrayDataFactory();
// DBRArrayDataオブジェクトの生成
  pArray1 = pFactory->CreateArrayData(COL_TYPE_CHAR,10);
  pArray1->Create(3);                 // 3要素分の領域を確保
  pArray1->SetData(1,"1syu");         // 1番目の要素に1syuを設定
  pArray1->SetData(2,"Database");     // 2番目の要素にDatabaseを設定
  pArray1->SetData(3,"Network");      // 3番目の要素にNetworkを設定
       :
  pResultSet = new DBRResultSet(pDB1);
                                      // ResultSetオブジェクトの生成
// 氏名IDがZ004のレコードを検索
  pResultSet-> Execute("SELECT 資格 from 社員表
                              where 氏名ID='Z004'",TYPE_EXCLUSIVE);
  pResultSet->Open();                 // レコードをResultSetに読込む
  pResultSet->Edit();                 // レコードの更新準備
// DBRArrayオブジェクトの繰り返し列をResultSetに反映
  pResultSet->SetField(1,pArray1);
  pResultSet->Update();               // データベースを更新
  pDB1->Commit();                     // データベースの更新をコミット
  :
```

2.7.4　パラメタを利用した要素の一括更新

SQL の UPDATE 文又は INSERT 文に?パラメタを利用して要素を設定する方法です。更新の単位は，要素全体になります。この方法は，詳細版だけで利用できます。

SQL 文は，DBPreparedStatement クラスを使います。繰り返し列を含まないレコードの更新と異なるのは， DBRArrayData クラスを利用して要素の値を設定する点です。

1. DBRArrayDataFactory オブジェクトの CreateArrayData メソッドに繰り返し列の型を指定し DBRArrayData オブジェクトを生成します。
2. 要素数分の領域を DBRArrayData オブジェクトの Create メソッドに繰り返し列の型を指定して確保し，SetData メソッドで要素の値を設定します。
3. SetParam メソッドで?パラメタに繰り返し列のデータを設定します。
4. ExecuteUpdate メソッドで SQL 文を実行します。

なお，繰り返し列全体を?パラメタとした場合，「2 要素目だけ」という更新はできません。必ず全体を更新することになります。2 要素目だけというような更新をする場合は，UPDATE 文で ARRAY 句を利用してください。

(例)UPDATE 文の繰り返し列に?パラメタを使用し更新する。

```
////////////////////////////////////////////////////////
/// SQLの実行(レコードの更新)-
/// DBPreparedStatementクラスで?パラメタのあるUPDATE文を指定

   // DBPreparedStatementオブジェクトの生成
pStatement = pConnect->CreatePreparedStatement(
                  "UPDATE TABLE1 SET F1=? WHERE ID='CPP'");
   // SQL文の解析
pStatement -> Execute();
   // Factoryオブジェクトの生成
DBRArrayDataFactory * pFactory = pConnect->
                                      GetArrayDataFactory();

DBRArrayDataPtr pArray =pFactory->
                          CreateArrayData(COL_TYPE_CHAR,10);
pArray->Create(2);
pArray->SetData(1,"NET");
pArray->SetData(2,"DB");
pStatement->SetParam(1,pArray);
pStatement->ExecuteUpdate();
```

2.7.5　SQL 文を利用した要素の更新

SQL の UPDATE 文を利用して，繰り返し列の全体又は特定の要素を DBRArrayData クラスを使用しないで更新できます。更新には，繰り返し列に要素を追加する，既存の要素を更新する，既存の要素を削除するの三つがあります。更新するときは，操作対象の要素の位置と値を UPDATE 文に直接指定します。

また，INSERT 文では，繰り返し列を含むレコードを追加することもできます。

簡易版の場合 DBRDatabase クラス，詳細版の場合 DBStatement クラス又は DBConnection クラスを使って SQL 文を実行します。詳細版では，パラメタを使った SQL 文も指定できます。

(1)　要素の追加

UPDATE 文では，要素の追加に ADD 句を使うことにより，要素の最後に要素を追加するこでがでます。

(例 1)繰り返し列 F1 の要素の最後に 100 を追加(ADD 句)する。要素位置の最後は，*(アスタリスク)で表します。

```
////////////////////////////////////////////////////////
/// SQLの実行(要素の追加) - DBStatementクラスでUPDATE文を指定

   // DBStatementオブジェクトの生成
pStatement = pConnect->CreateStatement();
   // SQL文の実行(繰り返し列の最後の要素位置に追加)
pStatement->Execute("UPDATE TABLE1 ADD F1[*]=ARRAY [100]
                                              WHERE F2=300");
```

(例 2)?パラメタを使用し繰り返し列 F1 の要素の最後に要素を追加(ADD 句)する。

```
////////////////////////////////////////////////////////
/// SQLの実行(要素の追加)
/// - DBPreparedStatementクラスで?パラメタのあるUPDATE文を指定

   // DBPreparedStatementオブジェクトの生成
pStatement = pConnect->CreatePreparedStatement(
              "UPDATE TABLE1 ADD F1[*] =ARRAY[?] WHERE F2=?");
   // SQL文の解析
pStatement -> Execute();
   // ?パラメタへの値の設定
pStatement->SetParam(1,100);
pStatement->SetParam(2,300);
   // 更新の実行
pStatement->ExecuteUpdate();
```

(2)　要素の更新

UPDATE 文では，要素の更新に SET 句を使い，列名の直後に[要素番号]を指定することにより，特定の要素を変更することができます。

(例) 繰り返し列 F1 の要素位置 1 の要素の値を 500 に更新(SET 句)する。

```
////////////////////////////////////////////////////////////
/// SQLの実行(要素の更新) - DBStatementクラスでUPDATE文を指定

   // DBStatementオブジェクトの生成
pStatement = pConnect->CreateStatement();
   // SQL文の実行(要素番号1の値を更新)
pStatement->Execute("UPDATE TABLE1 SET F1[1]=ARRAY [500]
                                              WHERE F2=300");
```

(例)?パラメタを使用し，繰り返し列 F1 の要素位置 1 の要素の値を更新(SET 句)する。

```
////////////////////////////////////////////////////////////
/// SQLの実行(要素の更新)
/// - DBPreparedStatementクラスで?パラメタのあるUPDATE文を指定

   // DBPreparedStatementオブジェクトの生成
pStatement = pConnect->CreatePreparedStatement(
              "UPDATE TABLE1 SET F1[1] =ARRAY[?] WHERE F2=?");
   // SQL文の解析
pStatement -> Execute();
   // ?パラメタへの値の設定
pStatement->SetParam(1,100);
pStatement->SetParam(2,300);
   // 更新の実行
pStatement->ExecuteUpdate();
```

(3)　要素の削除

UPDATE 文では，要素の削除に DELETE 句を使い，列名の直後に[要素番号]を指定することにより，特定の要素を削除することができます。

(例)繰り返し列 F1 の 1 番目の要素を削除(DELETE 句)する。

削除後，要素位置 2 以降の要素がある場合は，一つずつ位置番号が繰り上がります。

```
////////////////////////////////////////////////////////////
/// SQLの実行(要素の削除) - DBStatementクラスでUPDATE文を指定

   // DBStatementオブジェクトの生成
pStatement = pConnect->CreateStatement();
   // SQL文の実行(要素位置1の値を削除)
pStatement->Execute("UPDATE TABLE1 DELETE F1[1] WHERE F2=300");
```

(例)?パラメタを使用し，繰り返し列 F1 の要素位置 1 を削除(DELETE 句)する。

削除後，要素位置 2 以降の要素がある場合は，一つずつ位置番号が繰り上がります。

```
////////////////////////////////////////////////////////////
/// SQLの実行(要素の削除)
/// - DBPreparedStatementクラスで?パラメタのあるUPDATE文を指定

   // DBPreparedStatementオブジェクトの生成
pStatement = pConnect->CreatePreparedStatement(
                     "UPDATE TABLE1 DELETE F1[1] WHERE F2=?");
   // SQL文の解析
pStatement -> Execute();
   // ?パラメタへの値の設定
pStatement->SetParam(1,300);
   // 更新の実行
pStatement->ExecuteUpdate();
```

(4)　INSERT 文を使った繰り返し列を持つレコードの追加

INSERT 文では，繰り返し列のデータを ARRAY 句の後に要素データを指定します。

(例)最初の繰り返し列には 100，200 の要素を持ち，次の繰り返し列には A1,A2 の要素を持つレコードを追加する。

```
////////////////////////////////////////////////////////
/// SQLの実行(追加) - DBStatementクラスでINSERT文を指定

   // DBStatementオブジェクトの生成
pStatement = pConnect->CreateStatement();
   // SQL文の実行
pStatement->Execute("INSERT INTO TABLE1
                     VALUES (ARRAY[100,200],ARRAY['A1','A2'])");
```

2.8　XDM/SD へのアクセス

2.8.1　XDM/SD 接続機能とは

XDM/SD 接続機能とは, DABroker for C++から ACE3 の機能を利用して構造型データベース XDM/SD のレコードへアクセスする機能です。

DABroker を利用したデータベースアクセスは，リレーショナルデータベースへのアクセスであるため，構造型データベースである XDM/SD のデータを直接アクセスできません。XDM/SD 接続機能は，階層構造になっている XDM/SD のデータを，ACE3 と Database Connection Server を利用することで，リレーショナルデータベースであるかのようにアクセスできる機能です。

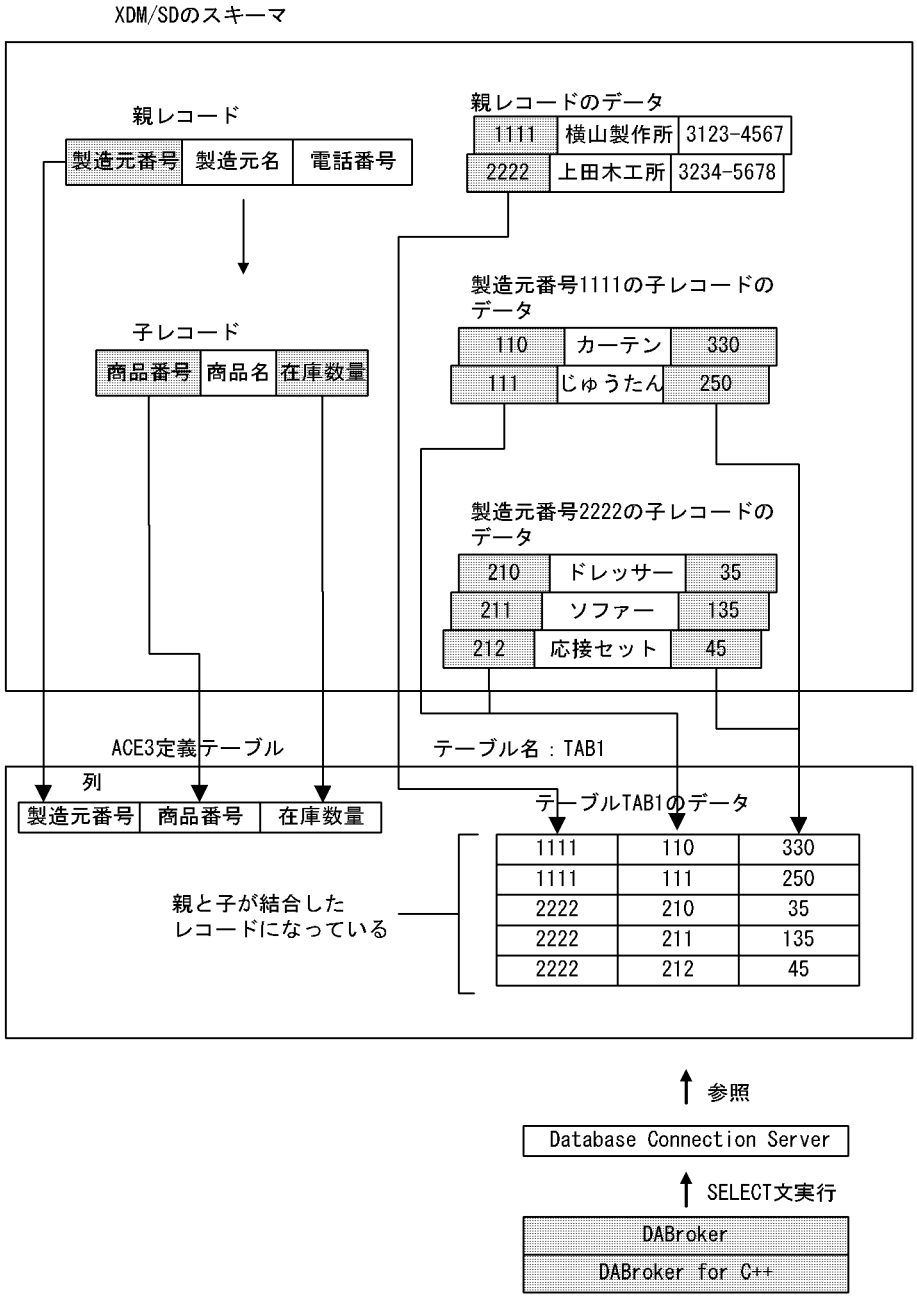
DABroker for C++のクラスライブラリを利用して，次のようなデータベースアクセスができます。

- データの検索と更新・削除（?パラメタ使用可）
- 任意の SQL の実行（詳細版クラスの場合?パラメタ使用可）
- トランザクション制御
- テーブル,フィールド情報などの参照（詳細版クラスの場合）

XDM/SD では，親子関係を持つレコード間の関連を，親子集合で表現し，親レコードは複数の子レコードを持つことができます。このため，ACE3 では，親と子を結合したレコードとすることで，一つのテーブルとしてアクセスします。

XDM/SD のスキーマと ACE3 のテーブルの対応を，図 2-5 に示します。

図 2-5　XDM/SD のスキーマと ACE3 のテーブルの対応



ACE3 で定義したテーブルに対して Database Connection Server の提供する RDML 文の SELECT 文を実行することで，WHERE 句で指定した検索条件に合致したレコードのデータが取得できます。

RDML 文については，マニュアル「Database Connection Server」を参照してください。

前述のテーブル TAB1 に対して次のような SELECT 文を実行します。

```
SELECT 製造元番号，商品番号，在庫数量　FROM TAB1 WHERE 商品番号=210
```

この結果の取得データは次のようになります。

2222	210	35

2.8.2　排他制御

XDM/SD の排他制御には，次の二つの方法があります。

1. C++クラスライブラリを利用する方法

   基本的には C++クラスライブラリを利用して排他制御を行ないます。

   簡易版クラスでは ResultSet に結果を得るとき DBRResultSet クラスの Execute メソッドの引数で，詳細版クラスでは ResultSet に結果を得るとき SetResultSetType メソッドの引数で排他の種類を設定します。排他の種類によって，次のオプションが設定されます。

   TYPE_EXCLUSIVE：LOCK SU

   TYPE_NONE：付加しません（SDEXCLUSIVE 値が仮定されます）。

   TYPE_WAIT：LOCK SR

   TYPE_NOWAIT：LOCK NR

   TYPE_SHARED：LOCK SR

   各オプションの詳細については，マニュアル「Database Connection Server」を参照してください。

2. Database Connection Server の RDML 文で指定する方法

   Database Connection Server の RDML 文の SELECT 文に LOCK 句を付加して排他制御を指定することもできます。

   ただし，資源競合時に共用できない場合，エラーになります。

   また，Database Connection Server の起動制御文の一つであるコントロール空間起動制御文でも指定できます。コントロール空間起動制御文の SDPARAM セクションのオペランド SDEXCLUSIVE，SDLOCKRANGE，SDRELEASE で指定できます。

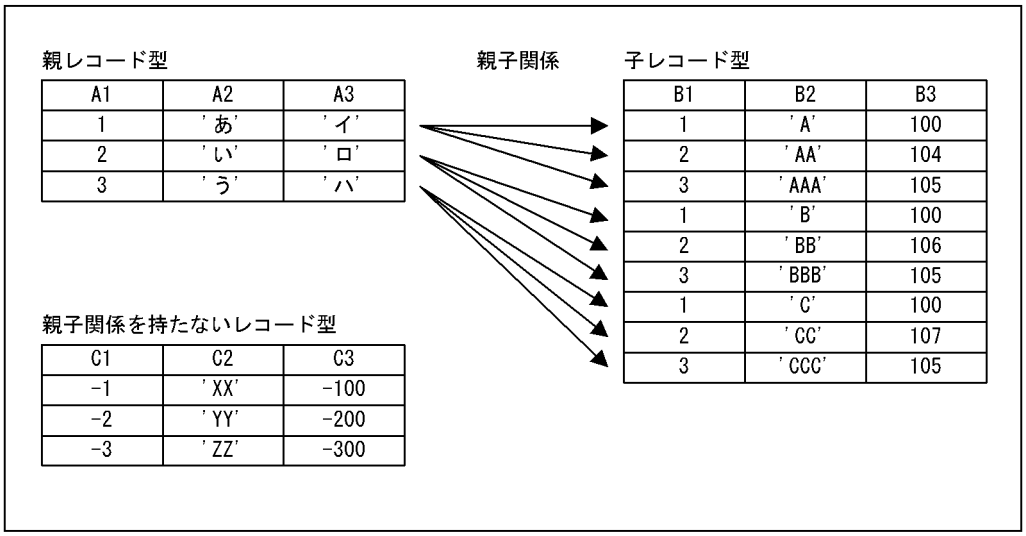
   起動制御文については，マニュアル「Database Connection Server」を参照してください。

2.8.3　データベースアクセス時の制限

XDM/SD のデータベースにアクセスする場合，アクセスできるテーブルやレコードに幾つかの制限があります。ここでは，データ操作ごとの制限について説明します。

以降の説明で使う XDM/SD のレコードの内容，ACE3 のテーブル定義を次に示します。

XDM/SDのスキーマ



A1	A2	A3
1	'あ'	'イ'
2	'い'	'ロ'
3	'う'	'ハ'

B1	B2	B3
1	'A'	100
2	'AA'	104
3	'AAA'	105
1	'B'	100
2	'BB'	106
3	'BBB'	105
1	'C'	100
2	'CC'	107
3	'CCC'	105

C1	C2	C3
-1	'XX'	-100
-2	'YY'	-200
-3	'ZZ'	-300

ACE3のテーブル定義

テーブル：OWNER_TABLE
（親レコード型だけで定義したテーブル）

A1	A2	A3
1	'あ'	'イ'
2	'い'	'ロ'
3	'う'	'ハ'

テーブル：OWNER_MEMBER_TABLE　※
（親レコード型A1と子レコード型B2，B3を用いて定義したテーブル）

A1	B2	B3
1	'A'	100
1	'AA'	104
1	'AAA'	105
2	'B'	100
2	'BB'	106
2	'BBB'	105
3	'C'	100
3	'CC'	107
3	'CCC'	105

テーブル：MEMBER_TABLE
（子レコード型だけで定義したテーブル）

B1	B2	B3
1	'A'	100
2	'AA'	104
3	'AAA'	105
1	'B'	100
2	'BB'	106
3	'BBB'	105
1	'C'	100
2	'CC'	107
3	'CCC'	105

注※　ACE3の仕様によって，親子レコードを含むテーブルでは，親子関係は2階層まで構成できます。

(1)　レコードの検索

次のテーブルのレコードを検索できます。

- 親レコード型だけで定義したテーブル
- 子レコード型だけで定義したテーブル
- 親レコード型と子レコード型を用いて定義したテーブル

- 他のレコードと親子関係を持たないレコード型だけで定義したテーブル

RDML 文の SELECT 文で，WHERE 句を指定して検索した場合，WHERE 句に指定した検索条件に該当するデータを取得できます。

WHERE 句を指定しない場合，SELECT 文で指定したフィールド名のすべてのデータが取得されます。

（例 1）

WHERE 句に指定した条件で検索します。

```
SELECT A1,B2,B3 FROM OWNER_MEMBER_TABLE WHERE A1=1 AND B2='A'
```

検索結果

A1	B2	B3
1	'A'	100

(2)　レコードの追加

次のテーブルのレコードを追加できます。

- 親レコード型だけで定義したテーブル
- 親レコード型と子レコード型を用いて定義したテーブル
- 他のレコードと親子関係を持たないレコード型だけで定義したテーブル

親レコード型だけで定義したテーブルにレコードを挿入した場合，XDM/SD 上には子レコードを持たない親レコードが追加されます。

（例 1）

親レコード型だけで定義したテーブルにレコードを追加します。

```
INSERT INTO OWNER_TABLE VALUES(4,'え','ニ')
```

テーブル上の実行結果

A1	A2	A3
1	'あ'	'イ'
2	'い'	'ロ'
3	'う'	'ハ'
4	'え'	'ニ'

←このレコードは，XDM/SD上では子レコードを　持たないレコードになります。

親レコード型と子レコード型を用いて定義したテーブルにレコードを追加する場合は，テーブルに定義した親レコード型の構成要素が，テーブル定義時にプライマリカラムとしたテーブルであることが必要です。

親レコード型と子レコード型を用いて定義したテーブルにレコードを追加すると，INSERT 文の VALUES 句に指定された挿入値をデータとする親レコードと，親子関係を持つ子レコードが追加されます。

ただし，親レコードはプライマリカラムのため，親レコードに重複するレコードは追加できません。

（例 2）

親レコード型と子レコード型両者で定義したテーブルにレコードを追加します。

```
INSERT INTO OWNER_MEMBER_TABLE(A1,B2,B3) VALUES(4,'D',109)
```

テーブル上の実行結果

テーブル：OWNER_MEMBER_TABLE

A1	B2	B3	
1	'A'	100	
1	'AA'	104	
1	'AAA'	105	
2	'B'	100	
2	'BB'	106	
2	'BBB'	105	
3	'C'	100	
3	'CC'	107	
3	'CCC'	105	
4	'D'	109	←追加レコード

テーブル：OWNER_TABLE

（親レコード型だけで定義したテーブル）

A1	A2	A3	
1	'あ'	'イ'	
2	'い'	'ロ'	
3	'う'	'ハ'	
4	(デフォルト値)	(デフォルト値)	←追加レコード

テーブル：MEMBER_TABLE

（子レコード型だけで定義したテーブル）

B1	B2	B3	
1	'A'	100	
2	'AA'	104	
3	'AAA'	105	
1	'B'	100	
2	'BB'	106	
3	'BBB'	105	
1	'C'	100	
2	'CC'	107	
3	'CCC'	105	
(デフォルト値)	'D'	109	←追加レコード

他のレコードと親子関係を持たないレコード型で定義したテーブルにレコードを追加する場合，XDM/SD 上では INSERT 文の VALUES 句に指定された挿入値をデータとするレコードが追加されます。

レコードを追加したとき，データを入力しないフィールドには省略時の設定値が入ります。テーブルの定義にない XDM/SD のレコード型の項目にも省略時の設定値が適用されます。

例えば，親レコード型の A1，A2 で定義したテーブルに INSERT 文でレコードを追加した場合，追加した親レコードの A3 に’’ (空白)が入ります。

省略時の設定値についてはマニュアル「XDM E2 系 システム定義」を参照してください。

(3)　レコードの削除

次のテーブルのレコードを削除できます。

- 親レコード型だけで定義したテーブル
- 子レコード型だけで定義したテーブル
- 他のレコードと親子関係を持たないレコード型だけで定義したテーブル

親レコードだけで定義したテーブルのレコードを削除した場合，XDM/SD 上では削除した親レコードと親子関係にあった子レコードの動作はスキーマ定義の保存モードの指定に従います。

保存モードについてはマニュアル「XDM E2 系 解説」を参照してください。

(例 1)

親レコード型だけで定義したテーブルのレコードを削除します。

```
DELETE FROM OWNER_TABLE WHERE A1=3
```

テーブル上の実行結果

A1	A2	A3
1	'あ'	'イ'
2	'い'	'ロ'

A1=3と親子関係にあった子レコードは，スキーマ定義の保存モードによって削除の動作が決まります。

子レコード型だけで定義したテーブルのレコードを削除した場合, 一つの子レコードを削除するとその子レコードだけ削除され，親レコードは削除されません。

親レコードに属する子レコードをすべて削除しても，親レコードは削除されません。

(例 2)

子レコード型だけで定義したテーブルのレコードを削除します。

```
DELETE FROM MEMBER_TABLE WHERE B2='A' OR B2='AA' OR B2='AAA'
```

テーブル上の実行結果

B1	B2	B3
1	'B'	100
2	'BB'	106
3	'BBB'	105
1	'C'	100
2	'CC'	107
3	'CCC'	105

親レコード型のA1=1のレコードの子レコードをすべて削除しても，
A1=1の親レコードは削除されません。

他のレコードと親子関係を持たないレコード型で定義したテーブルのレコードを削除した場合，XDM/SD 上でも該当するレコードが削除されます。

(4)　レコードの更新

次のテーブルのレコードを更新できます。

- 親レコード型だけで定義したテーブル
- 子レコード型だけで定義したテーブル
- 親レコード型と子レコード型を用いて定義したテーブル
- 他のレコードと親子関係を持たないレコード型だけで定義したテーブル

親レコード型だけで定義したテーブルの値を更新した場合，XDM/SD 上では該当する親レコードのデータが更新されます。

ただし，CALC キー，シーケンシャルインデックスや子レコードがある場合の接続キーなどの変更はできません。

（例１）

親レコード型だけで定義したテーブルの値を更新します。

```
UPDATE OWNER_TABLE SET A2='ア' WHERE A2='あ'
```

テーブル上の実行結果

A1	A2	A3
1	'ア'	'イ'
2	'い'	'ロ'
3	'う'	'ハ'

子レコード型だけで定義したテーブルの値を更新した場合，XDM/SD 上では該当する子レコードのデータが更新されます。

（例２）

子レコード型だけで定義したテーブルの値を更新します。

```
UPDATE MEMBER_TABLE SET B2='α' WHERE B2='A'
```

テーブル上の実行結果

B1	B2	B3
1	'α'	100
2	'AA'	104
3	'AAA'	105
1	'B'	100
2	'BB'	106
3	'BBB'	105
1	'C'	100
2	'CC'	107
3	'CCC'	105

親レコード型と子レコード型を用いて定義したテーブルのレコードを更新する場合は，テーブルに定義した親レコード型の構成要素が，テーブル定義時にプライマリカラムとしたテーブルである必要があります。この場合，子レコードの値を更新できます。

プライマリカラムである構成要素は値を更新できないので，親レコードの値は更新できません。

（例３）

親レコード型と子レコード型を用いて定義したテーブルの値を更新します。

```
UPDATE OWNER_MEMBER_TABLE SET B2='α' WHERE A1=1 AND B3=100
```

テーブル上の実行結果

A1	B2	B3
1	'α'	100
1	'AA'	104
1	'AAA'	105
2	'B'	100
2	'BB'	106
2	'BBB'	105
3	'C'	100
3	'CC'	107
3	'CCC'	105

他のレコードと親子関係を持たないレコード型で定義したテーブルの値を更新した場合，XDM/SD 上では該当するデータが更新されます。

2.8.4　注意事項

- 同じトランザクション内で，違う表の検索はできません。違う表を検索する場合は，一度トランザクションを終了(Commit，又は Rollback)してください。
- トランザクション内で実行する RDML 文の排他オプションは変更できません。異なる排他オプションの RDML 文を実行した場合，Execute メソッド，及び GetResultSet メソッドでエラーになります。異なる排他オプションの RDML 文を実行する場合は，一度トランザクションを終了(Commit，又は Rollback)してください。

3 アプリケーションの作成

この章では, データベースへアクセスするためのアプリケーションの作成時に必要な項目について説明します。

3.1　ヘッダーファイル

各クラスライブラリを利用するためには，DABroker for C++で提供するヘッダーをコーディング時にインクルード(include)します。インクルードするときは，すべてのクラスを使用するためのヘッダーを一度にインクルードするか，必要なクラスだけのヘッダーをインクルードするかを選択します。表 3-1 にヘッダーファイル一覧を示します。「dbbroker.h」をインクルードすると，クラスを利用するかどうかに関係なくすべてインクルードされます。

表 3-1　ヘッダーファイル一覧

クラス名	ヘッダーファイル名
全クラス	dbbroker.h
DBDriverManager クラス	dbbhdrmg.h
DBDriver クラス	dbbhdriv.h
DBConnection クラス	dbbhcnct.h
DBStatement クラス	dbbhstmt.h
DBResultSet クラス	dbbhrset.h
DBResultSetMetaData クラス	dbbhrsmd.h
DBPreparedStatement クラス	dbbhprep.h
DBCallableStatement クラス	dbbhprcd.h
DBDatabaseMetaData クラス	dbbhdbmd.h
DBTransaction クラス	dbbhtrns.h
DBRArrayDataFactory クラス	dbbharfc.h
DBRArrayData クラス	dbbharry.h
DBRArrayDataPtr クラス	dbbharry.h
DBRArrayDataConstPtr クラス	dbbharry.h
classListTables クラス	dbbhclss.h
classListColumns クラス	dbbhclss.h
classListProcedures クラス	dbbhclss.h
classListProcedureColumns クラス	dbbhclss.h
classListPrimaryKeys クラス	dbbhclss.h
DBRDatabase クラス	dbbhdbrd.h
DBRResultSet クラス	dbbhdbrr.h
DBSQLCA クラス	dbbhexcp.h

3.2　ビルド方法

3.2.1　HP-UX の場合のビルド方法

以下の方法でビルドをしてください。

(1)　コンパイルオプション

DABroker for C++はマルチスレッド対応のライブラリなので，マルチスレッド動作用のコンパイルオプションやリンクする共有ライブラリを指定する必要があります。詳細については，OS に付属するオンラインマニュアル「aCC」,「pthread」を参照してください。

なお，64 ビットモード用コンパイルオプションは使用しないでください。

コンパイル時の指定例を次に示します。少なくとも，以下の指定が必要です。"xxxx.cpp"には，ユーザプログラムのソースファイルを指定してください。

-I オプションでは，DABroker 運用ディレクトリの下の cpp/include ディレクトリを指定します。ここでは DABroker 運用ディレクトリが/opt/DABroker の場合の例を示します。

(a)　HP-UX（PA-RISC）の場合

```
aCC -c +DAportable -D_HPUX_SOURCE -D_POSIX_SOURCE
     -D_POSIX_1003_1C -D_REENTRANT
     -D_POSIX_C_SOURCE=199506L -I/opt/aCC/include
     -I/opt/DABroker/cpp/include xxxx.cpp
```

コンパイラが ac++ A.03.55 以降の場合は-AP オプションを指定してください。

```
aCC -c -AP +DAportable -D_HPUX_SOURCE -D_POSIX_SOURCE
     -D_POSIX_1003_1C -D_REENTRANT
     -D_POSIX_C_SOURCE=199506L -I/opt/aCC/include
     -I/opt/DABroker/cpp/include xxxx.cpp
```

TPBroker の OTS インタフェースを使ってトランザクション制御をする場合は，上記のほかに，TPBroker が指定するコンパイルオプションも指定してください。

- OpenTP1 連携を使ってトランザクション制御をする場合

  コンパイルオプションには最低限次の指定が必要です。"xxxx.cpp"はユーザプログラムのソースファイルを指定してください。

  ```
  aCC -c -AP +DAportable -D_HPUX_SOURCE -D_POSIX_SOURCE
       -D_POSIX_1003_1C -DDABCPP_SINGLE_THREAD -I/opt/aCC/include
       -I/opt/DABroker/cpp/include xxxx.cpp
  ```

  注　この例では，-I オプションで"/opt/DABroker/cpp/include"を指定していますが，実際のコンパイル時には，-I オプションで"DABroker 運用ディレクトリ/cpp/include"を絶対パスで指定してください。

  64 ビットモード用コンパイルオプションは使用しないでください。

(b)　HP-UX（IPF）の場合

```
aCC -c -D_HPUX_SOURCE -D_POSIX_SOURCE
     -D_POSIX_1003_1C -D_REENTRANT
     -D_POSIX_C_SOURCE=199506L -I/opt/aCC/include
     -I/opt/DABroker/cpp/include xxxx.cpp
     -AP +DD32
```

(2)　リンクオプション

リンケージ時の指定例を次に示します。少なくとも，以下の指定が必要です。"xxxx.out"は出力ファイル名を，"xxxx.o"はリンクするオブジェクトファイル名を指定してください。

-L オプションでは，DABroker 運用ディレクトリの下の lib ディレクトリを指定します。ここでは DABroker 運用ディレクトリが/opt/DABroker の場合の例を示します。

(a)　HP-UX（PA-RISC）の場合

```
aCC -Wl,+s -oxxxx.out xxxx.o -L/opt/DABroker/lib -lpthread
    -ldabcpp20
```

コンパイラが ac++ A.03.55 以降の場合は-AP オプションを指定してください。

```
aCC -Wl,+s -AP -oxxxx.out xxxx.o -L/opt/DABroker/lib -lpthread
    -ldabcpp20
```

TPBroker の OTS インタフェースを使ってトランザクション制御をする場合は，上記のほかに，TPBroker が指定するリンクオプションも指定してください。

また，TPBroker の OTS インタフェースを使ってトランザクション制御する場合で HiRDB 使用時は，必ず libzcltyk.sl をリンクしてください。

- OpenTP1 連携を使ってトランザクション制御をする場合

  リンクオプションには，最低限次の指定が必要です。"xxxx.out"は出力ファイル名を，"xxxx.o"はリンクするオブジェクトファイル名を指定してください。マルチスレッド用のライブラリとはライブラリ名が異なりますのでご注意ください。

  ```
  aCC Wl,+s -AP -oxxxx.out xxxx.o -L/opt/DABroker/lib -ldabcpp20st
  ```

  注　この例では-L オプションでは"/opt/DABroker/lib"を指定していますが，実際のリンク時には-L オプションでは絶対パスで"DABroker 運用ディレクトリ/lib"を指定してください。

  HiRDB 使用時は，必ず libzcltys.sl をリンクしてください。

(b)　HP-UX（IPF）の場合

```
aCC -Wl,+s -oxxxx.out xxxx.o -L/opt/DABroker/lib -lpthread
    -ldabcpp32 -AP
```

TPBroker の OTS インタフェースを使ってトランザクション制御する場合で HiRDB 使用時は，必ず libzcltyk.so をリンクしてください。

3.2.2　AIX の場合のビルド方法

以下の方法でビルドをしてください。

(1)　コンパイルオプション

DABroker for C++はマルチスレッド(UNIX98 スレッド)対応のライブラリなので，マルチスレッド(UNIX98 スレッド)動作用のコンパイルオプションやリンクする共有ライブラリを指定する必要があります。詳細については，VisualAge C++のマニュアルを参照してください。

なお，64 ビットモード用コンパイルオプションは使用しないでください。

コンパイル時の指定例を次に示します。少なくとも，以下の指定が必要です。"xxxx.cpp"には，ユーザプログラムのソースファイルを指定してください。

-I オプションでは，DABroker 運用ディレクトリの下の cpp/include ディレクトリを指定します。ここでは DABroker 運用ディレクトリが/opt/DABroker の場合の例を示します。

```
xlC_r -c  -DDAB_AIX -D_POSIX_1003_1C -I/opt/DABroker/cpp/include xxxx.cpp
```

- ヘッダーファイルは，<pthread.h>を取り込みます。
- OpenTP1 連携を使ってトランザクション制御をする場合

  OpenTP1 連携を使ってトランザクション制御をする場合，コンパイルオプションの指定によってシングルスレッド対応のオブジェクトを作成することができます。シングルスレッド対応のオブジェクトを作成する場合の指定例を次に示します。

  ```
  xlC -c  -DDAB_AIX -DDABCPP_SINGLE_THREAD -I/opt/DABroker/cpp/include xxxx.cpp
  ```

(2)　リンクオプション

リンケージ時の指定例を次に示します。少なくとも，次の指定が必要です。

"xxxx.out"は出力ファイル名を，"xxxx.o"はリンクするオブジェクトファイル名を指定してください。

```
xlC_r -oxxxx.out xxxx.o -L/opt/DABroker/lib -ldabcpp20 -lpthreads
```

TPBroker の OTS インタフェースを使ってトランザクション制御する場合で HiRDB 使用時は，必ず libzcltyk.a をリンクしてください。

- OpenTP1 連携を使ってトランザクション制御をする場合

  OpenTP1 連携を使ってトランザクション制御をする場合の指定例を次に示します。

  ```
  xlC -oxxxx.out xxxx.o -L/opt/DABroker/lib -ldabcpp20st
  ```

  HiRDB 使用時は，必ず libzcltys.a をリンクしてください。

3.2.3　Red Hat Linux の場合のビルド方法

以下の方法でビルドをしてください。

(1)　コンパイルオプション

DABroker for C++はマルチスレッド対応のライブラリなので，マルチスレッド動作用のコンパイルオプションやリンクする共有ライブラリを指定する必要があります。詳細については，GCC のマニュアルを参照してください。

なお，64 ビットモード用コンパイルオプションは使用しないでください。

コンパイル時の指定例を次に示します。少なくとも，以下の指定が必要です。"xxxx.cpp"には，ユーザプログラムのソースファイルを指定してください。

-I オプションでは，DABroker 運用ディレクトリの下の cpp/include ディレクトリを指定します。ここでは DABroker 運用ディレクトリが/opt/DABroker の場合の例を示します。

```
g++ -c -fPIC  -D_REENTRANT -D_POSIX_1003_1C
    -D_XOPEN_SOURCE=500
    -D_THREAD_SAFE -DDAB_PCLINUX
    -I/opt/DABroker/cpp/include xxxx.cpp
```

ヘッダーファイルは，<pthread.h>を取り込みます。

(2)　リンクオプション

リンケージ時の指定例を次に示します。少なくとも，次の指定が必要です。

"xxxx.out"は出力ファイル名を，"xxxx.o"はリンクするオブジェクトファイル名を指定してください。

```
g++ -oxxxx.out xxxx.o -L/opt/DABroker/lib
    -ldabcpp40 -lpthread
```

コマンド名称は，使用する Red Hat Enterprise Linux のバージョンによって異なることがあります。

- OpenTP1 連携を使ってトランザクション制御をする場合
  OpenTP1 連携を使ってトランザクション制御をする場合の指定例を次に示します。
  ```
  g++ -oxxxx.out xxxx.o -L/opt/DABroker/lib -ldabcpp40
  ```
  HiRDB 使用時は，必ず libzcltk.so をリンクしてください。

3.2.4　Windows の場合のビルド方法（Visual C++ 5.0，Visual C++ 6.0 の場合）

Visual C++の IDE を使用して，メニュー「プロジェクト」－「設定...」を選択します。

(1)　プロジェクトの設定

1.「C/C++」タブを選択します。
2.「コード生成」カテゴリを選択します。
3.「プロセッサ」設定欄に，Pentium(R)を設定します。
4.「使用するランタイムライブラリ」設定欄に，マルチスレッド(DLL)を設定します。デバッグ版の場合は，マルチスレッド(DLL，デバッグ)を設定します。

(2)　インクルードファイルの設定

1.「C/C++」タブを選択します。
2.「プリプロセッサ」カテゴリを選択します。
3.「インクルードファイルのパス」設定欄に，[インストールディレクトリ]¥cpp¥include を設定します。

（例）C:¥Program Files¥Hitachi¥DABroker¥cpp¥include

(3)　ライブラリの設定

1.「リンク」タブを設定します。
2.「インプット」カテゴリを選択します。
3.「オブジェクト/ライブラリモジュール」設定欄に次のライブラリを追加します。括弧内はデバッグ版です。
  ```
  dabcpp20.lib (dabcpp20d.lib)
  ```
  これらのライブラリは，[インストールディレクトリ]¥cpp¥lib にあります。

TPBroker の OTS インタフェースを使ってトランザクション制御する場合で HiRDB 使用時は，必ず pdcltxm5.lib をリンクしてください。

(4)　プリプロセッサの定義（Database Access for ORB 用のソースを移行する場合）

1.「C/C++」タブを選択します。

2.「プリプロセッサ」カテゴリを選択します。

3.「プリプロセッサの定義」設定欄に，"-DUSE_OLDNAMES"を設定します。

このとき，BOOL 型は BOOLEAN 型(Visual C++の boolean 型)で定義されますが，Visual C++の BOOL 型を使用したい場合には，同じ欄に"-DDAB_NOUSE_BOOL"を追加します。ただし，これを定義すると，DABroker のソースで BOOL 型の変数を引数にしてメソッドを使用している場合（例えば，GetField など）はリンク時にエラーになります。このような場合は，BOOL 型で宣言している変数を BOOLEAN 型で宣言するようにソースを修正する必要があります。

3.2.5　Windows の場合のビルド方法（Visual C++ .NET 2003, Visual Studio の場合）

Visual C++の IDE を使用して，メニュー「プロジェクト」－「プロパティ」を選択します。

(1)　プロジェクトの設定

1. ツリーの「構成プロパティ」－「C/C++」を選択します。

2.「コード生成」カテゴリを選択します。

3.「ランタイムライブラリ」設定欄に，マルチスレッド（DLL）を設定します。デバッグ版の場合は，マルチスレッド（DLL，デバッグ）を設定します。

(2)　インクルードファイルの設定

1. ツリーの「構成プロパティ」－「C/C++」を選択します。

2.「全般」を選択します。

3.「追加のインクルードディレクトリ」に［インストールディレクトリ］¥cpp¥include を設定します。
（例）C:¥Program Files¥Hitachi¥DABroker¥cpp¥include

(3)　ライブラリの設定

1. ツリーの「構成プロパティ」－「リンカ」を設定します。

2.「入力」を選択します。

3.「追加の依存ファイル」に次のライブラリを追加します。括弧内はデバッグ版です。
dabcpp20.lib (dabcpp20d.lib)
このライブラリは，［インストールディレクトリ］¥cpp¥lib にあります。

TPBroker の OTS インタフェースを使ってトランザクション制御する場合で HiRDB 使用時は，必ず pdcltxm5.lib をリンクしてください。

(4)　プリプロセッサの定義（Database Access for ORB 用のソースを移行する場合）

1. ツリーの「構成プロパティ」－「C/C++」を選択します。

2.「プリプロセッサ」を選択します。

3.「プロセッサの定義」設定欄に,”-DUSE_OLDNAMES“を設定します。

このとき，BOOL 型は BOOLEAN 型（Visual C++の boolean 型）で定義されますが，Visual C++の BOOL 型を使用したい場合には，同じ欄に”-DDAB_NOUSE_BOOL”を追加します。ただし，これを定義すると，DABroker のソースで BOOL 型の変数を引数にしてメソッドを使用している場合（例えば，GetField など）はリンク時にエラーになります。このような場合は，BOOL 型で宣言している変数を BOOLEAN 型で宣言するようにソースを修正する必要があります。

3.3　アプリケーション作成上の留意点

3.3.1　オブジェクトの生成と削除

(1)　簡易クラス

簡易クラスでは，Database，ResultSet オブジェクトを生成しデータベースアクセスを行います。このオブジェクトの生成には，DBRDatabase，DBRResultSet クラスのコンストラクタを利用しますので，オブジェクトを生成した関数が終了するとオブジェクトも削除されます。

(2)　詳細クラス

詳細クラスでは，new 演算子や auto 変数の宣言でオブジェクトを生成するのは，DBDriverManager クラスのオブジェクトです。これ以外のクラスのオブジェクトについては，new 演算子などを使った生成ではなく，その親のオブジェクトのメソッドでオブジェクトを生成してください。

new 演算子で生成した DriverManager オブジェクトは，delete 演算子で削除します。これ以外のオブジェクトは，削除メソッドでオブジェクトを削除してください。

例えば，DBStatement クラスや DBResultSet クラスなどはその親のオブジェクトが生成するため，親のクラスで提供する RemoveStatement メソッドや RemoveResultSet メソッド，又は自分自身を削除する Remove メソッドなどで削除してください。

classList～クラスや DBResultSetMetaData クラスなどは，それらの領域を削除するメソッドはありませんが，これらのオブジェクトは親のオブジェクトを削除した時に一緒に削除されます。

(3)　オブジェクト削除後の注意事項

オブジェクトを削除した時は，そのオブジェクトのポインタは使用しないでください。特に，親オブジェクトを削除するケースでは，子のオブジェクトも削除されることに注意ください。

また，ResultSet 中のフィールド値の参照で，DABroker が用意した領域を参照する場合にも注意が必要です。

(4)　オブジェクト名称の自動生成

Transaction，CreateStatement，CreatePreparedStatement，CreateCallableStatement メソッドでは，オブジェクト名称を自動生成しますので，オブジェクト名称が重複しないよう注意が必要です。自動生成する名前は，次のようになっています。

```
DBTransactionオブジェクト名:T_%010d (T_0000000000～T_2147483647)
DBStatementオブジェクト名:S_%010d (S_0000000000～S_2147483647)
DBPreparedStatementオブジェクト名:P_%010d (P_0000000000～P_2147483647)
DBCallableStatementオブジェクト名:C_%010d(C_0000000000～C_2147483647)
```

ここで引数とする数値は，0～2147483647 で 0 から始まり 2147483647 まで進むとまた 0 から始まります。この数値のカウンタは各オブジェクトごとに持っています。このとき，生成した名前を持つオブジェクトが既にあるかどうかのチェックはしません。同じ名前のオブジェクトを二つ以上生成すると異常終了することがあるので，同じ名前を付けることがないようにしてください。

3.3.2　データベースアクセスリソース数の制限

DABroker for C++ で SQL を実行する場合，カーソルというリソースを使用します。このカーソルは，処理内容によって必要な個数が異なります。

一つのカーソルに対する占有期間は，オブジェクトの生存期間に依存するケースとメソッド実行時にだけ必要とするケースの二つのケースに分けることができます。オブジェクトの生存期間に依存するケースは，オブジェクトが作成されてから削除されるまで占有します。メソッド実行時にだけ必要とするケースは，そのメソッドの実行が終了すると解放します。一つのコネクションで使用できるカーソル数は 64 個までです。64 個を超えた場合は，DBSQLCA 例外をスローしますので 64 個を超えないようにプログラムで制御する必要があります。

オブジェクトの生存期間に依存するケースで使用するカーソル数を表 3-1，メソッド実行時にだけ必要とするケースで使用するカーソル数を表 3-2 に示します。

表 3-2　オブジェクトの生存期間に依存するケース

クラス名	DBMS の種類※1	
	A	B
DBConnection※2	1	1
DBStatement	1	1
DBPreparedStatement	1	1
DBCallableStatement※3	2	2
DBRDatabase※2※4	1	1
DBRResultSet※5	1	1

注※1　DBMS の種類は以下のとおりです。

A	HiRDB,VOS3 XDM/RD,VOS3 XDM/SD,VOSK SQL/K,Oracle
B	SQL Anywhere,Adaptive Server Anywhere,SQL Server

注※2　非同期処理の場合は，A,B ともに 0 となります。

注※3　厳密には，オブジェクト生成時は一つだけ使用し，SetProcedure メソッドを実行するともう一つ使用します。

注※4　厳密には，Connect メソッド実行後から Close メソッド実行後まで占有します。

注※5　厳密には，オブジェクトを生成しただけでは使用しません。Execute メソッドを実行したときに使用します。

表 3-3　メソッド実行だけ必要とするケース

クラス名	メソッド名	DBMS の種類※1	
		A	B
DBConnection	ExecuteDirect※2	0	1
DBDatabaseMetaData	GetTable	1	1
	GetColumns	1	1
	GetProcedures	2	2
	GetprocedureColumns	1	1

クラス名	メソッド名	DBMS の種類※1	
		A	B
DBDatabaseMetaData	GetPrimaryKeys	1	1
DBStatement	Execute	0	1
	GetResult	0	1
DBPreparedStatement	Execute	0	1
	GetResult	0	1
DBCallableStatement	SetProcedure	0	1
	Execute	0	1
	Resume	−	1
DBResultSet	Update	1	2
	Delete	1	2
DBRDatabase	ExecuteDirect※2	0	1
DBRResultSet	Update	1	2
	Delete	1	2

注※1　DBMS の種類は以下のとおりです。

A	HiRDB,VOS3 XDM/RD,VOS3 XDM/SD,VOSK SQL/K,Oracle
B	SQL Anywhere,Adaptive Server Anywhere,SQL Server

注※2　非同期処理の場合は，A：1 B：2 となります。

（使用例）

HiRDB 使用時に，DBStatement オブジェクトを二つ，DBCallableStatement オブジェクトを一つ作成し，DBResultSet オブジェクトで Update メソッドを実行した場合のカーソル使用数を次に示します。

クラス数	オブジェクト数	オブジェクト単位のカーソル使用数	カーソル使用数
DBConnection	1	1	1
DBStatement	2	1	2
DBCallableStatement	1	2	2
合計			5

さらに，メソッド実行時に一時的に一つ使用するため，合計 6 個のカーソルを使用することになります。

3.3.3　検索性能の向上策

詳細クラスでは，2バッファ方式を利用することで，レコードの検索性能を向上させることができます。この方式は，取得する ResultSet が SetMaxRows メソッドで指定したレコード数より多く，PageNext メソッドで次の ResultSet を読み込んだりする必要のある場合に効果があります。取得する ResultSet が，SetMaxRows メソッドで指定したレコード数より少ないことが分かっている場合は，2バッファ方式を指定する意味はありません。

2バッファ方式は，DBStatement クラス，DBPreparedStatement クラス，DBCallableStatement クラスの SetResultSetType メソッドで設定します。

なお，ResultSet へ一度にレコードを取得できた場合は，2バッファ方式を指定した場合でも，バッファは一つしか作成されないため，2バッファ方式を指定することで処理が遅くなるようなことはありません。

(1)　バッファを一つ使う場合

DABroker は検索結果を取得するように要求された後，データを取得します。次の検索要求に対する実行は，先に取得したデータの処理が終わった後です。

この方法では，並行して処理を実行できませんが，2バッファの場合と比べて使用するメモリの量が半分で済みます。

また，次の検索結果を取得するまでにユーザの行う処理が短いときには，この方式が有効な場合があります。2バッファ方式を指定して検索すると，DABroker ではスレッドの生成とイベントによる待ち合わせが行われるため，連続して検索結果を取得するような場合には余分な処理によって遅くなることがあります。

(2)　バッファを二つ使う場合

一つめのバッファの検索結果に対してユーザが処理を行っている間に，並行して別のスレッドでもう一つのバッファにデータベースのデータを読み込めます。

デメリットとしては，バッファを一つしか使わない場合と比べて，検索結果を保持しておくためのメモリ使用量が2倍になります。検索結果を保持しておくための領域は検索するフィールド数と一度に取得するレコード数に比例して多く必要になります。

次のような場合，2バッファ方式で検索結果を効率よく取得できます。

- 検索結果を取得した後，次の検索結果を取得するまでにある程度時間（次の検索結果を取得するために掛かる程度の時間）の掛かる処理を行う場合
- 検索結果のデータ量が多い場合

3.3.4　BLOB 型データの取得方法についての制限

BLOB 型データを取得する際に，複数レコードを一度にメモリ上に取得するよう指定した場合(nBLOBType:TYPE_BLOB_MEMORY)，1 レコードにつき 32KB までのデータしか取得できません。32KB を超えるデータをすべて取得するためには，次のどちらかの方法で 1 レコードずつ取得する必要があります。

- DBStatement クラス，DBPreparedStatement クラス，DBCallableStatement クラスの SetMaxRows メソッドで 1 を指定
- SetResultSetType メソッドで，TYPE_EXCLUSIVE を指定

3.3.5　検索中の Commit,Rollback について

データベースの検索は，SQL 文を前処理してカーソルを準備（Execute メソッド）し，その後カーソルをオープン（GetResultSet メソッド）して行います。DBMS によっては Commit 又は Rollback 実行時にカーソルをクローズしたり，前処理を無効にするものがあります。カーソルをクローズすると，その状態で検索を続けることはできません。再度カーソルをオープンする必要があります。前処理が無効になった場合は，検索できません。検索するには，再度 SQL 文を前処理してカーソルを準備し，そのカーソルをオープンする必要があります。

検索時の Commit,Rollback には以下のタイミングがあります。

Execute (DBStatement, DBPreparedStatement)

↓ ① (Commit, Rollback)

GetResultSet (DBStatement, DBPreparedStatement)

↓ ② (Commit, Rollback)

PageNext,Next (DBResultSet)

↓ ③ (Commit, Rollback)

PageNext,Next (DBResultSet)

↓ ④ (Commit, Rollback)

Refresh (DBResultSet)

↓ ⑤ (Commit, Rollback)

PageNext,Next (DBResultSet)

：

- ORACLE の場合

DBStatement クラス, DBPreparedStatement クラスの SetResultSetType メソッドで TYPE_EXCLUSIVE を指定して検索を実行した場合は，①のタイミングではまだカーソルをオープンしていません。④のタイミングでは Refresh メソッドでカーソルを再オープンするためどちらも問題ありませんが，②，③，⑤のタイミングで Commit 又は Rollback を実行した場合はカーソルを閉じてしまうため次の処理に進めません。この場合は Refresh メソッドで再検索を実行する必要があります。

SetResultSetType メソッドで TYPE_NONE を指定した場合は，①～⑤のどのタイミングで実行しても検索処理は継続できます。

- HiRDB, XDM/RD の場合

SetResultSetType メソッドで指定した値にかかわらず，①～⑤のどのタイミングで Commit 又は Rollback を実行してもカーソルが閉じられ，前処理が無効になってしまうため，検索処理は継続できません。再度 Execute メソッドから実行し直す必要があります。

- SQL Anywhere,Adaptive Server Anywhere,SQL Server の場合

①～⑤のどのタイミングで実行しても検索を続行できます。特に制限はありません。

3.3.6　signal 使用時の注意

ユーザプログラムでは，シグナルを明示的にハンドリングしないでください。ユーザプログラムでシグナルをハンドリングした場合の動作は保証できません。

SIGALRM と SIGPIPE は DABroker 内部のシグナルハンドラで処理します。DBMS によっては，その他のシグナルも使用できない場合があるため，シグナルを使用したい場合は，使用する DBMS でそのシグナルが使用できるかどうか確認する必要があります。

3.3.7　Visual C++ 6.0 以降使用時の注意

Visual C++ 6.0 以降を使用してユーザプログラムを新しく作成する場合，型の別名を次のように定義してください。

また，既に作成したユーザプログラムを Visual C++ 6.0 以降でコンパイルする場合，エラーになることがあります。この場合も，型の別名を次のように変更してください。

Visual C++ 5.0 (変更前)	Visual C++ 6.0 以降 (変更後)	対応する標準 C++の型
INT8	DBR_INT8	char
UINT8	DBR_UINT8	unsigned char
INT16	DBR_INT16	short int
UINT16	DBR_UINT16	unsigned short int
INT32	DBR_INT32	long int
UINT32	DBR_UINT32	unsigned long int
SINGLE	DBR_SINGLE	float
DOUBLE	DBR_DOUBLE	double

変更例

- 変更前

```
INT32 i32Data = 10;
pPrep->SetParam(1,i32Data); // Visual C++ 6.0ではエラー
```

- 変更後

```
DBR_INT32 i32Data = 10; // 型の名前をINT32からDBR_INT32に変更
pPrep->SetParam(1,i32Data);
```

Visual C++ 5.0 を使用する場合は，型の別名を変更する必要はありません。従来どおりの名前を使用できます。

3.3.8　暗黙の setlocale 関数の実行

UNIX の場合，DABroker for C++のライブラリロード時に初期化されるグローバルオブジェクトのコンストラクタ内で，以下の setlocale 関数を暗黙に実行します。setlocale 関数は，メソッドの引数にテーブル名やフィールド名などのデータベースの定義情報を指定する場合，2 バイト文字を正しく判定するために使用します。

```
setlocale(LC_ALL,"")
```

このロケールには，DABroker の環境変数 LANG の指定値が適用されますので，DABroker の環境変数 LANG を正しく設定してください。

Windows の場合，setlocale 関数を暗黙に指定することはしません。テーブル名やフィールド名などで 2 バイト文字を使用する場合は，ユーザプログラム中で setlocale 関数を実行してください。

3.3.9　?パラメタに文字列を指定する場合の注意

ORACLE の場合，DBPreparedStatement や DBCallableStatement で使用している?パラメタに，文字列，又は DBR_BINARY の値を指定するときは，次の点に注意してください。

SQL を実行※する場合，前回の実行後に SetParam メソッドで値を変更して再度実行していると，カーソルの再解析が必要になることがあります。この場合は，再度 Execute メソッドを実行して SetParam メソッドで値を設定し直してから SQL を実行してください。

カーソルの再解析が必要になるのは次の場合です。

- 変更前と変更後のデータの長さが次に示す範囲をわたる場合
  1. 0～255 バイト
  2. 256～2000 バイト
  3. 2001 バイト以上

例えば，SetParam メソッドで 200 バイトの文字列を設定して ExecuteUpdate メソッドを実行した後，300 バイトの文字列を SetParam メソッドで設定して再度 ExecuteUpdate メソッドを実行すると，ORACLE のエラーになります。このような場合は，Execute メソッドを実行してから，SetParam メソッドで 300 バイトの文字列を設定して ExecuteUpdate メソッドを実行してください。

注※　ここでは ExecuteUpdate メソッド，又は GetResultSet メソッドの実行を指します。

3.3.10　データベースへの接続処理の複数スレッド同時実行

HiRDB 以外のデータベースへの接続処理，及び切断処理については，スレッド間で排他制御を行っているため，複数のスレッドから同時に実行した場合でも，並行に処理されることはありません。また，接続，切断処理は比較的重い処理であるため，同時に幾つものスレッドで接続処理，切断処理が頻繁に行われるようなプログラムでは，これらの処理が性能低下の原因になることがあります。このような場合，あらかじめデータベースへの接続を幾つか作成しておき，要求があるたびに空いている接続を割り当て，処理が終わった後も切断しないで，次の要求で再利用できるようにします。このような手法は，一般にコネクションプーリングと呼ばれています。

4 簡易版関数詳細

DABroker では，C++による開発向けのプログラミングインタフェースとして，C++クラスを提供します。ここでは，簡易版のクラスごとに，プロパティやメソッドの詳細について説明します。

4.1　文法の説明順序

この章では,メソッドごとに，次のような順序で説明しています。

メソッド名

機能

各メソッドの機能概要について記述しています。

形式

実際に指定する形式について記述しています。

引数は，特に断りのないかぎり，形式に示してある順序で記述してください。

```
void  Connect (LPCTSTR lpctUID,
                LPCTSTR lpctPWD,
                LPCTSTR lpctDBN,
                LPCTSTR lpctOPT = NULL,
                LPCTSTR lpctAbstractName,
                UINT16 swWait = LOCK_OPT_NOWAIT,
                UINT16 swTimeout = 0,
                UINT16 swSync = STMT_SYNC)       throw DBSQLCA
```

引数のうち=のない引数（lpctUIDなど）は，省略できないことを示します。

=のある引数（swWait = LOCK_OPT_NOWAITなど）は，指定を省略できることがあります。途中の引数を一つ又は複数省略することはできません。例えば上記の例では，引数 swWait, swTimeout, swSync のすべてを省略することはできますが，swTimeout だけを省略することはできません。

=の右辺の値は，デフォルト値であることを示しています。この例では，引数 swWait 指定を省略すると LOCK_OPT_NOWAIT が仮定されることを示しています。

引数

形式で説明されている引数とその意味について記述しています。

戻り値

戻り値には，メソッドの戻り値の型（データ型として記述）や説明が記述してあります。

また，主に戻り値の説明中にある（又は機能詳細などにある場合もあります），TRUE 及び FALSE の表記は，それぞれ非 0 及び 0 を意味しています。ヘッダでは TRUE を 1 と定義していますが，これは便宜上のものであり，TRUE の場合でも BOOLEAN 型の戻り値として 1 以外の値が返ることがあります。

したがって，BOOLEAN 型の戻り値で TRUE かどうかを判定する場合は，例 1 のような方法ではなく，例 2 に示す方法で判定してください。

（例 1）

```
BOOLEAN bFlag;
      :
if(bFlag == TRUE)
{
// TRUE
}
```

（例２）

```
BOOLEAN bFlag;
      :
if(bFlag)
{
  // TRUE
}
```

機能詳細

各メソッドの解説が記述してあります。

発生する例外

メソッドでスローする可能性のある代表的なエラーが記述してあります。ここに書かれていないエラーをスローすることもあります。また，項目の先頭に記述されている DBSQLCA(RetCode)は，DBSQLCA クラスの RetCode プロパティにエラーがスローされることを示しています。

DB_ERROR_NOT_ENOUGH_MEMORY

　エラーコードを示します。

アプリケーション作成時には，RetCode プロパティに返るエラーコードを直接，比較，判定に利用できます。なお，アプリケーションで利用するためには，include 文でヘッダーファイル dbbroker.h ファイルをインクルードしておきます。

```
DBSQLCA* pError = NULL;
  try
  {
    :
    pError = custom.GetErrorStatus();
                          // SQL実行時のエラー情報を取得
    if(pError->RetCode != 0)
                          // DBSQLCAオブジェクトのRetCodeで確認
      throw *pError;
    :
  }
  catch(DBSQLCA e)
  {
  }

  try
  {
    :
    pCon->Connect("uid","pass",NULL,"RDB");
                                    // データベースとの接続
    :
  }
  catch(DBSQLCA e)
  {
  if(e.RetCode==DB_ERROR_NOT_ENOUGH_MEMORY)
                                   //エラーコードを判定に使う
    :
  }
```

4.2　簡易版クラスの概要

Database オブジェクトをポイントして ResultSet オブジェクトを生成します。

HiRDB の繰り返し列を利用する場合は，DBRArrayDataFactory クラスを利用し，DBRArrayData オブジェクトを生成します。オブジェクトのデータ（繰り返し列の要素の値）は DBRArrayDataPtr クラス，又は DBRArrayDataConstPtr クラスを利用して取得します。

なお，繰り返し列用のクラスライブラリは，詳細版でも利用できるため，詳細については，「6.　共通関数詳細」を参照してください。

簡易版クラスライブラリのクラスの一覧を表 4-1 に示します。

表 4-1　簡易版クラスライブラリのクラスの一覧

クラス名	概 要
DBRDatabase クラス	接続するデータベースについて管理します。
DBRResultSet クラス	検索した ResultSet について管理します。
DBRArrayDataFactory クラス	繰り返し列を扱う DBRArrayData クラスを管理します。
DBRArrayData クラス	繰り返し列のデータを管理します。
DBRArrayDataPtr クラス	要素の値を変更したり，参照します。
DBRArrayDataConstPtr クラス	要素の値を参照します。

4.3　DBRDatabase クラスの詳細

DBRDatabase クラスが提供しているコンストラクタとメソッドの詳細について説明します。

コンストラクタとメソッドの機能の概要を次に示します。

機 能	コンストラクタ名
Database オブジェクトを生成します。	DBRDatabase

機 能	メソッド名
データベースとの接続を切り離します。	Close
トランザクションをコミットします。	Commit
データベースに接続します。	Connect
検索結果を必要としない SQL 文を実行します。	ExecuteDirect
繰り返し列を扱うために，DBRArrayDataFactory オブジェクトを生成し，ポインタを取得します。	GetArrayDataFactory
SQL 文の非同期実行時のエラー情報を得るために,DBSQLCA オブジェクトへのポインタを取得します。	GetErrorStatus
Database オブジェクトで実行待ち，又は実行中のステートメントがあるかどうかを確認します。	InWaitForDataSource
データベースが切断されているかどうかを確認します。	IsClosed
トランザクションをロールバックします。	Rollback
Database オブジェクトで要求した実行待ち，及び実行中の非同期処理がすべて終了するまで待ちます。	WaitForDataSource

DBRDatabase コンストラクタ

機能

DBRDatabase クラスのコンストラクタであり,Database オブジェクトを生成します。

形式

```
DBRDatabase (LPCTSTR lpctAbstractName,
             LPCTSTR lpctOptions = NULL,
             UINT16 ui16Timeout = 0);
```

- TPBroker の OTS インタフェース，又は OpenTP1 連携を使ったトランザクション制御の場合

```
DBRDatabase (UNIT16 ui16DriverType,
             LPCTSTR lpctOptions = NULL,
             UINT16 ui16Timeout = 0);
```

引数

lpctAbstractName

データベース種別名を指定します。

lpctOptions

DBMS にアクセスするときに，C++クラスライブラリを使ったトランザクション制御にするのか，TPBroker の OTS インタフェース，又は OpenTP1 連携を使ったトランザクション制御にするのかを指定します。次のどちらかの値で設定します。

- NULL ポインタ：C++クラスライブラリを使ったトランザクション制御。NULL ポインタ又は NULL 終端の文字列へのポインタを設定します。
- "XA"：TPBroker の OTS インタフェース，又は OpenTP1 連携を使ったトランザクション制御。OpenTP1 連携時，この値以外を指定した場合は，グローバルトランザクションの制御対象とすることができません。

ui16Timeout

現在のバージョンでは値を指定しても意味を持ちません。指定を省略するか，0 を指定します。

ui16DriverType

次の列挙型の値を指定します。

- DRV_TYPE_ORACLE7：RM として ORACLE を使用します。
- DRV_TYPE_HIRDB：RM として HiRDB を使用します。OpenTP1 連携時，この値以外を指定した場合，動作は保証しません。

ここに示した以外の値を指定した場合，動作は保証しません。

戻り値

なし

機能詳細

DBRDatabase クラスのコンストラクタで Database オブジェクトを生成します。

接続する DBMS の種類に対応したオブジェクトを生成するために,コンストラクタの引数に接続先データベース定義ファイルに記述されているデータベース種別名を指定してください。コンストラクタでは引数のチェックをしますが, エラーは，Commit, Rollback, 又は Connect メソッドを呼び出したときに，それぞれのメソッドで DB_ERROR_OCCURRED_IN_CONSTRUCTOR のエラーがスローされます。

データベースにアクセスする際にトランザクション制御を，C++クラスライブラリ，又は TPBroker の OTS インタフェースのどちらで行うかも指定します。

実際にデータベースに接続するためには，Database オブジェクトを生成した後，Connect メソッドを呼び出してください。その際，Connect メソッドの引数でデータベース名を指定します。接続先データベース定義ファイルの詳細については，「2.1　データベースへの接続と切断」を参照してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しました。

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　指定した引数が誤っています。

DB_ERROR_NOT_SUPPORTED
　サポートしていない DBMS を指定しました。

DB_ERROR_CANNOT_BE_NULL
　lpctAbstractName に NULL が指定されています。

DB_ERROR_DBDEFINITIONNAME_LENGTH_IS_ZERO
　lpctAbstractName に空文字列が指定されています。

DB_ERROR_CANNOT_USE_XADRIVER
　指定した非 XA インターフェース用のデータベース種別名は，既に XA インターフェース用として使われています。

DB_ERROR_CANNOT_USE_NONXADRIVER
　指定した XA インターフェース用のデータベース種別名は，既に非 XA インターフェース用として使われています。

DB_ERROR_DBDEFINITIONNAME_IS_UNAVAILABLE_IN_XA
　lpctAbstractName にデータベース種別名が指定されていますが, lpctOptions に"XA"が指定されています。

Close メソッド

機能

データベースとの接続を切り離します。

形式

```
void Close(void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

データベースを切断します。トランザクションが完了していないコネクションの場合には，ロールバック後，データベースとの接続を切り離します。接続していないときは何もしません。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

DB_ERROR_DAB_ACCESS_ERROR
　DABroker でエラーが発生しました。「8.2　C++クラスライブラリのエラー情報」を参照してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR
　システムコールでエラーが発生しました。「8.2　C++クラスライブラリのエラー情報」を参照してください。

Commit メソッド

機能

トランザクションをコミットします。

形式

```
void Commit(void) throw DBSQLCA
```

引数

なし

機能詳細

トランザクションをコミットします。

コミットする Database オブジェクトに非同期実行時，実行中，又は実行待ちのステートメントがある場合はコミットできません。コミットする前に，InWaitForDataSource メソッドを呼び出して，非同期実行中の処理がないことを確認してください。

コミットが成功すると，その時点から次のトランザクションが開始されます。

戻り値

なし

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_IN_TRANSACTION
　トランザクションが開始されていません。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中ステートメントがあります。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_ERROR_NOT_CONNECTED
　データベースに接続されていません。

DB_ERROR_OCCURRED_IN_CONSTRUCTOR
　コンストラクタでエラーが発生しました。エラーの内容は，GetErrorStatus メソッドで DBSQLCA オブジェクトを取得して参照してください。

DB_ERROR_TIMEOUT
　タイムアウトが発生したため制御を戻します。

Connect メソッド

機能

接続先のデータベース名やユーザ ID，パスワードなどを指定して，データベースに接続します。

また，接続に成功すると，トランザクションを開始します。

形式

```
void Connect (LPCTSTR lpctUID,
              LPCTSTR lpctPWD,
              LPCTSTR lpctDBN,
              LPCTSTR lpctOPT = NULL,
              LPCTSTR lpctAbstractName,
              UINT16 swWait = LOCK_OPT_NOWAIT,
              UINT16 swTimeout = 0,
              UINT16 swSync = STMT_SYNC)
```

引数

lpctUID

接続先データベースのユーザ ID を指定します。

NULL を指定した場合，接続先データベース定義ファイルに指定されたユーザ ID を利用します。

SQL/K，又は XDM/SD の場合，Database Connection Server の認証属性の設定によっては，不正な値を設定してもデータベースに接続できるので注意してください。認証属性の詳細については，マニュアル「Database Connection Server」を参照してください。

TPBroker の OTS インタフェース，又は OpenTP1 連携を使ったトランザクション制御の場合は省略できません。

lpctPWD

接続先データベースのパスワードを指定します。

NULL を指定した場合，接続先データベース定義ファイルに指定されたパスワードを利用します。SQL Anywhere，Adaptive Server Anywhere の場合は，ODBC のデータソース中でパスワードを指定している場合は，そのパスワードを利用できます。

SQL/K，又は XDM/SD の場合，Database Connection Server の認証属性の設定によっては，不正な値を設定してもデータベースに接続できるので注意してください。認証属性の詳細については，マニュアル「Database Connection Server」を参照してください。

TPBroker の OTS インタフェース，又は OpenTP1 連携を使ったトランザクション制御の場合は省略できません。

lpctDBN

- C++クラスライブラリを使ったトランザクション制御の場合
  通常は NULL を指定します。
  データベースによっては，lpctDBN に接続先情報を指定できます。詳しい内容については，このメソッドの機能詳細を参照してください。
- TPBroker の OTS インタフェースを使ったトランザクション制御の場合
  ORACLE 使用時は，xa_open 文字列の DB フィールドで指定するデータベース名を指定します。
  HiRDB 使用時は，NULL を指定します。
- OpenTP1 連携を使ったトランザクション制御の場合
  NULL を指定します。

lpctOPT

- C++クラスライブラリを使ったトランザクション制御の場合
  通常は NULL を指定します。データベースによっては，lpctDBN に接続先情報を指定できます。詳しい内容については，このメソッドの機能詳細を参照してください。
- TPBroker の OTS インタフェースを使ったトランザクション制御の場合
  ORACLE 使用時は，NULL を指定します。
  HiRDB 使用時は，"DABENVGRP=xxxx"という形式で，TM に HiRDB を登録する際に指定した環境変数グループ識別子を指定します。このとき，PDHOST，PDNAMEPORT は指定しないでください。
- OpenTP1 連携を使ったトランザクション制御の場合
  OpenTP1 連携の場合，OpenTP1 に一つの HiRDB しか登録されていない場合は，この引数は省略できます。OpenTP1 に複数の HiRDB が登録されている場合は，環境変数グループ識別子を指定しないとエラーになります。環境変数グループ識別子の詳細については HiRDB のマニュアルを参照してください。
- XDM/SD の場合
  lpctAbstractName に NULL を指定して，lpctOPT で次に示す形式で接続先を指定します。
  `SDNODE=xxxxx;DBHOST=xxxxx;SNDBUFSZ=xxxxx;RCVBUFSZ=xxxxx`

lpctAbstractName

- C++クラスライブラリを使ったトランザクション制御の場合
  接続先データベース定義ファイル中の接続するデータベース名を指定します。
- TPBroker の OTS インタフェース，又は OpenTP1 連携を使ったトランザクション制御の場合
  NULL を指定します。

swWait

検索対象のレコードがほかのトランザクションによってロックされている場合の動作を指定します。ここでの指定が，開始するトランザクションでのデフォルト値になります。DBRResultSet クラスの Execute メソッドの実行時にも指定できます。

SQL/K の場合は，どの値を指定しても，ロックが解除されるまで待ち状態になります。

XDM/SD の場合は，どの値を指定しても，ロックが解除されるまで待たないで，すぐにエラーを返します。

- LOCK_OPT_NOWAIT，又は LOCK_OPT_WITH_ROLLBACK
  ロックの解除を待たないで，すぐにエラーを返します。HiRDB 又は XDM/RD の場合は同時にロールバックを実行します。それ以外の DBMS の場合は，ロールバックは実行しません。
- LOCK_OPT_WITHOUT_ROLLBACK
  ロックの解除を待たないで，すぐにエラーを返します。ロールバックは実行しません。
  ただし，現在のバージョンでは，XDM/RD ではロールバックを実行します。また，HiRDB の DELETE,INSERT,UPDATE 文では，このオプションは無効で，LOCK_OPT_WAIT と同じ動作になります。
- LOCK_OPT_WAIT
  ロックが解除されるまで待ちます。

swTimeout

現在のバージョンでは値を指定しても意味を持ちません。指定を省略する又は 0 を指定します。

swSync

コネクションごとに，データベースアクセス（SQL 文）を同期処理，又は非同期処理のどちらで実行するかを指定します。

STMT_SYNC：SQL 文を同期処理で実行します。

STMT_ASYNC：SQL 文を非同期処理で実行します。

STMT_ASYNC を指定した場合，実行される SQL 文は，子スレッドで処理されます。

- TPBroker の OTS インタフェースでトランザクションを制御する場合
  非同期処理の STMT_ASYNC は指定できません。
- OpenTP1 連携を使ったトランザクション制御の場合
  省略するか，必ず STMT_SYNC を指定します。ただし，グローバルトランザクションの制御対象とする場合，非同期処理はできません。

戻り値

なし

機能詳細

接続するデータベースは，Database オブジェクトの生成時に指定したデータベース種別名と，このメソッドで指定するデータベース名によって決まります。

このメソッドは，非同期実行可能メソッドです。

接続先データベース定義ファイルの定義内容を使用しないで接続する場合の補足事項

データベースとの接続には，接続先データベース定義ファイルの使用をお勧めしますが，もし接続先データベース定義ファイル中の定義内容を使用しない場合には，引数 lpctAbstractName で NULL を指定し，使用する DBMS ごとに次の引数の指定が必要です。

ただし，Database Connection Server 経由で接続するデータベース，SQL Server， 及び Adaptive Server Anywhere を使用する場合は，lpctAbstractName に NULL を指定できません。
lpctAbstractName に NULL を指定しなくても，次の引数に値を指定すれば，接続先データベース定義ファイルの定義内容ではなく，引数に指定した内容で接続します。

- ORACLE の場合
  引数 lpctDBN に，SQL*NET 経由で接続する ORACLE のリスナー名称を指定します。リモートアクセスをしない場合は NULL を指定します。
- HiRDB の場合
  引数 lpctOPT に，" PDHOST = XXXX ; PDNAMEPORT = XXXX "という形式で接続先を指定します。環境変数に PDHOST，PDNAMEPORT が設定されていれば省略できます。
- SQL Anywhere，Adaptive Server Anywhere，SQL Server の場合
  引数 lpctDBN に ODBC のデータソース名を指定します。引数 lpctOPT でデータソース名が定義されている場合は，この引数は省略できます。
  引数 lpctOPT に SQL Anywhere,Adaptive Server Anywhere,SQL Server のデータベース接続パラメタ（ConnectionName 以外）を指定できます。データベース接続パラメタでユーザ ID やパスワードを指定した場合，Connect メソッドの引数に指定したユーザ ID やパスワードよりも優先されます。
- RDA Link for Gateway 経由の XDM/RD の場合
  引数 lpctOPT に，RD ノード名を指定します。
- Database Connection Server 経由の XDM/RD の場合
  引数 lpctOPT に，"RDNODE=xxxxx;DBHOST=xxxxx;SNDBUFSZ=xxxxx;RCVBUFSZ= xxxx" という形式で接続先を指定します。
- Database Connection Server 経由の XDM/SD の場合
  lpctOPT で次に示す形式で接続先を指定します。
  "SDNODE=xxxxx;DBHOST=xxxxx;SNDBUFSZ=xxxxx;RCVBUFSZ=xxxxx"
- Database Connection Server 経由の SQL/K の場合
  lpctOPT で次に示す形式で接続先を指定します。
  "DBID=xxx1;DBHOST=xxx2;SNDBUFSZ=xxx3;RCVBUFSZ=xxx4"
  xxx1：分散定義名を指定します。
  xxx2：コネクションマルチ名を指定します。
  xxx3：通信バッファサイズを指定します。この指定は省略できます。
  xxx4：受信バッファサイズを調整します。この指定は省略できます。
  各項目の詳細については，マニュアル「DABroker」を参照してください。
- TPBroker の OTS インタフェースを使ったトランザクション制御の場合
  引数 lpctAbstractName にだけ NULL を指定します。

HiRDB クライアント環境定義の環境変数指定

HiRDB の場合，HiRDB クライアント環境定義の環境変数を lpctOPT に指定できます。詳細は，「5.4 DBDriver クラスの詳細」の Connect メソッドを参照してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_CONNECTSTRING_INVALID
　接続文字列の形式が間違っています。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_ERROR_OCCURRED_IN_CONSTRUCTOR
　コンストラクタでエラーが発生しました。エラーの内容は GetErrorStatus メソッドで DBSQLCA オブジェクトを取得して参照してください。

DB_ERROR_CANNOT_USE_ABSTRACTNAME
　DBRDatabase オブジェクトにデータベース種別名が指定されていないので，データベース名を指定できません。

DB_ERROR_NOT_FOUND_DBDEFINITION
　データベース種別名，又はデータベース名が見つかりません。

DB_ERROR_CONNECTDBDEFINITIONNAME_LENGTH_IS_ZERO
　データベース名の長さが 0 です。

DB_ERROR_TIMEOUT
　タイムアウトが発生したため制御を戻します。

DB_ERROR_NOT_EXECUTE_ASYNC
　XA インターフェース使用時に非同期処理は指定できません。

ExecuteDirect メソッド

機能

検索結果を必要としない，SQL 文を実行します。

形式

```
void ExecuteDirect(LPCTSTR lpctStatement) throw DBSQLCA
```

引数

lpctStatement

実行する SQL 文を指定します。

機能詳細

引数 lpctStatement で指定された SQL 文を実行します。

このメソッドでは，?パラメタを使った SQL 文を指定できません。また，SELECT 文を実行しても，検索結果は取得できません。

SQL の実行方法については「2.5 簡易版クラスのデータベースアクセス」を参照してください。

非同期に６４個までの SQL 文が実行できます。

このメソッドは，非同期実行可能なメソッドです。

戻り値

なし

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_NOT_CONNECTED
　データベースに接続していません。

DB_ERROR_CANNOT_BE_NULL
　引数に NULL を指定しました。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_ERROR_DAB_ACCESS_ERROR
　DABroker でエラーが発生しました。「8.2　C++クラスライブラリのエラー情報」を参照してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR
　システムコールでエラーが発生しました。「8.2　C++クラスライブラリのエラー情報」を参照してください。

DB_ERROR_TIMEOUT
　タイムアウトが発生しました。

GetArrayDataFactory メソッド

機能

繰り返し列を扱うために，DBRArrayDataFactory オブジェクトを生成し，ポインタを取得します。

形式

```
DBRArrayDataFactory * GetArrayDataFactory(void) throw DBSQLCA
```

引数

なし

戻り値

データ型：DBRArrayDataFactory *

DBRArrayDataFactory オブジェクトのポインタ。

機能詳細

GetArrayDataFactory メソッドは，DBRArrayData オブジェクトを管理するための DBRArrayDataFactory オブジェクトを生成し，ポインタを取得するメソッドです。

この DBRArrayDataFactory オブジェクトの生成後，CreateArrayData メソッドで DBRArrayData オブジェクトを生成します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_DRIVER_NOT_LOADED
　データベースとの接続時に非同期処理を設定していますが，接続に失敗し，また，接続する DBMS 種別も不明です。

GetErrorStatus メソッド

機能

SQL 文の非同期実行時のエラー情報を得るために，DBSQLCA オブジェクトへのポインタを取得します。

形式

```
DBSQLCA *GetErrorStatus(void)
```

引数

なし

機能詳細

DBSQLCA オブジェクトへのポインタを取得します。

非同期処理のときは，エラーが発生しても直にエラーを取得できません。このため，ユーザは任意の時点でエラーを確認する必要があります。

非同期実行時にエラーが発生した場合，DBSQLCA オブジェクトにエラー情報が設定されます。

DBSQLCA オブジェクトでは，非同期実行中に発生したエラー情報を最大 100 回分保存できます。エラー情報は，新しいエラー情報から 100 回分保存されるため，エラー情報を確認したあとは DBSQLCA クラスの Delete メソッドを呼び出して，不要なエラー情報をクリアしておいてください。

同期実行時も DBSQLCA オブジェクトへのポインタを取得しますが，コンストラクタでエラーが発生しなかった場合，取得したオブジェクトにエラーは設定されません。

戻り値

データ型：DBSQLCA*

DBSQLCA オブジェクトへのポインタ。

発生する例外

なし

InWaitForDataSource メソッド

機能

Database オブジェクトで実行待ち，又は実行中のステートメントがあるかどうかを確認します。同じ Database オブジェクトで実行しているすべてのステートメントが対象になります。

形式

```
BOOLEAN  InWaitForDataSource(void)
```

引数

なし

戻り値

データ型：BOOLEAN

TRUE：非同期処理の継続中です。

FALSE：非同期処理は終了しています。

機能詳細

子スレッドで実行待ち又は実行中の SQL 文があれば TRUE を，実行待ち又は実行中の SQL 文がなければ FALSE を返します。

厳密には，SQL を実行する子スレッドがあれば TRUE を，なければ FALSE を返します。同期実行接続時には常に FALSE を返します。

なお，DBRResultSet クラスの InExecute メソッドと，InWaitForDataSource メソッドの違いは，前者が InExecute メソッドを実行したオブジェクトの非同期処理が実行中かどうかを確認するのに対して，後者は Database オブジェクトで生成したすべてのオブジェクトの非同期処理が実行中かどうかを確認できます（対象となるコネクション内のすべての非同期処理が終わるのを確認できます）。

発生する例外

なし

IsClosed メソッド

機能

データベースが切断されているかどうかを確認します。

形式

```
BOOLEAN  IsClosed(void)
```

引数

なし

戻り値

データ型：BOOLEAN

TRUE：データベースと接続されていません。

FALSE：データベースと接続しています。

機能詳細

なし

発生する例外

なし

Rollback メソッド

機能

トランザクションをロールバックします。

形式

```
void Rollback(void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

トランザクションをロールバックします。

対象とするコネクションで非同期処理を実行している場合，実行中又は実行待ちの処理があるとロールバックできません。ロールバックする前に，InWaitForDataSource メソッドを呼び出して，非同期実行中の処理がないことを確認してください。

ロールバックが成功すると，その時点から次のトランザクションが開始されます。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中ステートメントがあります。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_ERROR_NOT_CONNECTED
　データベースに接続していません。

DB_ERROR_OCCURRED_IN_CONSTRUCTOR
コンストラクタでエラーが発生しました。エラーの内容は，GetErrorStatus メソッドで DBSQLCA オブジェクトを取得して参照してください。

DB_ERROR_TIMEOUT
タイムアウトが発生しました。

WaitForDataSource メソッド

機能

Databese オブジェクトで要求した実行待ち，及び実行中の非同期処理がすべて終了するまで待ちます。非同期処理中の同期を取りたい場合に利用できます。

形式

```
BOOLEAN WaitForDataSource(UINT32 swWaitTime = DBR_INFINITE)
```

引数

dwWaitTime

非同期処理の終了を待つ最大時間（単位はミリ秒），又は DBR_INFINITE を指定します。

指定した時間内に非同期処理が終了した場合，又は非同期実行中の SQL がない場合は，TRUE が返ります。指定した値を経過しても非同期処理が終了しない場合，FALSE が返ります。

DBR_INFINITE を指定した場合はタイムアウト時間を設定しません。非同期処理がすべて終了するまで待ち続けます。

戻り値

データ型：BOOLEAN

TRUE：すべての非同期処理が終了しました。

FALSE：タイムアウト時間が経過しました。

機能詳細

非同期処理の実行待ち及び実行中の SQL が，終了するのを待ちます。

なお，DBRResultSet クラスの WaitForDataSource メソッドと，このクラスの WaitForDataSource メソッドの違いは，前者が WaitForDataSource メソッドを実行したオブジェクトの非同期処理の終了を待つのに対して，後者は Database オブジェクトで生成したすべてのオブジェクトの非同期処理の終了を待ちます（対象となるコネクション内のすべての非同期処理が終わるのを待ちます）。

同期実行時はすぐに TRUE を返します。

発生する例外

なし

4.4　DBRResultSet クラスの詳細

DBRResultSet クラスが提供しているコンストラクタ,メソッドと仮想関数の詳細について説明します。それらの機能の概要を次に示します。メソッドは,次の六つの機能に分類します。

- レコードの検索
- Resultset の操作
- カーソルの操作
- 検索したレコードの参照
- 検索したレコードの更新と削除
- 非同期メソッドの終了確認

- **コンストラクタ**

機 能	コンストラクタ名
Database オブジェクトから ResultSet オブジェクトを生成します。	DBRResultSet

- **レコードの検索に関するメソッド**

機 能	メソッド名
データベースに SQL 実行の情報を通知します。	Execute
SetParam メソッドで指定したパラメタの値を取得します。	GetParam
パラメタの数を取得します。	GetParamCount
パラメタの値が NULL かどうか確認します。	IsParamNull
検索結果を ResultSet に読込みます。	Open
先頭のレコードから再読み込みをします。	Refresh
?パラメタに値を設定します。	SetParam
パラメタに NULL を設定します。	SetParamNull
?パラメタの属性を設定します。	SetParamType

- **ResultSet の操作に関するメソッド**

機 能	メソッド名
ResultSet オブジェクトを削除します。	Close
SetMaxRows メソッドで指定したレコード数を取得します。	GetMaxRows
ResultSet に読込まれたレコード数を取得します。	GetRowCount
ResultSet に読込むレコード数の最大値を指定します。	SetMaxRows

- **カーソルの操作に関するメソッド**

機 能	メソッド名
カーソルを ResultSet の先頭から n 番目のレコードに移動します。	Absolute
カーソルを ResultSet の最後のレコードへ移動します。	Bottom
検索したすべてのレコードの先頭から数えた，カレントレコードの位置を取得します。	GetCurrent
ResultSet の先頭レコードから数えた，カレントレコードの位置を取得します。	GetCurrentOfResultSet
カーソルの位置が，ResultSet のレコードの最後を超えたかどうかを確認します。	IsEOF
カーソルを次のレコードへ移動します。	Next
カーソルを次の ResultSet の先頭へ移動します。	PageNext
カーソルを前のレコードへ移動します。	Previous
カーソルを現在のレコードの位置から n 個分移動します。	Relative
カーソルを ResultSet の先頭のレコードへ移動します。	Top

- 検索したレコードの参照に関するメソッド

機 能	メソッド名
繰り返し列の場合に要素の数を取得します。	GetArraySize
カレントレコードのフィールド値を取得します。	GetField
検索結果のフィールドの数を取得します。	GetFieldCount
フィールドのデータ型を C++のタイプで取得します。	GetFieldCType
フィールドのデータ型を DBMS のタイプで取得します。	GetFieldDBType
フィールド名称を取得します。	GetFieldName
フィールドの精度（桁数）を取得します。	GetFieldPrecision
フィールドの小数点以下の桁数を取得します。	GetFieldScale
フィールドのデータ型を取得します。	GetFieldType
フィールドの値が NULL かどうかを確認します。	IsFieldNull

- 検索したレコードの更新と削除に関するメソッド

機 能	メソッド名
カレントレコードを削除します。	Delete
カレントレコードを更新・削除するための準備をします。	Edit
指定されたフィールドにデータを設定します。	SetField
指定されたフィールドに NULL を設定します。	SetFieldNull
フィールドの更新結果をデータベースへ通知します。	Update

- 非同期実行に関するメソッド

機 能	メソッド名
SQL 文の非同期実行時のエラー情報を得るために，DBSQLCA オブジェクトのポインタを取得します。	GetErrorStatus
DBRResultSet オブジェクトに非同期実行中（又は実行待ち）のステートメントがあるかどうかを確認します。	InExecute
DBRResultSet オブジェクトで要求した実行待ち，及び実行中の非同期処理のメソッドが終了するまで待ちます。	WaitForDataSource

- 仮想関数

機 能	仮想関数名
Open メソッドを呼び出した場合はレコードが最初に読み込まれる前に，Refresh メソッドを呼び出した場合はレコードが読み込まれる前に呼び出される仮想関数です。	OnBeforeRefresh
カーソルが ResultSet の最後のレコードを超えたときに呼び出される仮想関数です。	OnEndRecord
カーソルが移動したときに呼び出される仮想関数です。	OnMoveRecord

DBRResultSet コンストラクタ

機能

DBRResultSet クラスのコンストラクタであり, Database オブジェクトから ResultSet オブジェクトを生成します。

形式

```
DBRResultSet(DBRDatabase *pDatabase)
```

引数

pdatabase

Database オブジェクトのポインタを指定します。

戻り値

なし

機能詳細

DBRResultSet クラスのコンストラクタで ResultSet オブジェクトを生成します。

ResultSet オブジェクトを生成後，Execute メソッドでレコードの検索条件を指定し,Open メソッドで検索レコードを ResultSet に読込んでください。

一つの Database オブジェクトに対して ResultSet オブジェクトは 64 まで指定できます。既に 64 個の ResultSet オブジェクトで使用されている場合,Execute メソッド実行時に，DB_ERROR_NOT_RELATIONSHIP_TO_DATABASE のエラーがスローされます。

発生する例外

なし

Absolute メソッド

機能

カーソルを ResultSet の先頭から n 番目のレコードに移動します。

形式

```
void Absolute(UINT32 dwCount) throw DBSQLCA
```

引数

dwCount

カーソルを検索した総レコードの何番目に位置付けるかを指定します。

ResultSet の範囲を超えるような番号は指定できません。

戻り値

なし

機能詳細

カーソルを ResultSet の先頭から引数 dwCount で指定した番号のレコードに移動します。

カーソルが検索したレコードの最後を超えている場合 (IsEOF メソッドで TRUE が返る場合), このメソッドは呼び出せません。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_RESULTSET_NOT_EXIST
　非同期実行時に，検索結果がまだ取得できていません。

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

DB_ERROR_OUT_OF_RESULTSET
　ResultSet の範囲を超えるような呼び出しです。

Bottom メソッド

機能

カーソルを ResultSet の最後のレコードへ移動します。

形式

```
void Bottom(void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

カーソルを ResultSet の最後のレコードへ移します。

カーソルが検索したレコードの最後を超えている場合（IsEOF メソッドで TRUE が返る場合），このメソッドは呼び出せません。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_RESULTSET_NOT_EXIST
　非同期実行時に，検索結果がまだ取得できていません。

DB_ERROR_OUT_OF_RESULTSET
　ResultSet の範囲を超えるような呼び出しです。

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

Close メソッド

機能

ResultSet を削除します。

形式

```
void Close(void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

なし

発生する例外

DBSQLCA(RetCode)

DB_ERROR_BEFORE_EXECUTE
　EXECUTE メソッドが実行されていません。

Delete メソッド

機能

カレントレコードを削除します。

形式

```
void Delete(void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

カレントレコードを削除します。削除する前に Edit メソッドを呼出して削除の準備をしてください。削除した後で Commit メソッドを呼び出してデータベースの更新を確定する必要があります。

このメソッドは，非同期実行可能なメソッドです。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_NOT_IN_EDIT
　Edit メソッドが実行されていません。

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行中です。

DB_ERROR_RESULTSET_NOT_EXIST
　非同期実行時に，検索結果がまだ取得できていません。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_ERROR_DAB_ACCESS_ERROR
　DABroker でエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR
　システムコールでエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_TIMEOUT
　タイムアウトが発生したため制御を戻します。

Edit メソッド

機能

カレントレコードを更新・削除するための準備をします。

形式

```
void Edit(void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

カレントレコードを更新又は削除する準備をします。

実際にカレントレコードを更新するためには，Edit メソッドを呼び出した後，SetField メソッドを呼び出してフィールドの値を更新します。必要なデータを更新した後に Update メソッドを呼び出すと，カレントレコードの更新情報はデータベースへ送信されます。カレントレコードを削除するためには, Delete メソッドを呼び出します。

なお，レコードを更新又は削除した後で Commit メソッドを発行してデータベースの更新を確定する必要があります。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_CURRENT_RECORD_DELETED
　カレントレコードが削除されています。

DB_ERROR_RESULTSET_NOT_EXIST
　非同期実行時に，検索結果がまだ取得できていません。

DB_ERROR_CANNOT_EDIT_LAST_RECORD
　カレントレコードがありません(検索結果の最後がカレントレコードの場合)。

DB_ERROR_CANT_UPDATE
　ResultSet が読み込み専用です。

Execute メソッド

機能

引数で指定した SELECT 文と ResultSet 生成のための情報をデータベースに通知し，検索結果を ResultSet に取得する準備をします。

形式

```
void Execute (LPCTSTR lpctStatement,
              UINT16 swWait = LOCK_OPT_DEFAULT,
              UINT16 swType = TYPE_EXCLUSIVE|BUFFER_TYPE_SINGLE,
              DBR_BLOB_TYPE nBLOBType = TYPE_BLOB_MEMORY,
              LPCTSTR lpctBLOBFileName = NULL)    throw DBSQLCA
```

引数

lpctStatement

SQL 文を指定します。SELECT 文だけが指定できます。?パラメタを使用した SELECT 文も指定できます。

SELECT 文以外の SQL を実行する場合は，DBRDatabase オブジェクトの ExecuteDirect メソッドを使用してください。

swWait

実行する SQL 単位に，検索対象のレコードがほかのトランザクションによってロックされていた場合の動作を指定します。

SQL/K の場合は，どの値を指定しても，ロックが解除されるまで待ち状態になります。

XDM/SD の場合は，どの値を指定しても，ロックが解除されるまで待たないで，すぐにエラーを返します。

次の値のうち，どれか一つを指定します。

- LOCK_OPT_DEFAULT
  親オブジェクトの DBRDatabase オブジェクトの設定を引き継ぎます。
- LOCK_OPT_NOWAIT，又は LOCK_OPT_WITH_ROLLBACK
  ロックの解除を待たないで，すぐにエラーを返します。HiRDB 又は XDM/RD の場合は同時にロールバックを実行します。それ以外の DBMS の場合は，ロールバックは実行しません。
- LOCK_OPT_WITHOUT_ROLLBACK
  ロックの解除を待たないで，すぐにエラーを返します。ロールバックは実行しません。
  ただし，現在のバージョンでは，XDM/RD ではロールバックを実行します。また，HiRDB の DELETE,INSERT,UPDATE 文では，このオプションは無効で，LOCK_OPY_WAIT と同じ動作になります。
- LOCK_OPT_WAIT
  ロックが解除されるまで待ちます。

swType

次の三つの値を指定します。

- 検索したレコードに対するロック方法
- 検索時に使用するバッファ数
- VARCHAR データを DBR_BINARY 型で取得した場合の Length メンバ，RealLength メンバの意味

指定する値は，上記三つの値の論理和になります。

このメソッドを実行しなかった場合の動作は，TYPE_EXCLUSIVE | BUFFER_TYPE_SINGLE | VARCHAR_LENGTH_DEF を指定した場合と同じになります。

検索したレコードに対するロック方法

次に示す TYPE_EXCLUSIVE，TYPE_NONE，TYPE_WAIT，TYPE_NOWAIT，TYPE_SHARED のどれかの値で，検索したレコードに対するロック方法を指定します。

- TYPE_EXCLUSIVE

  検索したレコードを DBRResultSet オブジェクトの Update，Delete メソッドで，更新，削除をする場合は，このオプションを指定します。検索したレコードに排他ロックが掛かり，他のトランザクションからの参照・更新を制限できます。実際に排他ロックが掛かるのは，Open メソッドを実行した時点からです。

  このオプションを指定した場合，ResultSet に検索できるレコードは 1 レコードだけで，SetMaxRows で指定したレコード数は無視されます。また，検索条件に一致する次のレコードをアクセスするには，Next メソッドを使用します。TYPE_EXCLUSIVE で取得するレコードの場合，カーソル制御のメソッドとしては Next メソッドだけが指定できます。

  なお，この引数の値は，実行時に SQL 文のオプション文字列に変換されて実行されます。DBMS 別に，どのオプション文字列へ変換されて実行されるのかを次に示します。

  - ORACLE：FOR UPDATE
  - SQL Anywhere,Adaptive Server Anywhere：オプション文字列は付加しません。
  - HiRDB：FOR UPDATE（"WITH EXCLUSIVE LOCK"が仮定されます）
  - XDM/RD：FOR UPDATE（"WITH EXCLUSIVE LOCK"が仮定されます）
  - SQL Server：(テーブル名の後に)UPDLOCK
  - SQL/K：FOR UPDATE

    詳細については，マニュアル「SQL/K」を参照してください。
  - XDM/SD：LOCK SU

- TYPE_NONE

  参照専用で ResultSet に検索結果を読み込みます。

  DBMS 種別によって，検索するレコードに対して参照ロックが掛かかります。このオプションを指定した場合，一度に複数レコードを ResultSet に読み込めます。

  なお，この引数の値は，実行時に SQL 文のオプション文字列に変換されて実行されます。DBMS 別に，どのオプション文字列へ変換されて実行されるのかを次に示します。

  - ORACLE：付加しません。
  - SQL Anywhere,Adaptive Server Anywhere：付加しません。
  - HiRDB：付加しません。
  - XDM/RD：付加しません。
  - SQL Server：付加しません。
  - SQL/K：付加しません。
  - XDM/SD：付加しません(SDEXCLUSIVE 値が仮定されます)。

    SDEXCLUSIVE 値の詳細については，マニュアル「Database Connection Server」のコントロール空間起動制御文を参照してください。

- TYPE_WAIT(HiRDB，XDM/RD，SQL Server，SQL/K，XDM/SD の場合だけ有効です)

  参照専用で ResultSet に検索結果を読み込みます。

検索したレコードに対して参照ロックを掛けますが，見終わった行から排他制御を解除します。このため，DABroker for C++では，ResultSet にレコードを読み込んだ時点，つまり，Open メソッド，又は PageNext メソッドが完了した時点で，ロックが解除されています。

このオプションを指定した場合，一度に複数レコードを ResultSet に読み込めます。

なお，この引数の値は，実行時に SQL 文のオプション文字列に変換されて実行されます。DBMS 別に，どのオプション文字列へ変換されて実行されるのかを次に示します。

- HiRDB：WITHOUT LOCK WAIT
- XDM/RD：WITHOUT LOCK WAIT
- SQL Server：(テーブル名の後に)HOLDLOCK
- SQL/K：付加しません。
- XDM/SD：LOCK SR

- TYPE_NOWAIT(HiRDB，XDM/RD，SQL Server，SQL/K，XDM/SD の場合だけ有効です)

  参照専用で ResultSet に検索結果を求めます。

  該当するレコードがほかのトランザクションで更新中だったり，TYPE_EXCLUSIVE で検索している場合でも読み込めます。排他制御を全くしません。

  このオプションを指定した場合，一度に複数レコードを ResultSet に読み込めます。

  なお，この引数の値は，実行時に SQL 文のオプション文字列に変換されて実行されます。DBMS 別に，どのオプション文字列へ変換されて実行されるのかを次に示します。

  - HiRDB：WITHOUT LOCK NOWAIT
  - XDM/RD：WITHOUT LOCK NOWAIT
  - SQL Server：(テーブル名の後に)NOLOCK
  - SQL/K：付加しません。
  - XDM/SD：LOCK NR

- TYPE_SHARED(HiRDB，XDM/RD，SQL/K，XDM/SD の場合だけ有効です)

  検索するレコードに共有ロックを掛けます。共有ロックの掛かったレコードは，ほかのトランザクションから参照できますが，更新はできません。

  - HiRDB：WITH SHARE LOCK
  - XDM/RD：WITH SHARE LOCK
  - SQL/K：付加しません。
  - XDM/SD：LOCK SR

検索時に使用するバッファ数

次に示す BUFFER_TYPE_SINGLE，又は BUFFER_TYPE_DOUBLE のどちらかの値で，検索時に使用するバッファ数を指定します。指定を省略した場合は，BUFFER_TYPE_SINGLE が仮定されます。なお，バッファの利用方法については，「3.3.3 検索性能の向上策」を参照してください。

- BUFFER_TYPE_SINGLE

  検索結果を保持するバッファを一つだけ使います。

- BUFFER_TYPE_DOUBLE

  検索結果を保持するバッファを二つ使います。

VARCHAR データを DBR_BINARY 型で取得したときの Length メンバ，RealLength メンバの意味

次に示す値のどれか一つを指定します。

- VARCHAR_LENGTH_DEF
  - GetField メソッドの形式 2 を使用した場合，又は形式 3 を使用してもデータの切り捨てが発生しなかった場合
    Length：カラムの定義長
    RealLength：0
  - GetField メソッドの形式 3 を使用してデータの切り捨てが発生した場合
    Length：切り捨てたデータ長
    RealLength：カラムの定義長
- VARCHAR_LENGTH_REAL
  - GetField メソッドの形式 2 を使用した場合，又は形式 3 を使用してもデータの切り捨てが発生しなかった場合
    Length：実際のデータ長
    RealLength：0
  - GetField メソッドの形式 3 を使用してデータの切り捨てが発生した場合
    Length：切り捨てたデータ長
    RealLength：実際のデータ長

nBLOBType

BLOB 型データの取得方法として，次のどちらかを指定します。

- TYPE_BLOB_MEMORY
  メモリ上に一括して取得します。
  ResultSet のレコード数が 1 の場合，BLOB 型データをすべて取得します。
  ResultSet のレコード数が複数の場合，先頭から 32K バイトまでを取得します。
- TYPE_BLOB_FILE
  ファイルを経由して取得します。
  引数 lpctBLOBFileName に指定された名称が，BLOB 型データを格納するファイル名称のプレフィックスとして使用されます。
  なお，引数 swType で BUFFER_TYPE_DOUBLE を指定している場合，検索の途中で終了しても先読みしている ResultSet 分のファイルは作成されます。

lpctBLOBFileName

BLOB 型データを格納するファイル名称のプレフィックスを指定します。

引数 nBLOBType で TYPE_BLOB_FILE を指定した場合，BLOB 型データを格納するファイルのプレフィックスには，この引数で指定した名称が使われます。TYPE_BLOB_FILE 以外の値を指定した場合は，この引数での指定は無視されます。

BLOB 型データを格納したファイル名称は，「プレフィックス+レコード番号+フィールド番号」で表されます。NULL が指定された場合，DABroker が自動的に名称を生成します。

戻り値

なし

機能詳細

検索のための SELECT 文と，検索レコードに対するロック方法，検索時にバッファを幾つ使うか，及び BLOB 型データの取得方法についてデータベースに通知し，検索結果を ResultSet に取得する準備をします。

ファイル上で BLOB 型データを扱う場合の制限事項

データの格納されているファイルは，DABroker が終了しても削除されません。アプリケーション自身で削除してください。

作成されたファイルのアクセス権限は，アプリケーションの実行ユーザ及びデフォルトの権限に依存します。例えば，HP-UX 上で"user1"が実行しているアプリケーションで，デフォルト umask が 0666 の場合，取得した BLOB 型データが格納されるファイルも，"user1"の所有する 0666 のアクセス権限が付与されたファイルとして生成されます。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_ERROR_NOT_CONNECTED
　データベースに接続されていません。

DB_ERROR_NOT_SELECT_STATEMENT
　指定された SQL 文は SELECT 文ではありません。

DB_ERROR_DAB_ACCESS_ERROR
　DABroker でエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR
　システムコールでエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_DRV_ERROR_PHOLDER_AND_QP
　プレースホルダと？パラメタが混在する SQL 文の指定はできません。

DB_ERROR_TIMEOUT
　タイムアウトが発生したため制御を戻します。

DB_ERROR_NOT_RELATIONSHIP_TO_DBRDATABASE
　DBRResultSet コンストラクタで指定された DBRDatabase オブジェクトと関連付けることができませんでした。一つの DBRDatabase オブジェクトに対して関連付けられるオブジェクトは 64 個までです。

DB_DRV_ERROR_INVALID_SQL_EXCLUSIVE
　WITH 句で始まる SQL 文の指定時には，DBResultSet オブジェクトの生成オプションを TYPE_EXCLUSIVE 以外に設定してください。

DB_DRV_ERROR_STAT_COUNT
　ステートメントの個数が上限値を超えました。

GetArraySize メソッド

機能

繰り返し列の定義された要素の数を取得します。

形式

フィールド番号で指定する場合

```
UINT32 GetArraySize(UINT32 ui32Index) throw DBSQLCA
```

フィールド名で指定する場合

```
UINT32 GetArraySize(LPCTSTR lpctFieldName) throw DBSQLCA
```

引数

ui32Index

フィールドの列番号を，1 ～フィールド数の範囲の値で指定します。

lpctFieldName

フィールドのフィールド名を指定します。

戻り値

データ型：UINT32

繰り返し列の定義された要素数を返します。

機能詳細

フィールド番号，又はフィールド名によって特定した繰り返し列の，要素数を取得します。

指定したフィールドが繰り返し列でない場合は，0 が返ります。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_BEFORE_EXECUTE
　Execute メソッドが実行されていません。

DB_ERROR_CANNOT_BE_NULL
　引数 lpctFieldName に NULL を指定しています。

DB_ERROR_NOT_FOUND
　インデックス指定の場合は，指定したフィールド番号が 1 未満の値，又はフィールド数より大きい値を指定しています。フィールド名指定の場合は, 指定したフィールド名が間違っています(NULL を除く)。

GetCurrent メソッド

機能

検索したすべてのレコードの先頭から数えた，カレントレコードの位置を取得します。

形式

```
UINT32 GetCurrent(void) throw DBSQLCA
```

引数

なし

戻り値

データ型：UINT32

カレントレコードの先頭レコードからの位置を取得します。

機能詳細

検索したすべてのレコードの先頭を 1 として数えた，カレントレコードの位置を取得します。

レコードの終端（EOF）で GetCurrent メソッドを呼び出すと，検索レコード数+1 の値を返します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_RESULTSET_NOT_EXIST
　非同期実行時に，検索結果がまだ取得できていません。

GetCurrentOfResultSet メソッド

機能

ResultSet の先頭レコードから数えた，カレントレコードの位置を取得します。

形式

```
UINT32 GetCurrentOfResultSet(void) throw DBSQLCA
```

引数

なし

戻り値

データ型：UINT32

カレントレコードの ResultSet の先頭レコードからの位置を取得します。

機能詳細

ResultSet のレコードの先頭を 1 として数えた，カレントレコードの位置を取得します。

ResultSet が 0 レコードの場合も 1 を返します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_RESULTSET_NOT_EXIST
　非同期実行時に，検索結果がまだ取得できていません。

GetErrorStatus メソッド

機能

SQL 文の非同期実行時のエラー情報を得るために，DBSQLCA オブジェクトへのポインタを取得します。

形式

```
DBSQLCA *GetErrorStatus(void)
```

引数

なし

戻り値

データ型：DBSQLCA*

DBSQLCA オブジェクトへのポインタ。

機能詳細

DBSQLCA オブジェクトへのポインタを取得します。

非同期処理のときは，エラーが発生しても直にエラーを取得できません。このため，ユーザは任意の時点でエラーを確認する必要があります。

非同期実行時にエラーが発生した場合，DBSQLCA オブジェクトにエラー情報が設定されます。

DBSQLCA オブジェクトでは，非同期実行中に発生したエラー情報を最大 100 回分保存できます。エラー情報は，新しいエラー情報から 100 回分保存されるため，エラー情報を確認したあとは DBSQLCA クラスの Delete メソッドを呼び出して，不要なエラー情報をクリアしておいてください。

発生する例外

DB_ERROR_BEFORE_EXECUTE
　EXECUTE メソッドが実行されていません。

GetField メソッド

機能

カレントレコードのフィールドの値を取得します。

形式１：ユーザ確保領域へコピーする場合

インデクス番号指定

```
void GetField(UINT32 dwIndex, INT16 * psData)  throw DBSQLCA
void GetField(UINT32 dwIndex, INT32 * plData)  throw DBSQLCA
void GetField(UINT32 dwIndex, UINT16* pswData) throw DBSQLCA
void GetField(UINT32 dwIndex, UINT32* pdwData) throw DBSQLCA
void GetField(UINT32 dwIndex, SINGLE* psfData) throw DBSQLCA
void GetField(UINT32 dwIndex, DOUBLE* pdfData) throw DBSQLCA
void GetField(UINT32 dwIndex, DBR_DATETIME* pdtData)
                                               throw DBSQLCA
void GetField(UINT32 ui32Index, DBRArrayDataConstPtr*
                                    cparData) throw DBSQLCA
```

フィールド名指定

```
void GetField(LPCTSTR lpctFieldName, INT16 * psData)
                                               throw DBSQLCA
void GetField(LPCTSTR lpctFieldName, INT32 * plData)
                                               throw DBSQLCA
void GetField(LPCTSTR lpctFieldName, UINT16* pswData)
                                               throw DBSQLCA
void GetField(LPCTSTR lpctFieldName, UINT32* pdwData)
                                               throw DBSQLCA
void GetField(LPCTSTR lpctFieldName, SINGLE* psfData)
                                               throw DBSQLCA
void GetField(LPCTSTR lpctFieldName, DOUBLE* pdfData)
                                               throw DBSQLCA
void GetField(LPCTSTR lpctFieldName, DBR_DATETIME* pdtData)
                                               throw DBSQLCA
void GetField(LPCTSTR lpctFieldName, DBRArrayDataConstPtr*
                                    cparData) throw DBSQLCA
```

形式２：DABroker が用意した領域を参照する場合

インデクス番号指定

```
void GetField(UINT32 dwIndex, LPTSTR* lptData) throw DBSQLCA
void GetField(UINT32 dwIndex, DBR_BINARY* pblobData)
                                               throw DBSQLCA
```

フィールド名指定

```
void GetField(LPCTSTR lpctFieldName, LPTSTR* lptData)
                                               throw DBSQLCA
void GetField(LPCTSTR lpctFieldName, DBR_BINARY* pblobData)
                                               throw DBSQLCA
```

形式３：ユーザ確保領域へコピーする場合

インデクス番号指定

```
UINT32 GetField(UINT32 dwIndex, LPTSTR lptData,
                                UINT32 dwSize) throw DBSQLCA
UINT32 GetField(UINT32 dwIndex, DBR_BINARY* pblobData,
                                UINT32 dwSize) throw DBSQLCA
```

フィールド名指定

```
UINT32 GetField(LPCTSTR lpctFieldName, LPTSTR lptData,
                                UINT32 dwSize) throw DBSQLCA
UINT32 GetField(LPCTSTR lpctFieldName, DBR_BINARY* pblobData,
                                UINT32 dwSize) throw DBSQLCA
```

引数

dwIndex, ui32Index

1 から始まるフィールドの番号を指定します。指定するフィールドの番号は，SQL 文中での出現順に割り当てられます。

lpctFieldName

フィールド名を指定します。

なお，SELECT 文でフィールド演算や，COUNT などの関数を使用した場合，フィールド名はデータベースによって決められます。このような場合にフィールド名を調べるためには，GetFieldName メソッドを呼び出します。取得するフィールド名は，重複することもあります。このため，SELECT 文で JOIN を指定したときなどに，フィールド名が重複している可能性がある場合や，SELECT 文でのフィールド演算などでフィールド名が不定の場合は，フィールドの番号を使用してください。

*xxData（形式１，３の第２引数）

データをコピーする領域のポインタを指定します。

*xxData（形式２の第２引数）

ポインタ変数を指定します。

dwSize

コピー先の領域サイズをバイト数で指定します。

戻り値

データ型：UINT32

コピーした文字列の長さが返ります。フィールドの値を指定された型にキャストして取得します（形式３の場合）。

機能詳細

形式１の処理

フィールド値を，引数に指定されたポインタが指す領域にコピーします。

形式２の処理

DABroker が確保した領域にフィールド値をコピーし，その先頭アドレスをポインタ変数に設定します。このため，LPTSTR 又は DBR_BINARY の変数の確保だけで，フィールド値をコピーする領域を確保する必要はありません。

- フィールド値の有効期間

  フィールド値は，Open メソッドや Next メソッドなどで次の ResultSet 取得時まで，又は ResultSet を削除するまで有効です。前の ResultSet のフィールド値を残したい場合には，ユーザが確保した領域にデータを取得してください。

形式３の処理

- LPTSTR 型

  引数 lptData の指す領域は，引数 dwSize バイト分確保されている必要があります。

  - 引数 dwSize がデータベースから取得したデータの長さ以下の場合

    データベースのデータ型が可変長文字列型の場合，引数 dwSize-1 分のデータ（最後には NULL 終端文字が入る）がコピーされ，残りのデータは切り捨てられます。

    データベースのデータ型が固定長文字列の場合，引数 dwSize 分のデータ（最後に NULL 終端文字が入らない）がコピーされ，残りのデータは切り捨てられます。

  - 引数 dwSize がデータベースから取得したデータの長さよりも大きい場合

    文字列はすべてコピーします。固定長文字列の場合，残りの領域には NULL 終端文字が埋められます。

- DBR_BINARY 型

  フィールド値をコピーする領域（LPTSTR 型）を確保し，DBR_BINARY 型変数（構造体）の Data メンバに設定し，この変数を GetField メソッドの引数に指定します。引数 dwSize はこの領域のサイズを指定します。

  引数 dwSize がデータベースから取得したデータの長さ以下の場合，引数 dwSize 分のデータ（最後に NULL 終端文字が入らない）がコピーされ，残りのデータは切り捨てられます。引数 dwSize がデータベースから取得したデータの長さより大きい場合，データはすべてコピーされ，残りの領域には NULL 終端文字が埋められます。

データ変換について

データベースのデータ型と GetField の引数に指定したデータ型とが異なる場合は，値の変換できるものについては引数で指定したデータ型に変換して返します。

文字列から数値データ型への変換に失敗した場合は 0 を返します。データ型の変換規則については，「7.1 クラスライブラリで扱うデータ型と変換規則」の GetField，GetParam メソッドでのデータ型変換規則を参照してください。

NULL 値の扱い

NULL であるフィールドに対して GetField メソッドを呼び出すと，戻される値は意味のない値(0，空文字列，要素がすべて 0，空文字列からなる構造体，又は不定値)となります。

このため，NULL が格納されている可能性のあるフィールドでは，IsFieldNull メソッドを呼び出して，NULL 値かどうかを確認してください。

DBR_BINARY 型で値を取得した場合

各メンバの値は次のようになります。

- SetMaxFieldSize メソッドで指定した長さよりも実データ長が長い場合
  Length：取得したデータ長
  RealLength：実データ長
  Data：Length の長さ分のデータ（データの切り捨てあり）
- SetMaxFieldSize メソッドで指定した長さよりも実データ長が短い，又は等しい場合
  Length：取得したデータ長
  RealLength：0
  Data：Length の長さ分のデータ（データの切り捨てなし）

TYPE_BLOB_FILE について

SetResultSetType メソッドで，TYPE_BLOB_FILE を指定した場合，BLOB 型のフィールドに対する GetField メソッドの引数にはファイル名称が返ります。この場合，GetField メソッドの引数の型には，文字列と同じ LPTSTR を使います。

繰り返し列の扱い

引数 cparData に繰り返し列のコピー先である DBArrayData オブジェクトへのポインタを指定します。DBRArrayData オブジェクトは参照専用で生成されるため，更新する場合は更に更新用のオブジェクトを生成する必要があります。

SQL/K の場合

データが設定されていない列を検索しても，GetField メソッドを実行すると，VOS K/FS のファイル定義で定義されたデータを返します。ファイル定義の詳細は，マニュアル「ファイル運用」を参照してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_RESULTSET_NOT_EXIST
　非同期実行時に，検索結果がまだ取得できていません。

DB_ERROR_NOT_FOUND
　引数 dwIndex の範囲が不正です。

DB_ERROR_CANNOT_BE_NULL
　引数 lpctFieldName，又は引数 xxData に NULL を指定しています。

DB_ERROR_DATA_TRUNCATED
　取得した値を指定した型に変換できません。

DB_DRV_ERROR_INVALID_ARGUMENT
　引数に指定したフィールド名は不正です。

DB_ERROR_CONVERT_ARRAY_TO_SCALAR
　引数 cparData で繰り返し列以外のフィールド値を取得しました，又は引数 cparData 以外の引数で繰り返し列の値を取得しようとしました。

GetFieldCount メソッド

機能

ResultSet のフィールド数を取得します。

形式

```
UINT32 GetFieldCount(void) throw DBSQLCA
```

引数

なし

戻り値

データ型：UINT32

ResultSet 中のフィールド数。

機能詳細

Execute メソッドで指定された SELECT 文の，検索結果中に含まれるフィールドの個数を取得します。

GetFieldCount メソッドを呼び出す前に，Execute メソッドを呼び出しておく必要があります。

発生する例外

DB_ERROR_BEFORE_EXECUTE
　Execute メソッドが実行されていません。

GetFieldCType メソッド

機能

指定したフィールドのデータ型を C++の型で取得します。

形式

インデクス番号で指定する場合

```
DBR_CTYPE GetFieldCType(UINT32  dwIndex) throw DBSQLCA
```

フィールド名で指定する場合

```
DBR_CTYPE GetFieldCType(LPCTSTR  lpctFielsName) throw DBSQLCA
```

引数

dwIndex

1 から始まるフィールドの番号を指定します。

lpctFieldName

フィールドの名称を指定します。

戻り値

データ型：DBR_CTYPE

指定されたフィールドの C++データ型を取得します。

COL_CTYPE_INT16：INT16

COL_CTYPE_INT32：INT32

COL_CTYPE_UINT16：UINT16

COL_CTYPE_UINT32：UINT32

COL_CTYPE_DOUBLE：DOUBLE

COL_CTYPE_SINGLE：SINGLE

COL_CTYPE_CHAR：TCHAR, LPTSTR, LPCTSTR

COL_CTYPE_OFFSET：OFFSET

COL_CTYPE_DATETIME：DBR_DATETIME

COL_CTYPE_BINARY：DBR_BINARY

COL_CTYPE_BOOLEAN：BOOLEAN

機能詳細

指定したフィールドのデータ型を C++のデータ型で取得します。

それぞれのデータ型は，ResultSet 中に読み込まれたときに，デフォルトの C++データ型に変換されます。

変換されたデフォルトのデータ型がどのような型になるかは，使用する DBMS によって異なります。型変換の詳細については，「7.1 クラスライブラリで扱うデータ型と変換規則」を参照してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_FOUND
　指定したフィールドがありません。

DB_ERROR_CANNOT_BE_NULL
　引数 lpctFieldName に NULL を指定できません。

DB_ERROR_BEFORE_EXECUTE
　Execute メソッドが実行されていません。

GetFieldDBType メソッド

機能

指定したフィールドのデータ型を DBMS の型で取得します。

形式

インデクス番号で指定した場合

```
UINT16 GetFieldDBType(UINT32   dwIndex) throw DBSQLCA
```

フィールド名で指定する場合

```
UINT16 GetFieldDBType(LPCTSTR lpctFieldName) throw DBSQLCA;
```

引数

dwIndex

1 から始まるフィールドの番号を指定します。

lpctFieldName

フィールドの名称を指定します。

戻り値

データ型：UINT16

指定したフィールドのデータ型を DBMS のデータ型で取得します。

機能詳細

指定したフィールドの属性を DBMS のデータ型で取得します。

それぞれのデータ型は，DBMS によって異なります。データ型の詳細については，使用する DBMS の仕様をご確認ください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_FOUND
　指定したフィールドがありません。

DB_ERROR_CANNOT_BE_NULL
　引数 lpctFieldName に NULL を指定できません。

DB_ERROR_BEFORE_EXECUTE
　Execute メソッドが実行されていません。

GetFieldName メソッド

機能

フィールド名称を取得します。

形式

```
LPCTSTR GetFieldName(UINT32   dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まるフィールドの番号を指定します。指定するフィールド番号は，SQL 文中での出現順に割り当てられます。

戻り値

データ型：LPCTSTR

指定されたフィールドの名称を取得します。

機能詳細

指定されたフィールドのフィールド名を取得します。

SELECT 文でフィールド演算や，COUNT などの関数を使用した場合，フィールド名はデータベースによって決められます。このような場合にこのメソッドを使用します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_FOUND
　指定したフィールドがありません。

DB_ERROR_BEFORE_EXECUTE
　Execute メソッドが実行されていません。

GetFieldPrecision メソッド

機能

フィールドの精度（桁数）を取得します。

形式

インデクス番号で指定する場合

```
UINT32  GetFieldPrecision(UINT32  dwIndex) throw DBSQLCA
```

フィールド名で指定する場合

```
UINT32  GetFieldPrecision(LPCTSTR lpctFieldName) throw DBSQLCA
```

引数

dwIndex

1 から始まるフィールドの番号を指定します。

lpctFieldName

フィールドの名称を指定します。

戻り値

データ型：UINT32

フィールドの精度（桁数）を取得します。COL_TYPE_NUMERIC の場合，小数点以下を含めた数値の有効桁数を取得します。

機能詳細

指定されたフィールドのデータの精度（桁数）を取得します。

指定されたフィールドのデータ型によって, 戻り値の持つ意味が異なるものがあります。各データ型で返す値（Precision,Scale）を，表 4-2 から 4-9 に示します。

表 4-2　Precision,Scale で返す値（ORACLE）

データ型	Precision	Scale
NUMBER(固定小数点数)	定義長	小数点以下の桁数
NUMBER（浮動小数点数）	45	0
VARCHAR2	定義長	0
RAW	定義長	0
LONG	0	0
LONG RAW	0	0
DATE	32※	0
ROWID	18	0

注※ この値は，sizeof(DBR_DATETIME)で取得した値です。

表 4-3　Precision,Scale で返す値（SQL Anywhere）

データ型	Precision	Scale
CHAR CHARACTER VARCHAR CHARACTER VARYING	定義長	0
SYSNAME	30	0
LONG VARCHAR TEXT	0	0
SMALLINT BIT TINYINT	6	0
INT	11	0

データ型	Precision	Scale
INTEGER	11	0
DECIMAL NUMERIC	定義長	定義長
MONEY	19	4
SMALLMONEY	10	4
REAL	9	0
DOUBLE	17	0
DATE	32※	0
TIME	32※	0
TIMESTAMP DATETIME SMALLDATETIME	32※	0
BINARY	定義長	0
LONG BINARY IMAGE	0	0

注※ この値は，sizeof(DBR_DATETIME)で取得した値です。

表 4-4　Precision,Scale で返す値(Adaptive Server Anywhere)

データ型	Precision	Scale
CHAR CHARACTER VARCHAR CHARACTER VARYING	定義長	0
SYSNAME	30	0
LONG VARCHAR TEXT	0	0
SMALLINT BIT TINYINT OLDBIT	6	0
INT INTEGER	11	0
DECIMAL NUMERIC	定義長	定義長
MONEY	19	4
SMALLMONEY	10	4

データ型	Precision	Scale
REAL	9	0
DOUBLE	17	0
DATE	32※	0
TIME	32※	0
TIMESTAMP DATETIME SMALLDATETIME	32※	0
BINARY VARBINARY	定義長	0
LONG BINARY IMAGE java serialization java.lang.Object	0	0
BIGINT	20	0

注※ この値は，sizeof(DBR_DATETIME)で取得した値です

表 4-5　Precision,Scale で返す値(SQL Server)

データ型	Precision	Scale
CHAR VARCHAR	定義長	0
TEXT	0	0
SMALLINT TINYINT BIT	6	0
INT	11	0
DECIMAL NUMERIC	定義長	定義長
MONEY	19	4
SMALLMONEY	10	4
REAL	9	0
DOUBLE	17	0
DATETIME SMALLDATETIME	32※	0
TIMESTAMP	8	0
BINARY	定義長	0

データ型	Precision	Scale
VARBINARY	定義長	0
IMAGE	0	0

注※ この値は，sizeof(DBR_DATETIME)で取得した値です

表 4-6　Precision,Scale で返す値（HiRDB）

データ型	Precision	Scale
SMALLINT	6	0
INT	11	0
SMALLFLT	9	0
FLOAT	17	0
DECIMAL	定義長	小数点以下の桁数
CHAR	定義長	0
VARCHAR	定義長	0
MCHAR	定義長	0
MVARCHAR	定義長	0
NCHAR	定義長×2	0
NVARCHAR	定義長×2	0
DATE	32※	0
TIME	32※	0
INTERVAL YEAR TO DAY	32※	0
INTERVAL HOUR TO SECOND	32※	0
BLOB	定義長	0
BINARY	定義長	0

注※ この値は，sizeof(DBR_DATETIME)で取得した値です。

表 4-7　Precision,Scale で返す値(XDM/RD)

データ型	Precision	Scale
DECIMAL	定義長	小数点以下の桁数
LARGE DECIMAL	定義長	小数点以下の桁数
INT	11	0
SMALLINT	6	0
FLOAT	17	0

データ型	Precision	Scale
SMALLFLT	9	0
CHAR	定義長	0
VARCHAR	定義長	0
LONG VARCHAR	定義長	0
MCHAR	定義長	0
MVARCHAR	定義長	0
LONG MVARCHAR	定義長	0
NCHAR	定義長×2	0
NVARCHAR	定義長×2	0
LONG NVARCHAR	定義長×2	0
DATE	32※	0

注※ この値は，sizeof(DBR_DATETIME)で取得した値です。

表 4–8　Precision,Scale で返す値（SQL/K）

データ型	Precision	Scale
SMALLINT	2	0
INTEGER	4	0
DECIMAL	定義長	定義長
NUMERIC TRAILING	定義長	定義長
NUMERIC UNSIGNED	定義長	定義長
CHAR	定義長	0
NCHAR	定義長×2	0
MCHAR	定義長	0
XCHAR	定義長	0

表 4–9　Precision,Scale で返す値（XDM/SD）

データ型	Precision	Scale
BIT	定義長	0
$DBK(データベースキー)	定義長	0
NUMERIC TRAILING	定義長	定義長
DECIMAL	定義長	定義長
INTEGER	4	0

データ型	Precision	Scale
SMALLFLT	2	0
NCHAR	定義長×2	0
CHAR	定義長	0

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_FOUND
　指定したフィールドがありません。

DB_ERROR_CANNOT_BE_NULL
　引数 lpctFieldName に NULL を指定しています。

DB_ERROR_BEFORE_EXECUTE
　Execute メソッドが実行されていません。

GetFieldScale メソッド

機能

フィールドの小数点以下の桁数を取得します。

形式

インデクス番号で指定する場合

```
INT32 GetFieldScale(UINT32 dwIndex) throw DBSQLCA
```

フィールド名で指定する場合

```
INT32 GetFieldScale(LPCTSTR lpctFieldName) throw DBSQLCA
```

引数

dwIndex

1 から始まるフィールドの番号を指定します。

lpctFieldName

フィールドの名称を指定します。

戻り値

データ型：INT32

COL_TYPE_NUMERIC の場合，小数点の以下の桁数を取得します。

0 の場合は整数となります。また，マイナスの場合は桁上がりとなります。

機能詳細

指定されたフィールドが数値データの場合，データの小数点以下の桁数を取得します。そのほかのデータの場合，返される値は意味を持ちません。

負の値が戻された場合，戻された値分の桁が上がります。

また，GetFieldPrecision メソッドで取得した値よりも大きな正の値が戻された場合，戻された値分の桁が下がります。

指定されたフィールドのデータ型によって，戻り値の持つ意味が異なるものがあります。各データ型で返す値（Scale）については，GetFieldPrecision メソッドの表 4-2 から 4-7 を参照してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_FOUND
　指定したフィールドがありません。

DB_ERROR_CANNOT_BE_NULL
　引数 lpctFieldName に NULL を指定しています。

DB_ERROR_BEFORE_EXECUTE
　Execute メソッドが実行されていません。

GetFieldType メソッド

機能

フィールドのデータ型を取得します。

形式

インデクス番号で指定する場合

```
UINT16 GetFieldType(UINT32 dwIndex) throw DBSQLCA
```

フィールド名で指定する場合

```
UINT16 GetFieldType(LPCTSTR lpctFieldName) throw DBSQLCA
```

引数

dwIndex

1 から始まるフィールドの番号を指定します。

lpctFieldName

フィールドの名称を指定します。

戻り値

データ型：UINT16

指定したフィールドのデータ型を取得します。

COL_TYPE_INT16：2 バイト長符号付き 2 進整数

COL_TYPE_INT32：4 バイト長符号付き 2 進整数

COL_TYPE_UINT16：2 バイト長符号なし 2 進整数

COL_TYPE_UINT32：4 バイト長符号なし 2 進整数

COL_TYPE_NUMERIC：符号付き 10 進数

COL_TYPE_SINGLE：4 バイト長浮動小数点数

COL_TYPE_DOUBLE：8 バイト長浮動小数点数

COL_TYPE_CHAR：固定長文字列

COL_TYPE_VARCHAR：可変長文字列（最大長指定あり）

COL_TYPE_LONGVARCHAR：可変長文字列（最大長指定なし）

COL_TYPE_DATE：日付型

COL_TYPE_TIME：時間型

COL_TYPE_TIMESTAMP：日付・時間型

COL_TYPE_INTERVAL：時間間隔型

COL_TYPE_BOOLEAN：ブーリアン型

COL_TYPE_BINARY：固定長バイナリ型

COL_TYPE_VARBINARY：可変長バイナリ型（最大長指定あり）

COL_TYPE_LONGVARBINARY：可変長バイナリ型（最大長指定なし）

COL_TYPE_MONEY：金額型

COL_TYPE_SERIAL：自動インクリメント型

COL_TYPE_ROWID：ROWID

機能詳細

指定されたフィールドのデータベース中でのデータ型を取得します。

それぞれのデータ型は，ResultSet 中に読み込まれたときに，デフォルトの C++データ型に変換されます。

変換されたデフォルトのデータ型がどのような型になるかは，使用する DBMS によって異なります。

型変換の詳細については，「7.1 クラスライブラリで扱うデータ型と変換規則」を参照してください。

繰り返し列かどうかの判断は，GetArraySize メソッドで確認してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_FOUND
　指定したフィールドがありません。

DB_ERROR_CANNOT_BE_NULL
　引数 lpctFieldName に NULL を指定しています。

DB_ERROR_BEFORE_EXECUTE
　Execute メソッドが実行されていません。

GetMaxRows メソッド

機能

SetMaxRows メソッドで指定したレコード数を取得します。

形式

```
UINT32 GetMaxRows(void)
```

引数

なし

戻り値

データ型：UINT32

SetMaxRows メソッドで設定した，レコード数の最大値を取得します。

機能詳細

SetMaxRows メソッドで設定した，ResultSet に読込まれるレコード数の最大値を取得します。
SetMaxRows メソッドを一度も呼び出していない場合はデフォルト値の 100 を返します。

発生する例外

なし

GetParam メソッド

機能

SetParam メソッドで指定したパラメタの値を取得します。

形式 1：ユーザ確保領域へコピーする場合

```
void GetParam(UINT32 dwIndex, INT16 * sParam) throw DBSQLCA
void GetParam(UINT32 dwIndex, INT32 * lParam) throw DBSQLCA
void GetParam(UINT32 dwIndex, UINT16 * swParam) throw DBSQLCA
void GetParam(UINT32 dwIndex, UINT32 * dwParam) throw DBSQLCA
void GetParam(UINT32 dwIndex, SINGLE * sfParam) throw DBSQLCA
void GetParam(UINT32 dwIndex, DOUBLE * dfParam) throw DBSQLCA
void GetParam(UINT32 dwIndex, DBR_DATETIME * dtParam)
```

```
                                                    throw DBSQLCA
void GetParam(UINT32 ui32Index, DBRArrayDataPtr* parParam)
                                                    throw DBSQLCA
```

形式２：DABroker が用意した領域を参照する場合

```
void GetParam(UINT32 dwIndex, LPTSTR * lptParam) throw DBSQLCA
void GetParam(UINT32 dwIndex, DBR_BINARY * blobParam)
                                                    throw DBSQLCA
```

形式３：ユーザ確保領域へコピーする場合

```
UINT32 GetParam(UINT32 dwIndex, LPTSTR lptParam, UINT32 dwSize)
                                                     throw DBSQLCA
UINT32 GetParam(UINT32 dwIndex, DBR_BINARY* blobParam,
                                  UINT32 dwSize) throw DBSQLCA
```

引数

dwIndex，ui32Index

1 から始まるパラメタの番号を指定します。指定するパラメタの番号は，SQL 文中での出現順に割り当てられます。

*xxParam（形式１，３の第２引数）

データをコピーする領域のポインタを指定します。

*xxParam（形式２の第２引数）

ポインタ変数を指定します。

dwSize

コピー先の領域サイズをバイト数で指定します。

戻り値

データ型：UINT32

コピーした文字列の長さが返ります。パラメタの値を指定された型にキャストして取得します。

機能詳細

SetParam で設定した値を取得します。このメソッドは，Execute を呼び出し，SetParam メソッドでパラメタを渡した後に実行してください。SetParam を実行する前に GetParam を実行した場合, NULL 文字, 又は 0 が返ります。

形式１の処理

パラメタ値を，引数に指定されたポインタが指す領域にコピーします。

形式２の処理

DABroker が確保した領域にパラメタ値をコピーし，その先頭アドレスをポインタ変数に設定します。このため，LPTSTR 又は DBR_BINARY の変数の確保だけで，パラメタ値をコピーする領域を確保する必要はありません。

- パラメタ値の有効期間

パラメタ値は，Execute メソッドや SetParam メソッドを実行するまで有効です。

形式３の処理

- LPTSTR 型
  引数 lptParam の指す領域は，引数 dwSize バイト分確保されている必要があります。引数 dwSize がデータベースから取得したデータの長さ以下の場合，引数 dwSize-1 分のデータ（最後には NULL 終端文字が入る）がコピーされ，残りのデータは切り捨てられます。
  引数 dwSize がデータベースから取得したデータの長さよりも大きい場合，文字列はすべてコピーします。
- DBR_BINARY 型
  パラメタ値をコピーする領域（LPTSTR 型）を確保し，DBR_BINARY 型変数（構造体）の Data メンバに設定し，この変数を GetField メソッドの引数に指定します。引数 dwSize はこの領域のサイズを指定します。
  引数 dwSize がデータベースから取得したデータの長さ以下の場合，引数 dwSize 分のデータがコピーされ，残りのデータは切り捨てられます。この場合，NULL 終端文字は入りません。引数 dwSize がデータベースから取得したデータの長さより大きい場合，データはすべてコピーされ，残りの領域には NULL 終端文字が埋められます。

データ変換について

データベースのデータ型と GetParam の引数に指定したデータ型とが異なる場合は，値の変換できるものについては引数で指定したデータ型に変換して返します。

文字列から数値データ型への変換に失敗した場合は 0 を返します。データ型の変換規則については，「7.1 クラスライブラリで扱うデータ型と変換規則」の GetField，GetParam メソッドでのデータ型変換規則を参照してください。

NULL 値の扱い

NULL であるパラメタに対して GetParam メソッドを呼び出すと，戻される値は意味のない値(0，空文字列，要素がすべて 0，空文字列からなる構造体，又は不定値)となります。

このため，パラメタの値が NULL である可能性がある場合は，IsNull メソッドを呼び出して，NULL 値かどうかを確認してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数 dwRow 又は dwIndex の範囲が不正です。(Execute 実行前に実行した場合含む)

DB_ERROR_DATA_TRUNCATED
　パラメタの値を指定した型に変換できません。

DB_ERROR_BEFORE_EXECUTE
　Execute メソッドが実行されていません。

GetParamCount メソッド

機能

パラメタの数を取得します。

形式

```
UINT32 GetParamCount(void) throw DBSQLCA
```

引数

なし

戻り値

データ型：UINT32

パラメタの個数。

機能詳細

SELECT 文中に含まれる？パラメタの個数を取得します。

GetParamCount メソッドを呼び出す前に，Execute メソッドを呼び出しておく必要があります。

発生する例外

DB_ERROR_BEFORE_EXECUTE
　EXECUTE メソッドが実行されていません。

GetRowCount メソッド

機能

ResultSet に読込まれたレコード数を取得します。

形式

```
UINT32 GetRowCount(void) throw DBSQLCA
```

引数

なし

戻り値

データ型：UINT32

ResultSet に読込まれたレコード数が戻されます。

機能詳細

ResultSet に読み込まれたレコード数を取得します。

GetRowCount メソッドの返す値が GetMaxRows メソッドの返す値よりも少ない場合，データベース中のすべての検索結果が読み込まれたことを示します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_RESULTSET_NOT_EXIST
　非同期実行時に，検索結果がまだ取得できていません。

InExecute メソッド

機能

DBRResultSet オブジェクトに非同期実行中（又は実行待ち）のステートメントがあるかどうかを確認します。

形式

```
BOOLEAN  InExecute(void) throw DBSQLCA
```

引数

なし

戻り値

データ型：BOOLEAN

TRUE：ステートメントが非同期実行中(又は実行待ち)です。

FALSE：非同期実行中(又は実行待ち)ではありません。

機能詳細

InExecute メソッドを実行したオブジェクト内のステートメントが非同期実行中(又は実行待ち）かどうかを確認します。

非同期実行中(又は実行待ち）の場合は TRUE を，非同期実行中(又は実行待ち）でない場合は FALSE を返します。

同期実行接続時は FALSE を返します。

コネクション内のすべてのステートメントを対象に非同期実行中かどうかを確認するには，InWaitForDataSource メソッドを呼び出します(DBRDatabase クラス)。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_BEFORE_EXECUTE
　Execute メソッドが実行されていません。

IsEOF メソッド

機能

カーソルの位置が，ResultSet のレコードの最後を超えたかどうかを確認します。

形式

```
BOOLEAN  IsEOF(void) throw DBSQLCA
```

引数

なし

戻り値

データ型：BOOLEAN

TRUE：カーソルの位置が，ResultSet のレコードの最後を超えました。

FALSE：カーソルの位置は，まだ ResultSet の途中のレコードにあります。

機能詳細

Next メソッドで次のレコードへカーソルを移動した後，まだレコードがあるかどうかを判断するときに使います。TRUE が戻された場合，Next,Previous,Top,Bottom,Absolute,Relative メソッド又は PageNext メソッドを呼ぶとエラーになります。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_RESULTSET_NOT_EXIST
　非同期実行時に，検索結果がまだ取得できていません。

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

IsFieldNull メソッド

機能

フィールドの値が NULL かどうかを確認します。

形式

インデクス番号で指定する場合

```
BOOLEAN  IsFieldNull(UINT32   dwIndex) throw DBSQLCA
```

フィールド名称で指定する場合

```
BOOLEAN  IsFieldNull(LPCTSTR lpctName) throw DBSQLCA
```

引数

dwIndex

1 から始まるフィールドの番号を指定します。

lpctName

フィールド名を指定します。

戻り値

データ型：BOOLEAN

TRUE：フィールドの値が NULL です。

FALSE：フィールドの値は NULL ではありません。

SQL/K の場合，データが設定されていない列を検索しても，戻り値は常に FALSE になります。

機能詳細

指定されたフィールドの値が NULL 値かどうかを確認します。

NULL は C 言語で使用する NULL ポインタの意味ではなく，データベースシステムでは「値がない」ことを意味します。

NULL であるフィールドに対して GetField メソッドを呼び出すと，戻される値は意味のない値（0，NULL 文字列，要素がすべて 0，NULL 文字列からなる構造体，又は不定値）となるため注意してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_RESULTSET_NOT_EXIST
　非同期実行時に，検索結果がまだ取得できていません。

DB_ERROR_NOT_FOUND
　引数 dwIndex の範囲が不正です。

DB_ERROR_CANNOT_BE_NULL
　引数 lpctName に NULL を指定しています。

DB_DRV_ERROR_INVALID_ARGUMENT
　引数に指定したフィールド名は不正です。

IsParamNull メソッド

機能

パラメタの値が NULL かどうかを確認します。

形式

インデクス番号で指定する場合

```
BOOLEAN  IsParamNull(UINT32  dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まるフィールドの番号を指定します。

戻り値

データ型：BOOLEAN

TRUE：フィールドの値が NULL です。

FALSE：フィールドの値は NULL ではありません。

機能詳細

指定されたパラメタの値が NULL 値かどうかを確認します。

NULL は C 言語で使用する NULL ポインタの意味ではなく，データベースシステムでは「値がない」ことを意味します。

NULL であるパラメタに対して GetField メソッドを呼び出すと，戻される値は意味のない値（0，NULL 文字列，要素がすべて 0，NULL 文字列からなる構造体，又は不定値）となるため注意してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_RESULTSET_NOT_EXIST
　非同期実行時に，検索結果がまだ取得できていません。

DB_ERROR_NOT_FOUND
　引数 dwIndex の範囲が不正です。

DB_ERROR_CANNOT_BE_NULL
　引数 lpctName に NULL を指定しています。

DB_ERROR_BEFORE__EXECUTE
　Execute メソッドが実行されていません。

Next メソッド

機能

カーソルを次のレコードへ移動します。

形式

```
void Next(void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

次のレコードへカーソルを移します。

更新可能な ResultSet の場合, 検索条件に一致する次のレコードを ResultSet に読み込みます。参照専用の ResultSet の場合，カーソルを次のレコードへ移します。ResultSet の最後のレコードにカーソルがある場合，Next メソッドを呼び出すと SetMaxRows メソッドで指定した数分の次のレコードを読み込みます。

次のレコードを ResultSet に読み込む場合は, 非同期実行可能です。単にカーソルを次のレコードに移す場合は，非同期に実行しません。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_RESULTSET_NOT_EXIST
　非同期実行時に，検索結果がまだ取得できていません。

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

DB_ERROR_OUT_OF_RANGE
　レコード終端を超えてメソッドが呼び出されました。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_ERROR_DAB_ACCESS_ERROR
　DABroker でエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR
　システムコールでエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_TIMEOUT
　タイムアウトが発生したため制御を戻します。

OnBeforeRefresh メソッド

機能

Open メソッドを呼び出した場合はレコードが最初に読み込まれる前に，Refresh メソッドを呼び出した場合はレコードが読み込まれる前に呼び出される仮想関数です。

形式

```
virtual void OnBeforeRefresh(BOOLEAN bRefresh)
```

引数

bRefresh

呼び出し元が Refresh メソッドの場合は引数 bRefresh に TRUE が指定されて呼び出されます。呼び出し元が Open メソッドの場合は引数 bRefresh に FALSE が指定されて呼び出されます。

戻り値

なし

機能詳細

Open メソッドを呼び出した場合はレコードが最初に読み込まれる前に，Refresh メソッドを呼び出した場合はレコードが読み込まれる前に呼び出されます。そのため，アプリケーションから直接このメソッドは呼び出せません。提供されている OnBeforeRefresh メソッドには，呼ばれたときの処理が定義されていないので，実際に呼ばれる事象が発生しても何もしません。呼ばれたときにユーザ固有の処理を実行させる場合，DBRResultSet クラスの派生クラスを生成し，そこに OnBeforeRefresh 仮想関数のソースコードを定義します。

発生する例外

なし

OnEndRecord メソッド

機能

カーソルが ResultSet の最後のレコードを超えたときに呼び出される仮想関数です。

形式

```
virtual void OnEndRecord(BOOLEAN bEndRecord)
```

引数

bEndRecord

カレントレコードを移動して IsEOF が TRUE になったとき，先頭と最後のうち，最後のレコードを超えた場合に，引数 bEndRecord に TRUE が指定されて呼び出されます。

戻り値

なし

機能詳細

検索したレコードをすべて読み込んだ後，Next メソッドを呼び出したときに呼び出される仮想関数です。そのため，アプリケーションからはこのメソッドを直接呼び出せません。提供されている OnEndRecord 仮想関数には呼ばれたときの処理が定義されていませんので, 実際に呼ばれる事象が発生しても何もしません。呼ばれる事象が発生したときにユーザ固有の処理を実行させる場合, DBRResultSet の派生クラスを生成し，そこに OnEndRecord 仮想関数のソースコードを定義します。

発生する例外

なし

OnMoveRecord メソッド

機能

カーソルを移動したときに呼び出される仮想関数です。

形式

```
virtual void OnMoveRecord(BOOLEAN bSaveAndValidate)
```

引数

bSaveAndValidate

Open メソッドを呼び出して検索結果を ResultSet に初めて読み込む場合，Next メソッドなどのカーソル移動系のメソッドを呼び出した場合は, 引数 bSaveAndValidate に FALSE が指定されて呼び出されます。また，Update メソッドを呼び出してデータを更新する場合は，引数 bSaveAndValidate に TRUE が指定されて呼び出されます。

戻り値

なし

機能詳細

レコードを読み込んだり，更新したりする場合に呼び出されます。そのため，アプリケーションから直接この仮想関数を呼び出せません。提供されている OnMoveRecord 仮想関数には呼ばれたときの処理が定義されていませんので, 実際に呼ばれる事象が発生しても何もしません。呼ばれる事象が発生したきにユーザ固有の処理を実行させる場合，DBRResultSet の派生クラスを生成しそこに OnMoveRecord 関数のソースコードを定義します。

TRUE,FALSE の指定により，仮想関数実行時は, アプリケーションが検索レコードを参照するのか更新するのかを判断できます。

発生する例外

なし

Open メソッド

機能

検索結果を ResultSet に読み込みます。

形式

```
void Open(void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

Execute メソッドで指定した SELECT 文に対する検索結果を，ResultSet に読み込みます。

ResultSet に読み込める最大レコードの数は，SetMaxRows メソッドで指定した数です。次の ResultSet を読み込むためには，PageNext メソッドを呼び出してください。ResultSet は，次の Execute メソッドを呼出したとき，Close メソッドを呼び出したとき，親の Database オブジェクトが削除されたときまで有効です。

このメソッドは非同期実行接続時に非同期に実行するメソッドです。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_BEFORE_EXECUTE
　Execute メソッドが実行されていません。

DB_ERROR_DRIVER_ERROR
　DBMS でエラー発生が発生しました。

DB_ERROR_SELECT_NOT_EXECUTED
　SELECT 文が実行されていません。

DB_ERROR_DAB_ACCESS_ERROR
　DABroker でエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR
　システムコールでエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_DRV_ERROR_RDA_PARAM_TYPE
　パラメタの型が未設定か又は設定された型が不正です。

DB_ERROR_TIMEOUT
　タイムアウトが発生したため制御を戻します。

PageNext メソッド

機能

カーソルを次の ResultSet の先頭へ移動します。

形式

```
void Previous(void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

次のレコード群（SetMaxRows メソッドで設定したレコード数）を ResultSet に読込みます。カーソルは次のレコード群の先頭レコードに位置付けられます。

すべてのレコードが読み込まれている（IsEOF メソッドで TRUE が返る場合)ときにこのメソッドは呼び出せません。

このメソッドは非同期実行接続時に非同期に実行するメソッドです。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_RESULTSET_NOT_EXIST
　非同期実行時に，検索結果がまだ取得できていません。

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

DB_ERROR_OUT_OF_RANGE
　ResultSet の範囲を超えるような呼び出しです。

DB_ERROR_DRIVER_ERROR
　DBMS でエラー発生しました。

DB_ERROR_DAB_ACCESS_ERROR
　DABroker でエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR
　システムコールでエラーが発生しました。

DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_ILLEGAL_VALUE
指定した引数が不正です。
DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_TIMEOUT
タイムアウトが発生したため制御を戻します。

Previous メソッド

機能

カーソルを一つ前のレコードへ移動します。

形式

```
void Previous(void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

カーソルを一つ前のレコードへ移します。

ResultSet の先頭で Previous メソッドは呼び出せません。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_RESULTSET_NOT_EXIST
非同期実行時に，検索結果がまだ取得できていません。

DB_ERROR_IN_ASYNC_EXECUTE
ステートメントが非同期実行処理中です。

DB_ERROR_OUT_OF_RESULTSET
ResultSet の範囲を超えるような呼び出しです。

Refresh メソッド

機能

先頭のレコードから再読み込みをします。

形式

```
void Refresh(void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

データベースからレコードを再検索して，先頭レコードから再読み込みをします。

実行する SELECT 文が？パラメタを持つ場合，Refresh メソッドを呼び出す前に SetParam メソッドを呼び出して？パラメタの値を変更できます。この場合，Refresh メソッドによって再検索された結果は，新しく設定されたパラメタの値を反映したものになります。

例えば「商品番号」を？パラメタとして使用している場合，Refresh メソッドを呼び出す前に SetParam メソッドで新しい商品番号を設定すると，Refresh メソッドを呼び出して再検索されたレコードは，新しい商品番号が反映された検索結果となります。

このメソッドは非同期実行可能なメソッドです。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_RESULTSET_NOT_EXIST
　非同期実行時に，検索結果がまだ取得できていません。

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_ERROR_DAB_ACCESS_ERROR
　DABroker でエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR
　システムコールでエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_TIMEOUT
　タイムアウトが発生したため制御を戻します。

Relative メソッド

機能

カーソルを現在のレコードの位置から n 個分移動します。

形式

```
void Relative(INT32  lCount) throw DBSQLCA
```

引数

lCount

指定した数分現在のレコードの位置からカーソルを移動します。

戻り値

なし

機能詳細

現在の位置から数えて，引数 lCount 番目のレコードにカーソルを移します。

正の値を指定した場合現在の位置より指定した値分後ろのレコードへ,負の値を指定した場合は指定した値分前のレコードへカーソルを移動します。

すべてのレコードが読み込まれている（IsEOF メソッドで TRUE が返る）ときにこのメソッドは呼び出せません。

また，引数 lCount に検索結果の範囲を超えるような値は指定できません。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_RESULTSET_NOT_EXIST
　非同期実行時に，検索結果がまだ取得できていません。

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

DB_ERROR_OUT_OF_RESULTSET
　ResultSet の範囲を超えるような呼び出しです。

SetField メソッド

機能

指定されたフィールドにデータを設定します。

形式

インデクスで指定する場合

```
void SetField(UINT32 dwIndex, INT16 sData) throw DBSQLCA
void SetField(UINT32 dwIndex, INT32 lData) throw DBSQLCA
void SetField(UINT32 dwIndex, UINT16 swData) throw DBSQLCA
void SetField(UINT32 dwIndex, UINT32 dwData) throw DBSQLCA
void SetField(UINT32 dwIndex, SINGLE sfData) throw DBSQLCA
void SetField(UINT32 dwIndex, DOUBLE dfData) throw DBSQLCA
void SetField(UINT32 dwIndex, LPCTSTR lpctData) throw DBSQLCA
void SetField(UINT32 dwIndex, const DBR_DATETIME& dtData)
                                              throw DBSQLCA
void SetField(UINT32 dwIndex, const DBR_BINARY& blobData)
```

```
                                                throw DBSQLCA
void SetField(UINT32 ui32Index, const DBRArrayDataConstPtr&
                                      cparData) throw DBSQLCA
```

フィールド名で指定する場合

```
void SetField(LPCTSTR lpctFieldName, INT16 sData) throw DBSQLCA
void SetField(LPCTSTR lpctFieldName, INT32 lData) throw DBSQLCA
void SetField(LPCTSTR lpctFieldName, UINT16 swData)
                                                throw DBSQLCA
void SetField(LPCTSTR lpctFieldName, UINT32 dwData)
                                                throw DBSQLCA
void SetField(LPCTSTR lpctFieldName, SINGLE sfData)
                                                throw DBSQLCA
void SetField(LPCTSTR lpctFieldName, DOUBLE dfData)
                                                throw DBSQLCA
void SetField(LPCTSTR lpctFieldName, LPCTSTR lpctData)
                                                throw DBSQLCA
void SetField(LPCTSTR lpctFieldName, const DBR_DATETIME& dtData)
                                                throw DBSQLCA
void SetField(LPCTSTR lpctFieldName, const DBR_BINARY& blobData)
                                                throw DBSQLCA
void SetField(LPCTSTR lpctFieldName, const DBRArrayDataConstPtr&
                                      cparData) throw DBSQLCA
```

引数

dwIndex, ui32Index

1 から始まるフィールドの番号を指定します。指定するフィールドの番号は，SQL 文中での出現順に割り当てられます。

lpctFieldName

フィールド名を指定します。

xxData（形式の第 2 引数）

設定する値を指定します。

戻り値

なし

機能詳細

カレントレコードのフィールドに値を設定します。

更新可能な ResultSet に対して指定できます。Edit メソッドを呼び出した後に実行してください。

値はデータベースのデータ型に合わせて変換されます。データ型の変換規則については，「7.1 クラスライブラリで扱うデータ型と変換規則」の SetField メソッドでのデータ型変換規則を参照してください。

BLOB 型データをファイル経由で設定する場合

Execute メソッドで TYPE_BLOB_FILE を指定した場合，BLOB 型のフィールドに対する SetField メソッドの引数にはファイル名称を指定します。この場合，SetField メソッドの引数の型には，文字列と同じ LPCTSTR を使います。

DBR_BINARY 型を指定する場合

DBR_BINARY 型の各メンバには次の値を設定します。

Length：設定するデータの長さ（単位はバイト）

RealLength：設定不要

Data：実際のデータの領域を指すポインタ

繰り返し列を扱う場合

引数 cparData に DBArrayData オブジェクトを指定します。DBRArrayData オブジェクトは DBRArrayDataFactory クラスを利用して事前に生成しておく必要があります。なお，これ以外の引数には，繰り返し列は指定できません。

また，引数 cparData に指定されたオブジェクトが，DBRArrayData クラスのインスタンスを持っていない場合は，エラーになります。

SetField メソッドで値を設定した後，DBRArrayData オブジェクトの SetData メソッドを実行すると，SetField メソッド実行時点のデータは失われて，SetData メソッドで設定したデータに置き換わるため注意が必要です。

XDM/RD で更新できる型の制限

LONG VARCHAR 型，LONG NVARCHAR 型，LONG MVARCHAR 型のフィールドについては更新できません。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_NOT_FOUND
　引数 dwIndex の範囲，又は引数 lpctFieldName が不正です。

DB_ERROR_CANNOT_BE_NULL
　引数 lpctFieldName に NULL を指定しています。

DB_ERROR_NOT_IN_EDIT
　Edit メソッドが実行されていません。

DB_ERROR_DATA_TRUNCATED
　取得した値を指定した型に変換できません。

DB_ERROR_RESULTSET_NOT_EXIST
　非同期実行時に，検索結果がまだ取得できていません。

DB_DRV_ERROR_INVALID_ARGUMENT
　引数に指定したフィールド名は不正です。

DB_ERROR_NO_INSTANCE
　引数 cparData に指定されたオブジェクトが，DBRArrayData クラスのインスタンスを持っていません。

DB_ERROR_CONVERT_ARRAY_TO_SCALAR
　引数 cparData に繰り返し列以外のフィールド値を設定しようとしました，又は引数 cparData 以外の引数に繰り返し列の値を設定しようとしました。

DB_ERROR_CANNOT_ACCESS_WHILE_EXECUTED
　Update メソッドを実行中に，DBRArrayData オブジェクトの SetData メソッドを呼び出しました。

SetFieldNull メソッド

機能

指定されたフィールドに NULL を設定します。

形式

インデクス番号で指定する場合

```
void SetNull(UINT32 dwIndex) throw DBSQLCA
```

フィールド名で指定する場合

```
void SetNull(LPCTSTR lpctFieldName) throw DBSQLCA
```

引数

dwIndex

1 から始まるフィールドの番号を指定します。

lpctFieldName

フィールド名を指定します。

戻り値

なし

機能詳細

指定されたフィールドに NULL を設定します。

NULL は C 言語で使用する NULL ポインタの意味ではなく，データベースシステムでは「値がない」ことを意味します。

SetField メソッドと同様に Edit メソッドを呼び出した後に実行してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_FOUND
　引数 dwIndex の範囲が不正です。
DB_ERROR_CANNOT_BE_NULL
　引数 lpctFieldName に NULL を指定しています。
DB_ERROR_NOT_IN_EDIT
　Edit メソッドが実行されていません。
DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

DB_ERROR_RESULTSET_NOT_EXIST
　非同期実行時に，検索結果がまだ取得できていません。
DB_DRV_ERROR_INVALID_ARGUMENT
　引数に指定したフィールド名が不正です。

SetMaxRows メソッド

機能

ResultSet に読込むレコード数の最大値を指定します。

形式

```
void SetMaxRows(UINT32 dwMaxSize=MAX_ROWS_DEFAULT) throw DBSQLCA
```

引数

dwMaxSize

ResultSet に読込むレコード数の最大値を指定します。必ず 1 以上を指定してください。

システムデフォルト値は，MAX_ROWS_DEFAULT です(=100 が仮定されます)。

- SQL/K，又は XDM/SD の場合，指定できる範囲は 4096 レコード以下です。

戻り値

なし

機能詳細

一度の読み込み（Execute メソッド，又は PageNext メソッドの呼び出し）で，ResultSet に読込むレコード数の最大値を設定します。

指定できるレコード数の範囲

指定できるレコード数の最大値は使用している DBMS によって異なります。各 DBMS ごとに，指定できるレコード数の最大値を次に示します。

- ORACLE を使用している場合：32767 レコード以下
- ORACLE 以外の DBMS を使用している場合：4096 レコード以下

1 レコードのフィールド数は，GetFieldCount メソッドを呼び出して取得できます。

更新可能な ResultSet での扱い

更新可能な ResultSet では(Execute メソッドの引数 swType で TYPE_EXCLUSIVE を指定した場合)，SetMaxRows メソッドによる指定は無視され，常に 1 が仮定されます。更新可能な ResultSet は常に 1 レコードだけを読み込みます。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数 dwMaxSize が 1 より小さい値です。

SetParam メソッド

機能

?パラメタに値を設定します。

形式

```
void SetParam(UINT32 dwIndex, LPCTSTR lpctParam) throw DBSQLCA
void SetParam(UINT32 dwIndex, INT16 sParam) throw DBSQLCA
void SetParam(UINT32 dwIndex, INT32 lParam) throw DBSQLCA
void SetParam(UINT32 dwIndex, UINT16 swParam) throw DBSQLCA
void SetParam(UINT32 dwIndex, UINT32 dwParam) throw DBSQLCA
void SetParam(UINT32 dwIndex, SINGLE sfParam) throw DBSQLCA
void SetParam(UINT32 dwIndex, DOUBLE dfParam) throw DBSQLCA
void SetParam(UINT32 dwIndex, const DBR_DATETIME& dtParam)
                                              throw DBSQLCA
void SetParam(UINT32 dwIndex, const DBR_BINARY& blobParam)
                                              throw DBSQLCA
void SetParam(UINT32 ui32Index, const DBRArrayDataConstPtr&
                                    cpArray) throw DBSQLCA
```

引数

指定されたパラメタを，データベースの型にキャストして設定します。

dwIndex，ui32Index

1 から始まるパラメタの番号を指定します。指定するパラメタの番号は，SQL 文中での出現順に割り当てられます。

xxParam（データ型：第 2 引数を参照してください）

パラメタに設定する値を指定します。

cpArray

インスタンスを保持している DBRArrayDataConstPtr オブジェクトを指定します。

戻り値

なし

機能詳細

指定されたパラメタの値を設定します。

値は指定するデータ型からデータベース固有の型に変換されます。データ型変換規則については，「7.1 クラスライブラリで扱うデータ型と変換規則」の SetParam メソッドでのデータ型変換規則を参照してください。

SetParam メソッドは，Execute メソッドを実行した後，Open メソッドまでの間に実行してください。

パラメタに NULL を設定したい場合は，SetNull メソッドを使用してください。

引数 lpctParam が NULL（LPCTSTR 型の変数に NULL を代入して引数に渡した場合，又は NULL を LPCTSTR 型にキャストした場合）の時も NULL が設定されます。キャストしないで引数に NULL だけ指定した場合は NULL とはみなしません。

BLOB 型データをファイル経由で設定する場合

Execute メソッドで TYPE_BLOB_FILE を指定した場合，BLOB 型のフィールドに対する SetParam メソッドの引数にはファイル名称を指定します。SetParamType メソッドは SetParam メソッドの前に実行してください。

DBR_BINARY 型を指定する場合

DBR_BINARY 型の各メンバには次の値を設定します。

Length：設定するデータの長さ（単位はバイト）

RealLength：設定不要

Data：実際のデータの領域を指すポインタ

繰り返し列を扱う場合の注意

引数 cpArray で指定する DBRArrayDataConstPtr オブジェクトがインスタンスを保持していない場合は，エラーになります。インスタンスを保持する DBRArrayDataPtr オブジェクトは，次の手順で生成し，引数へ設定します。

1. DBRArrayDataFactory オブジェクトの CreateArrayData メソッドで，DBRArrayData オブジェクトを生成します。
2. 新しく繰り返し列を作成する場合だけ, DBRArrayData オブジェクトの Create メソッドで要素の格納領域を確保します。
3. DBRArrayData オブジェクトの SetData メソッドで要素の値を設定します。
4. 引数 pArray に DBRArrayDataPtr オブジェクトを設定し，SetParam メソッドを呼び出します。
5. ExecuteUpdate メソッドで実行します。

なお，SetParam メソッドで値を設定した後に，DBRArrayData オブジェクトの SetData メソッドを実行すると，SetParam メソッド実行時点のデータは失われ，SetData メソッドに指定した値が設定されるため注意してください。

また，Update メソッドを実行中に，DBRArrayData オブジェクトの SetData メソッドは呼び出せません。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数 dwIndex の範囲が不正です（Execute 実行前に実行した場合を含みます）。

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

DB_ERROR_BEFORE_EXECUTE
　Execute メソッドが実行されていません。

DB_ERROR_NO_INSTANCE
　引数 cparArray に指定されたオブジェクトが, DBRArrayDataPtr クラスのインスタンスを持っていません。

DB_ERROR_CANNOT_ACCESS_WHILE_EXECUTED
　Update メソッドを実行中に，DBRArrayData オブジェクトの SetData メソッドを呼び出しました。

SetParamNull メソッド

機能

指定したパラメタに NULL を設定します。

形式

```
void SetParamNull(UINT32 uiIndex) throw DBSQLCA
```

引数

uiIndex

1 から始まるパラメタの番号を指定します。

戻り値

なし

機能詳細

指定されたパラメタに NULL を設定します。

NULL は C 言語で使用する NULL ポインタの意味ではなく，データベースシステムでは「値がない」ことを意味します。

SetParam メソッドと同様に，Execute メソッドを実行した後に実行してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_FOUND
　引数 dwIndex の範囲が不正です。

DB_ERROR_NOT_IN_EDIT
　Edit メソッドが実行されていません。
DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。
DB_ERROR_BEFORE_EXECUTE
　Execute メソッドが実行されていません。

SetParamType メソッド

機能

パラメタの属性を指定します。

形式

```
void SetParamType (UINT32 dwIndex,
                   UINT16 ui16Type = TYPE_BLOB_MEMORY) throw DBSQLCA
```

引数

dwIndex

1 から始まるパラメタの番号を指定します。

ui16Type

パラメタの属性を指定します。 指定できる値を次に示します。

- TYPE_BLOB_MEMORY：メモリ上のデータを長大データとして読み込みます。
- TYPE_BLOB_FILE_TEXT：テキストファイルを長大データとして読み込みます。
- TYPE_BLOB_FILE_BINARY：バイナリファイルを長大データとして読み込みます。

RDA Link for Gateway を使用している場合にだけ指定できる値を次に示します。

- RDA_DT_ROWID
- RDA_DT_XCHAR
- RDA_DT_NUM_TRAILING
- RDA_DT_NUM_UNSIGNED
- RDA_DT_FLOAT
- RDA_DT_SMALLFLT
- RDA_DT_DECIMAL
- RDA_DT_LARGE_DECIMAL
- RDA_DT_INTEGER
- RDA_DT_SMALLINT
- RDA_DT_MVARCHAR
- RDA_DT_MCHAR
- RDA_DT_LONG_MVARCHAR

- RDA_DT_NVARCHAR
- RDA_DT_NCHAR
- RDA_DT_LONG_NVARCHAR
- RDA_DT_VARCHAR
- RDA_DT_CHAR
- RDA_DT_LONG_VARCHAR
- RDA_DT_DATE

～LONG_(M,N)VARCHAR はメモリ上のデータを長大データとして読み込みます。

戻り値

なし

機能詳細

パラメタの属性を指定します。

BLOB 型のフィールドに対してファイル経由でのパラメタ設定をしたい場合だけ，SetParamType メソッドを TYPE_BLOB_FILE_TEXT，又は TYPE_BLOB_FILE_BINARY オプションを指定して呼び出します。この場合，SetParam では「値」として「ファイル名」を指定する必要があります。詳細については，SetParam メソッドを参照してください。

SetParamType メソッドは SetParam メソッドを呼び出す前に実行してください。SetParam メソッドで値を設定した後に SetParamType メソッドを実行した場合，そのパラメタの値はクリアされます。

HiRDB の Binary 型の場合

TYPE_BLOB_FILE_TEXT および TYPE_BLOB_FILE_BINARY は指定できません。メモリ経由でのパラメタ設定だけ実行できます。

XDM/RD の場合

Open メソッドを呼び出す前に，すべてのパラメタについてこのメソッドを呼び出してパラメタのデータ型を指定してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数 dwIndex の範囲が不正です（Execute メソッド実行前に実行した場合を含みます)。

DB_DRV_ERROR_SQL_NOT_FOUND
　SQL 文が設定されていません。

Top メソッド

機能

カーソルを ResultSet の先頭のレコードへ移動します。

形式

```
void Top(void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

カーソルを ResultSet の先頭のレコードへ移動します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_RESULTSET_NOT_EXIST
　非同期実行時に，検索結果がまだ取得できていません。

DB_ERROR_OUT_OF_RESULTSET
　ResultSet の範囲を超えるような呼び出しです。

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

Update メソッド

機能

フィールドの更新結果をデータベースへ通知します。

形式

```
void Update(void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

検索したレコードを更新する場合，Edit メソッドの後，SetField メソッドでフィールドの値を更新し，Update メソッドを呼び出すと，更新結果がデータベースへ通知されます。

通知されたデータはコミットされるまでデータベースに反映されません。

SetField メソッドで更新したあと，Update メソッドを呼び出さないで Next，又は PageNext メソッドを呼び出した場合，更新したフィールドの内容は無効になります。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_NOT_IN_EDIT
　Edit メソッドが実行されていません。

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行中です。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_ERROR_RESULTSET_NOT_EXIST
　非同期実行時に，検索結果がまだ取得できていません。

DB_ERROR_DAB_ACCESS_ERROR
　DABroker でエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR
　システムコールでエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_TIMEOUT
　タイムアウトが発生したため制御を戻します。

WaitForDataSource メソッド

機能

DBRResultSet オブジェクトで要求した実行待ち, 及び実行中の非同期処理が終了するまで待ちます。非同期処理中に同期を取りたい場合に利用できます。

形式

```
BOOLEAN WaitForDataSource(UINT32 swWaitTime = DBR_INFINITE)
```

引数

swWaitTime

非同期処理の終了を待つための最大時間（単位：ミリ秒），又は DBR_INFINITE を指定します。

指定した時間内に非同期処理が終了した場合，TRUE が返ります。指定した時間を経過しても非同期処理が終了しない場合，FALSE が返ります。非同期実行中の SQL がない場合は，TRUE が返ります。

DBR_INFINITE を指定した場合はタイムアウト時間を設定しません。非同期処理が終了するまで待ち続けます。

同期実行時はすぐに TRUE を返します。

戻り値

データ型：BOOLEAN

TRUE：すべての非同期処理が終了しました。

FALSE：タイムアウト時間が経過しました。

機能詳細

DBRResultSet オブジェクトで要求した実行待ち，及び実行中の非同期処理(SQL)が，終了するのを待ちます。

DBRDatabase クラスの WaitForDataSource メソッドとの違い

ResultSet オブジェクトの WaitForDataSource メソッドでは, WaitForDataSource メソッドを実行した ResultSet オブジェクトの非同期実行処理が終了するまで待ちます。ほかのオブジェクトの非同期実行処理については待ちません。そのため，ほかのオブジェクトの非同期実行処理が実行中でも，WaitForDataSource メソッドを呼び出したオブジェクトの非同期実行処理が終了すれば，WaitForDataSource メソッドは TRUE を返します。

同じ Database オブジェクトで実行しているすべての非同期実行処理が終了するのを待ちたい場合は，Database の WaitForDataSource メソッドを実行してください。

発生する例外

なし

5 詳細版関数詳細

DABroker では，C++による開発向けのプログラミングインタフェースとして，C++クラスライブラリを提供します。ここでは，詳細版のクラスライブラリごとに，プロパティやメソッドの詳細について説明します。

5.1　文法の説明順序

この章では,プロパティ,メソッドごとに，次のような順序で説明しています。

プロパティ名，又はメソッド名

機能

各プロパティ，又はメソッドの機能概要について記述しています。

形式

実際に指定する形式について記述しています。

引数は，特に断りのないかぎり，形式に示してある順序で記述してください。

```
DBDriver *Driver (LPCTSTR lpctAbstractName,
                  LPCTSTR lpctOptions = NULL,
                  UINT16 swTimeout = 0)          throw DBSQLCA
```

引数のうち=のない引数（lpctAbstractName）は，省略できないことを示します。

=のある引数（lpctOptions=NULL，swTimeout=0）は，指定を省略できることがあります。途中の引数を一つ又は複数省略することはできません。上記の例では，引数lpctAbstractName だけを指定することはできますが，lpctOptions だけを省略して，lpctAbstract 及び NameswTimeout を指定することはできません。 また，=の右辺の値は，デフォルト値であることを示しています。この例では，引数 lpctOptions の指定を省略すると NULL が，引数 swTimeout の指定を省略すると 0 が仮定されることを示しています。

引数

形式で説明されている引数とその意味について記述しています。

戻り値

戻り値には，メソッドの戻り値の型（データ型として記述）や説明が記述してあります。

また，主に戻り値の説明中にある（又は機能詳細などにある場合もあります），TRUE 及び FALSE の表記は，それぞれ非 0 及び 0 を意味しています。ヘッダでは TRUE を 1 と定義していますが，これは便宜上のものであり，TRUE の場合でも BOOLEAN 型の戻り値として 1 以外の値が返ることがあります。

したがって，BOOLEAN 型の戻り値で TRUE かどうかを判定する場合は，例 1 のような方法ではなく，例 2 に示す方法で判定してください。

（例 1）

```
BOOLEAN bFlag;
      :
if(bFlag == TRUE)
{
  // TRUE
}
```

（例 2）

```
BOOLEAN bFlag;
      :
if(bFlag)
{
```

```
  // TRUE
}
```

機能詳細

各プロパティやメソッドの解説が記述してあります。

発生する例外

メソッドでスローする可能性のある代表的なエラーが記述してあります。ここに書かれていないエラーをスローすることもあります。また，項目の先頭に記述されている DBSQLCA(RetCode)は，DBSQLCA クラスの RetCode プロパティにエラーがスローされることを示しています。

DB_ERROR_NOT_ENOUGH_MEMORY
　エラーコードを示します。

アプリケーション作成時には，RetCode プロパティに返るエラーコードを直接，比較，判定に利用できます。なお，アプリケーションで利用するためには，include 文でヘッダーファイル dbbroker.h ファイルをインクルードしておきます。

```
DBSQLCA* pError = NULL;
  try
  {
    :
    pError = pPrep->GetErrorStatus();
                              // SQL実行時のエラー情報を取得
    if(pError->RetCode != 0)
                              // DBSQLCAオブジェクトのRetCodeで確認
      throw *pError;
    :
  }
  catch(DBSQLCA e)
  {
  }

  try
  {
    :
    pDriver = DriverManager.Driver(StrDBMSName);
                                        // 使用するDBMSの定義
    :
  }
  catch(DBSQLCA e)
  {
  if(e.RetCode==DB_ERROR_NOT_ENOUGH_MEMORY)
                                        //エラーコードを判定に使う
    :
  }
```

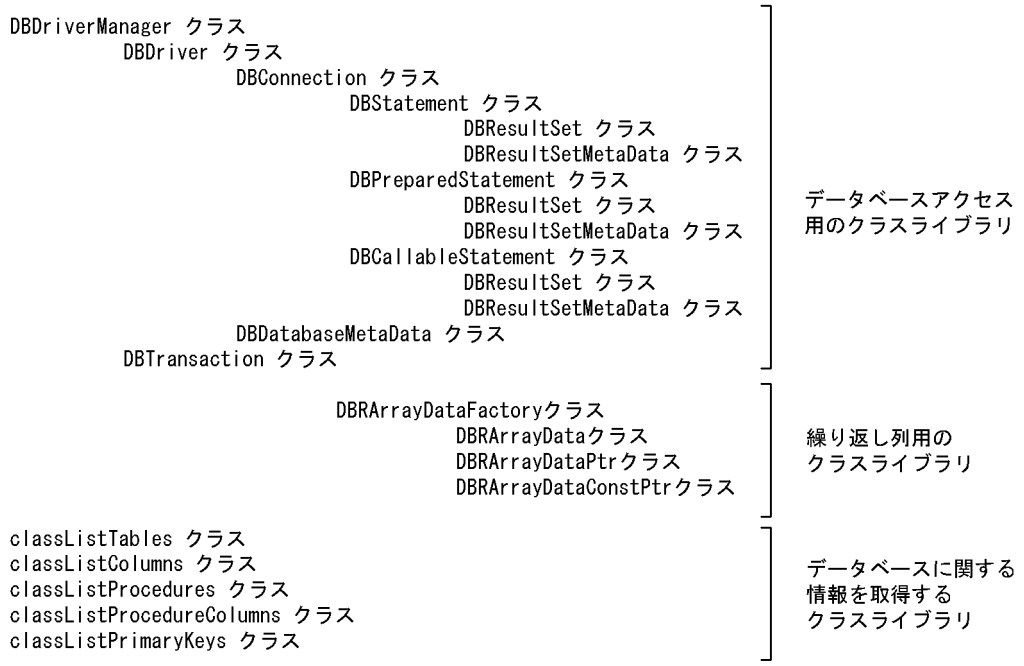
5.2　詳細版で利用できるクラスの概要

C++クラスライブラリのオブジェクト階層を図 5-1 に示します。

上位のクラスは，下位のクラスのオブジェクトを生成することを意味します。

なお，HiRDB の繰り返し列用のクラスライブラリは，簡易版でも利用できるため，詳細については，「6. 共通関数詳細」を参照してください。

図 5-1　C++クラスライブラリのオブジェクト階層



C++クラスライブラリのクラスの一覧を表 5-1 に示します。

表 5-1　C++クラスライブラリのクラスの一覧

クラス名	概 要
DBDriverManager クラス	DABroker のトップオブジェクトです。
DBDriver クラス	接続するデータベースについて管理します。
DBConnection クラス	データベースへの接続を管理します。
DBStatement クラス	SQL 文の実行を管理します。
DBResultSet クラス	ResultSet を管理します。
DBResultSetMetaData クラス	ResultSet の情報を管理します。
DBPreparedStatement クラス	実行時に動的に決まる?パラメタ（又はプレースフォルダ）を利用する SQL 文を管理します。
DBCallableStatement クラス	ストアドプロシジャの実行を管理します。
DBDatabaseMetaData クラス	データベースの情報を管理します。

クラス名	概 要
DBTransaction クラス	トランザクションを管理します。
DBRArrayDataFactory クラス	繰り返し列を扱う DBRArrayData を管理します。
DBRArrayData クラス	繰り返し列を管理します。
DBRArrayDataPtr クラス	要素の値を変更したり，参照します。
DBRArrayDataConstPtr クラス	要素の値を参照します。
classListTables クラス	テーブル一覧情報を管理します。
classListColumns クラス	フィールド一覧情報を管理します。
classListProcedures クラス	プロシジャ一覧情報を管理します。
classListProcedureColumns クラス	プロシジャのパラメタ一覧情報を管理します。
classListPrimaryKeys クラス	テーブル中のプライマリキーを構成するフィールドの一覧情報を管理します。

5.3　DBDriverManager クラスの詳細

C++クラスライブラリでのトップオブジェクトです。

ドライバオブジェクト(DBDriver)，及びトランザクションオブジェクト（DBTransaction）の生成や削除などのメソッドを提供するクラスです。

機 能	メソッド名
DBDriver オブジェクトを生成します。	Driver
例外が発生したときにメッセージテキストを取得できるように初期化します。	InitializeMessage
DBDriver オブジェクトを削除します。	RemoveDriver
DBTransaction オブジェクトを削除します。	RemoveTransaction
DBTransaction オブジェクトを生成します。	Transaction

Driver メソッド

機能

DBDriver オブジェクトを生成します。

DBDriver オブジェクトは，DBMS に接続するため情報を設定するオブジェクトで，接続する DBMS ごとに生成します。

形式

```
DBDriver *Driver (LPCTSTR lpctAbstractName,
                  LPCTSTR lpctOptions = NULL,
                  UINT16 swTimeout = 0)          throw DBSQLCA
```

- TPBroker の OTS インタフェース，又は OpenTP1 連携を使ったトランザクション制御の場合

```
DBDriver *Driver (UNIT16 ui16Driver,
                  LPCTSTR lpctOptions = NULL,
                  UINT16 ui16Timeout = 0);       throw DBSQLCA
```

引数

lpctAbstractName

接続先データベース定義ファイルに記述のあるデータベース種別名を指定します。

指定した名称と同じ名前を持つ DBDriver オブジェクトが既にある場合，そのオブジェクトのポインタを返します。

lpctOptions

DBMS にアクセスするときに，C++クラスライブラリを使ったトランザクション制御にするのか，TPBoker の OTS インタフェース，又は OpenTP1 連携を使ったトランザクション制御にするのかを指定します。次のどちらかの値で設定します。

- NULL ポインタ：C++クラスライブラリを使ったトランザクション制御。NULL ポインタ又は NULL 終端の文字列へのポインタを設定します。

- "XA"：TPBroker の OTS インタフェース，又は OpenTP1 連携を使ったトランザクション制御。OpenTP1 連携で"XA"以外を指定した場合は，グローバルトランザクションの制御対象とすることができません。

swTimeout

現在のバージョンでは値を指定しても意味を持ちません。指定を省略するか，0 を指定します。

ui16Driver

次の列挙型の値を指定します。

- DRV_TYPE_ORACLE7：RM として ORACLE を使用します。
- DRV_TYPE_HIRDB：RM として HiRDB を使用します。OpenTP1 連携時，この値以外を指定した場合の動作は保証しません。

ここに示した以外の値を指定した場合，動作は保証しません。

戻り値

データ型：DBDriver*

DBDriver オブジェクトへのポインタ。

機能詳細

データベースにアクセスするには，そのデータベースに接続することが必要です。通常，接続には接続先データベース定義ファイルで定義されている情報を使います。このメソッドでは，その情報のうち，データベース種別について引数 lpctAbstractName で指定します。

データベースにアクセスする際にトランザクション制御を，C++クラスライブラリ，又は TPBroker の OTS インタフェースのどちらで行うかも指定します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　指定した引数が誤っています。

DB_ERROR_NOT_SUPPORTED
　サポートされていない DBMS を指定しました。

DB_ERROR_CANNOT_BE_NULL
　引数 lpctAbstractName に NULL を指定しました。

DB_ERROR_DBDEFINITIONNAME_LENGTH_IS_ZERO
　引数 lpctAbstractName に空文字列を指定しました。

DB_ERROR_INITIALIZE_ERROR
　メモリ不足もしくは環境設定の不正によって，処理を続行できません。

DB_ERROR_CANNOT_USE_XADRIVER
　指定した非 XA インターフェース用のデータベース種別名は，既に XA インターフェース用として使われています。

DB_ERROR_CANNOT_USE_NONXADRIVER
　指定した XA インターフェース用のデータベース種別名は，既に非 XA インターフェース用として使われています。

DB_ERROR_DBDEFINITIONNAME_IS_UNAVAILABLE_IN_XA
　lpctAbstractName にデータベース種別名が指定されていますが, lpctOptions に"XA"が指定されています。

InitializeMessage メソッド

機能

例外が発生したときにメッセージテキストを取得できるように初期化します。

形式

```
void InitializeMessage(void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

DABroker for C++クラスライブラリ（以下クラスライブラリと呼びます）のメッセージテキストを初期化します。メッセージテキストとは，次のものを指します。

DB_ERROR_NOT_ENOUGH_MEMORY ＜－エラーコード
　メモリ容量が不足しています。 ＜－メッセージテキスト

メッセージテキストは，DABroker for C++が提供するメッセージテキストファイル（dabcppmj.txt 又は dabcppme.txt）に格納されています。

取得するメッセージテキストが日本語（SJIS モードの場合）になるか，英語（EUC モードまたは ASCII モードの場合）になるかは，DABroker の DAB_LANG（LANG 環境変数)に依存します。

クラスライブラリのメッセージテキストを取得したい場合は, このメソッドを呼び出してメッセージテキストを初期化しておきます。通常，トップオブジェクトを生成した後，すぐにこのメソッドを呼び出して初期化してください。

メッセージテキストは，ErrorMessage プロパティで取得します。

なお，クラスライブラリのエラーコードだけの取得でもよい場合は，このメソッドを呼び出す必要はありません。また，このメソッドを呼び出さなくても，データベースのエラーメッセージは取得できます。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_FILE_NOT_FOUND
　メッセージテキストファイルが見つかりません。

DB_ERROR_INITIALIZE_ERROR
　メモリ不足もしくは環境設定の不正によって，処理を続行できません。

RemoveDriver メソッド

機能

DBDriver オブジェクトを削除します。

形式

```
void RemoveDriver (LPCTSTR lpctAbstractName)
```

引数

lpctAbstractName

DBDriver オブジェクト作成時に指定したデータベース種別名，又は NULL を指定します。

データベース種別名を指定した場合，その名称を持つ DBDriver オブジェクトを削除します。

NULL を指定した場合は，データベース種別名を指定して作成されたすべての DBDriver オブジェクトを削除します。

戻り値

なし

機能詳細

引数に指定した名称の DBDriver オブジェクトを削除します。このメソッドを実行すると，対象とする DBDriver オブジェクトで生成した DBConnection オブジェクトをすべて削除します。このとき，トランザクションが終了していない場合は，ロールバックします。

非同期実行時に，実行待ち，又は実行中のステートメントがある場合は終了するのを待って，DBDriver オブジェクトを削除します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_CANNOT_FIND_OBJECT
　指定した名前を持つ DBDriver オブジェクトはありません。

RemoveTransaction メソッド

機能

DBTransaction オブジェクトを削除します。

形式

```
void RemoveTransaction (LPCTSTR lpctName) throw DBSQLCA
```

引数

lpctName

削除する DBTransaction オブジェクトの名前，又は NULL を指定します。

名前を指定した場合，その名前を持つ DBTransaction オブジェクトを削除します。

NULL を指定した場合，すべての DBTransaction オブジェクトを削除します。

戻り値

なし

機能詳細

引数に指定した名称の DBTransaction オブジェクトを削除します。対象となる DBTransaction オブジェクトのトランザクション実行中にこのメソッドを呼び出すと，登録されている DBConnection オブジェクトをロールバックし，トランザクションを破棄し DBTransaction オブジェクトを削除します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_CANNOT_FIND_OBJECT
　引数に指定した名前を持つ DBTransaction オブジェクトがありません。

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

Transaction メソッド

機能

DBTransaction オブジェクトを生成します。

DBTransaction オブジェクトは，トランザクション管理のためのオブジェクトで，1 コネクションにつき，一つの DBTransaction オブジェクトとしてください。

なお，複数のコネクションを一つのトランザクションで管理したい場合は，TPBroker の OTS 機能を利用してください。Driver メソッドで，TPBroker を使用する設定ができます。

形式

```
DBTransaction *Transaction (LPCTSTR lpctName=NULL) throw DBSQLCA
```

引数

lpctName

DBTransaction オブジェクトの名前，又は NULL を指定します。

名称を指定すると，その名前を持つ DBTransaction オブジェクトを生成し，オブジェクトへのポインタを返します。

NULL を指定する，又は指定を省略すると，DABroker が名前を自動生成して，DBTransaction オブジェクトを生成し，オブジェクトへのポインタを返します。

戻り値

データ型：DBTransaction*

DBTransaction オブジェクトへのポインタ。

機能詳細

引数に指定した名称で DBTransaction オブジェクトを生成します。データベースを更新するアプリケーションの作成では，必ずトランザクションを設定してください。設定しないと，SQL 文単位にコミットされます。

オブジェクト名の自動生成については，「3.3.1　オブジェクトの生成と削除」を参照してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_INITIALIZE_ERROR
　メモリ不足もしくは環境設定の不正によって，処理を続行できません。

5.4　DBDriver クラスの詳細

接続するデータベース種別，及びデータベースに関する情報を管理します。

DBConnection オブジェクトの生成，削除などのメソッドを提供するクラスです。

機 能	メソッド名
DBConnection オブジェクトを生成します。	Connect
データベース種別を取得します。	GetDriverType
SQL 文の非同期実行時のエラー情報を得るために，DBSQLCA オブジェクトへのポインタを取得します。	GetErrorStatus
対象となるオブジェクトを生成した DBDriverManager オブジェクトのポインタを取得します。	Parent
DBDriver(自分自身)を削除します。	Remove
DBConnection オブジェクトを削除します。	RemoveConnection

Connect メソッド

機能

DBConnection オブジェクトを生成します。

接続先のデータベース名やユーザ ID，パスワードなどを指定して，データベースに接続します。

形式

```
DBConnection *Connect(LPCTSTR lpctName,
                      LPCTSTR lpctUID,
                      LPCTSTR lpctPWD,
                      LPCTSTR lpctDBN,
                      LPCTSTR lpctOPT = NULL,
                      LPCTSTR lpctAbstractName = NULL,
                      UINT16 swWait = LOCK_OPT_NOWAIT,
                      UINT16 swTimeout = 0,
                      UINT16 swSync = STMT_SYNC)  throw DBSQLCA
```

引数

lpctName

生成する DBConnection オブジェクトの名前を指定します。このとき，同じ DBMS を使用する DBDriver オブジェクト内でユニークな名前を指定してください。

ここで指定した名前は，DABroker のデータベースアクセストレースの PAPNAME および拡張データベースアクセストレースの Client Name に出力されます。また，HiRDB を使用している場合は，HiRDB クライアント環境定義の PDCLTAPNAME に設定されます。

lpctUID

接続先データベースのユーザ ID を指定します。

データベースにアクセスするためのユーザ ID，又は NULL を指定します。

NULL を指定した場合，接続先データベース定義ファイルに指定されたユーザ ID を利用します。

SQL/K，又は XDM/SD の場合，Database Connection Server の認証属性の設定によっては，不正な値を設定してもデータベースに接続できるので注意してください。認証属性の詳細については，マニュアル「Database Connection Server」を参照してください。

TPBroker の OTS インタフェース，又は OpenTP1 連携を使ったトランザクション制御の場合は省略できません。

lpctPWD

接続先データベースのパスワードを指定します。

ユーザ ID に対応したパスワード，又は NULL を指定します。

NULL を指定した場合，接続先データベース定義ファイルに指定されたパスワードを利用します。SQL Anywhere，Adaptive Server Anywhere の場合は，ODBC のデータソース中でパスワードを指定している場合は，そのパスワードを利用することができます。

SQL/K，又は XDM/SD の場合，Database Connection Server の認証属性の設定によっては，不正な値を設定してもデータベースに接続できるので注意してください。認証属性の詳細については，マニュアル「Database Connection Server」を参照してください。

TPBroker の OTS インタフェース，又は OpenTP1 連携を使ったトランザクション制御の場合は省略できません。

lpctDBN

- C++クラスライブラリを使ったトランザクション制御の場合
  通常は NULL を指定します。
  データベースによっては，lpctDBN に接続先情報を指定できます。詳しい内容については，このメソッドの機能詳細を参照してください。
- TPBroker の OTS インタフェースを使ったトランザクション制御の場合
  ORACLE 使用時は，xa_open 文字列の DB フィールドで指定するデータベース名を指定します。
  HiRDB 使用時は，NULL を指定します。
- OpenTP1 連携を使ったトランザクション制御の場合
  NULL を指定します。

lpctOPT

- C++クラスライブラリを使ったトランザクション制御の場合
  通常は NULL を指定します。
  以降の引数をすべて省略する場合はこの引数の指定も省略できます。
  HiRDB の場合，HiRDB クライアント環境定義の環境変数を指定できます。
  データベースによっては，lpctDBN に接続先情報を指定できます。
  また，HiRDB の場合，HiRDB クライアント環境定義の環境変数を指定できます。
  詳しい内容については，このメソッドの機能詳細を参照してください。
- TPBroker の OTS インタフェースを使ったトランザクション制御の場合
  ORACLE 使用時は，NULL を指定します。

HiRDB 使用時は，"DABENVGRP=xxxx"という形式で，TM に HiRDB を登録する際に指定した環境変数グループ識別子を指定します。このとき，PDHOST，PDNAMEPORT は指定しないでください。

- OpenTP1 連携を使ったトランザクション制御の場合
  OpenTP1 連携の場合，OpenTP1 に一つの HiRDB しか登録されていない場合は，この引数は省略できます。OpenTP1 に複数の HiRDB が登録されている場合は，環境変数グループ識別子を指定しないとエラーになります。環境変数グループ識別子の詳細については HiRDB のマニュアルを参照してください。

lpctAbstractName

- C++クラスライブラリを使ったトランザクション制御の場合
  接続先データベース定義ファイル中の接続するデータベース名を指定します。
- TPBroker の OTS インタフェース，又は OpenTP1 連携を使ったトランザクション制御の場合
  NULL を指定します。

swWait

検索対象のレコードがほかのトランザクションによってロックされている場合の動作を指定します。ここでの処理が，DBConnection 内でのデフォルト値になります。DBStatement クラスの Execute，DBPreparedStatement クラスの Execute，及び DBCallableStatement クラスの SetProcedure メソッドの実行時にも指定できます。

SQL/K の場合は，どの値を指定しても，ロックが解除されるまで待ち状態になります。

XDM/SD の場合は，どの値を指定しても，ロックが解除されるまで待たないで，すぐにエラーを返します。

- LOCK_OPT_NOWAIT，又は LOCK_OPT_WITH_ROLLBACK
  ロックの解除を待たないで，すぐにエラーを返します。HiRDB 又は XDM/RD の場合は同時にロールバックを実行します。それ以外の DBMS の場合は，ロールバックは実行しません。
- LOCK_OPT_WITHOUT_ROLLBACK
  ロックの解除を待たないで，すぐにエラーを返します。ロールバックは実行しません。
  ただし，現在のバージョンでは，XDM/RD ではロールバックを実行します。また，HiRDB の DELETE,INSERT,UPDATE 文では，このオプションは無効で，LOCK_OPT_WAIT と同じ動作になります。
- LOCK_OPT_WAIT
  ロックが解除されるまで待ちます。

swTimeout

現在のバージョンでは値を指定しても意味を持ちません。指定を省略する又は 0 を指定します。

swSync

コネクションごとに，データベースアクセス（SQL 文）を同期処理，又は非同期処理のどちらで実行するかを指定します。

STMT_SYNC：SQL 文を同期処理で実行します。

STMT_ASYNC：SQL 文を非同期処理で実行します。

STMT_ASYNC を指定した場合，各ステートメントオブジェクトで実行される SQL 文は，子スレッドで処理されます。

- TPBroker の OTS インタフェースでトランザクションを制御する場合
  非同期処理の STMT_ASYNC は指定できません。
- OpenTP1 連携を使ったトランザクション制御の場合
  省略するか，必ず STMT_SYNC を指定します。ただし，グローバルトランザクションの制御対象とする場合，非同期処理はできません。

戻り値

データ型：DBConnection*

DBConnection オブジェクトへのポインタ。

機能詳細

DBConnection オブジェクトを生成し，データベースに接続します。DBConnection オブジェクトはデータベースへの接続ごとに生成してください。

データベースの接続は，DBDriver オブジェクトの生成時に指定したデータベース種別名と，このメソッドで指定するデータベース名によって決められます。

また，いったん，Close して再接続する場合は，DBConnection クラスの Connect メソッドでも各項目について指定できます。

このメソッドは，非同期実行可能メソッドです。

- 接続先データベース定義ファイルの定義内容を使用しないで接続する場合の補足事項
  データベースとの接続には，接続先データベース定義ファイルの使用をお勧めしますが，もし接続先データベース定義ファイル中の定義内容を使用しない場合には，引数 lpctAbstractName で NULL を指定し，使用する DBMS ごとに次の引数の指定が必要です。
  ただし，Database Connection Server 経由で接続するデータベース，SQL Server， 及び Adaptive Server Anywhere を使用する場合は，lpctAbstractName に NULL を指定できません。
  lpctAbstractName に NULL を指定しなくても，次の引数に値を指定すれば，接続先データベース定義ファイルの定義内容ではなく，引数に指定した内容で接続します。
  データベースとの接続には，接続先データベース定義ファイルの使用をお勧めしますが，もし使用しない場合には引数 lpctAbstractName で NULL を指定し，使用する DBMS ごとに次の引数の指定が必要です。
  - ORACLE の場合
    引数 lpctDBN に，SQL*NET 経由で接続する ORACLE のリスナー名称を指定します。リモートアクセスをしない場合は NULL を指定します。
  - HiRDB の場合
    引数 lpctOPT に，"PDHOST=XXXX;PDNAMEPORT=XXXX"という形式で接続先を指定します。環境変数に PDHOST，PDNAMEPORT が設定されていれば省略できます。
  - SQL Anywhere，Adaptive Server Anywhere，SQL Server の場合
    引数 lpctDBN に ODBC のデータソース名を指定します。引数 lpctOPT でデータソース名が定義されている場合は，この引数は省略できます。
    引数 lpctOPT に SQL Anywhere,Adaptive Server Anywhere,SQL Server のデータベース接続パラメタ（ConnectionName 以外）を指定できます。データベース接続パラメタでユーザ ID やパ

スワードを指定した場合，Connect メソッドの引数に指定したユーザ ID やパスワードよりも優先されます。

- RDA Link for Gateway 経由の XDM/RD の場合
  引数 lpctOPT に，RD ノード名を指定します。
- Database Connection Server 経由の XDM/RD の場合
  引数 lpctOPT に，次に示す形式で接続先を指定します。
  ”RDNODE=xxxxx;DBHOST=xxxxx;SNDBUFSZ=xxxxx;RCVBUFSZ= xxxx”
- Database Connection Server 経由の XDM/SD の場合
  lpctOPT で次に示す形式で接続先を指定します。
  ”SDNODE=xxxxx;DBHOST=xxxxx;SNDBUFSZ=xxxxx;RCVBUFSZ=xxxxx”
- Database Connection Server 経由の SQL/K の場合
  lpctOPT で次に示す形式で接続先を指定します。
  ”DBID=xxx1;DBHOST=xxx2;SNDBUFSZ=xxx3;RCVBUFSZ=xxx4”
  xxx1：分散定義名を指定します。
  xxx2：コネクションマルチ名を指定します。
  xxx3：通信バッファサイズを指定します。この指定は省略できます。
  xxx4：受信バッファサイズを調整します。この指定は省略できます。
  各項目の詳細については，マニュアル「DABroker」を参照してください。
- TPBroker の OTS インタフェースを使ったトランザクション制御の場合
  引数 lpctAbstractName にだけ NULL を指定します。

- HiRDB クライアント環境定義の環境変数指定
  HiRDB の場合，HiRDB クライアント環境定義の環境変数を lpctOPT に指定できます。
  引数の先頭に"DAB_HIRDBENV"がある場合，この接続でだけ有効にしたい HiRDB クライアント環境定義の環境変数を DAB_HIRDBENV+区切り文字(1 バイト)"以降に記述します。
  区切り文字に@を指定した場合の指定例を次に示します。

  (例 1)
  ”DAB_HIRDBENV@PDHOST=host.hitachi.co.jp@PDNAMEPORT=20000@PDSRVTYPE=VOS3@PDCWAITTIME=300”
  接続時に HiRDB クライアントに以下の値を設定します。
  PDHOST=host.hitachi.co.jp
  PDNAMEPORT=20000
  PDSRVTYPE=VOS3
  PDCWAITTIME=300

  (例 2)
  ”DAB_HIRDBENV@DABENVGRP=XXXX”
  接続時に HiRDB 環境変数グループ識別子 XXXX を指定します。

  注意事項
  - DAB_HIRDBENV の次の 1 文字には，必ず DABroker 動作環境定義ファイル(dasysconf)中で指定した区切り文字(変数名：DAB_HIRDB_DBINF_ESC)を指定してください。
  - DAB_HIRDBENV の文字列は必ず引数の先頭から指定してください。
    それ以外の場所で記述しても有効になりません。
    次のような指定をした場合の動作は保証しません。

```
” DAB_HIRDBENV@PDHOST=host@PDNAMEPORT=20000”
”DAB_HIRDBENVTEST@PDHOST=host@PDNAMEPORT=20000”
```

- 指定できる HiRDB の環境変数は PDUSER 以外となります（PDUSER の指定は無視されます）。また，HiRDB で指定できる環境変数以外を指定した場合，動作は保証しません。
- 環境変数で指定した値は，DABroker for C++，及び DABroker ではエラーをチェックしないため，環境変数に誤りがある場合は HiRDB のエラーとなります。
- この機能を使用する場合，従来 DABroker で設定していた，PDCLTAPNAME は設定されません。必要な場合は，この機能で PDCLTAPNAME を指定してください。
- HiRDB 環境変数グループ識別子を指定する場合，他の HiRDB 環境変数の指定は無効となるので御注意ください。
- この機能の指定値は，実行ユーザ(.profile)の指定よりも優先されます。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_NAME_INVALID
　DBConnection オブジェクトの名前を指定していません。

DB_ERROR_CANNOT_BE_NULL
　DBConnection オブジェクトの名前に NULL を指定しています。

DB_ERROR_CONNECTSTRING_INVALID
　接続文字列の形式が間違っています。

DB_ERROR_NAME_ALREADY_USED
　指定した名前は既に使用されています。

DB_DRV_ERROR_CONNECT_NAME_ALREADY
　同じ DBConnection オブジェクト名で既に接続しています。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_ERROR_DAB_ACCESS_ERROR
　DABroker でエラーが発生しました。DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR
　システムコールでエラーが発生しました。DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_ILLEGAL_VALUE
　指定した引数が不正です。DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_CANNOT_USE_ABSTRACTNAME
　DBDriver オブジェクトにデータベース種別名が指定されていないので，データベース名を指定できません。

DB_ERROR_NOT_FOUND_DBDEFINITION
　データベース種別名，又はデータベース名が見つかりません。

DB_ERROR_CONNECTDBDEFINITIONNAME_LENGTH_IS_ZERO
　データベース名の長さが 0 です。
DB_ERROR_TIMEOUT
　タイムアウトが発生したため制御を戻します。
DB_ERROR_NOT_EXECUTE_ASYNC
　XA インターフェース使用時に非同期処理は指定できません。

GetDriverType メソッド

機能

データベース種別を取得します。

形式

```
UINT16 GetDriverType(void)
```

引数

なし

戻り値

データ型：UINT16

RV_TYPE_HIRDB：HiRDB 用ドライバ

DRV_TYPE_ORACLE7：Oracle 用ドライバ

DRV_TYPE_ANYWHERE：SQL Anywhere5.x 用ドライバ

DRV_TYPE_ANYWHERE6：Adaptive Server Anywhere6.x 用ドライバ

DRV_TYPE_SQLSERVER7：SQL Server 用ドライバ

DRV_TYPE_VOS3XDMRD：XDM/RD 用ドライバ（RDA Link for Gateway 経由）

DRV_TYPE_VOS3XDMRD_DBS：XDM/RD 用ドライバ（Database Connection Server 経由）

DRV_TYPE_VOSKSQLK_DBS：SQL/K 用ドライバ（Database Connection Server 経由）

DRV_TYPE_VOS3XDMSD_DBS：XDM/SD 用ドライバ（Database Connection Server 経由）

DRV_TYPE_HIRDB_XA：XA インタフェースを利用した HiRDB 用ドライバ

DRV_TYPE_ORACLE7_XA：XA インタフェースを利用した Oraclex 用ドライバ

DRV_TYPE_UNKNOWN：不明

機能詳細

データベース種別を取得できます。

Connect メソッドを呼び出して DBConnection オブジェクトが生成されたか，Connect メソッドで DB_ERROR_DRIVER_ERROR がスローされた後に，接続しようとしたデータベース種別を取得できます。Connect メソッドを一度実行するまでは DRV_TYPE_UNKNOWN が返ります。

発生する例外

なし

GetErrorStatus メソッド

機能

SQL 文の非同期実行時のエラー情報を得るために，DBSQLCA オブジェクトへのポインタを取得します。

形式

```
DBSQLCA *GetErrorStatus(void)
```

引数

なし

戻り値

データ型：DBSQLCA*

DBSQLCA オブジェクトへのポインタ。

機能詳細

DBSQLCA オブジェクトへのポインタを取得します。

非同期処理のときは，エラーが発生しても直にエラーを取得できません。このため，ユーザは任意の時点でエラーを確認する必要があります。

非同期実行でエラーが発生した場合，DBSQLCA オブジェクトにエラー情報が設定されます。

DBSQLCA オブジェクトでは，非同期実行中に発生したエラー情報を最大 100 回分保存できます。エラー情報は，新しいエラー情報から 100 回分保存されるため，エラー情報を確認したあとは DBSQLCA クラスの Delete メソッドを呼び出して，不要なエラー情報をクリアしておいてください。

発生する例外

なし

Parent メソッド

機能

対象となるオブジェクトを生成した DBDriverManager オブジェクトのポインタを取得します。

形式

```
LPOBJECT Parent(void)
```

引数

なし

戻り値

データ型：LPOBJECT

対象となるオブジェクトを生成した DBDriverManager オブジェクトへのポインタ。

機能詳細

なし

発生する例外

なし

Remove メソッド

機能

DBDriver オブジェクト(自分自身)を削除します。

形式

```
void Remove(void)
```

引数

なし

戻り値

なし

機能詳細

DBDriver オブジェクト(自分自身)を削除します。

このメソッドを呼び出した DBDriver オブジェクト以外の DBDriver オブジェクトを削除する場合は，DBDriverManager オブジェクトの RemoveDriver メソッドが利用できます。詳細については，「5.3 DBDriverManager クラスの詳細」の RemoveDriver メソッドを参照してください。

発生する例外

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中ステートメントがあります。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

RemoveConnection メソッド

機能

DBConnection オブジェクトを削除します。

形式

```
void RemoveConnection(LPCTSTR lpctName) throw DBSQLCA
```

引数

lpctName

削除する DBConnection オブジェクトの名前を指定します。

戻り値

なし

機能詳細

引数 lpctName で指定した名前の DBConnection オブジェクトを削除します。また，この DBConnection オブジェクトで生成されたステートメントはすべて削除され，トランザクション中のコネクションがある場合はロールバックしてトランザクションを破棄します。

実行待ちステートメントがある場合（非同期処理），DBConnection オブジェクトは削除できません。その場合は，実行待ち，及び実行中の処理の終了を確認してから，DBConnection オブジェクトを削除してください。

なお，一時的にデータベースとの接続を切り離したい場合は，Close メソッドが使えます。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_CANNOT_BE_NULL
　オブジェクトの名前に NULL を指定しています。

DB_ERROR_CANNOT_FIND_OBJECT
　指定した名前を持つオブジェクトがありません。

DB_ERROR_IN_ASYNC_EXECUTE
　SQL の非同期実行中です。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_ERROR_DAB_ACCESS_ERROR
　DABroker でエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR
　システムコールでエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

5.5　DBConnection クラスの詳細

データベースへの接続を管理します。

データベースとの接続・切断や，SQL 文を実行するオブジェクトの生成・削除などのメソッドを提供するクラスです。

データベースに接続するためには，DBDriver クラスの Connect メソッドを呼び出して，DBConnection オブジェクトを生成します。このとき，同期接続，又は非同期接続を選択できます。

非同期接続を選択した場合，データベースへの接続処理を含めて，同一セッション中のデータベースへの処理要求(クラスライブラリのメソッド呼び出し)は，非同期に実行されます。この場合は，別スレッド上で処理されるため，メソッドの呼び出しでアプリケーションが「待ち」になることはありません。その代わりに，メソッドの実行中にエラーが発生した場合，非同期で処理されているメソッドではエラーをスローできません。アプリケーション側で適当なタイミングを設定し，必要であればエラーを確認してください。

機 能	メソッド名
データベースとの接続を切り離します。	Close
Close メソッドを使って切断したデータベースと再接続します。	Connect
DBCallableStatement オブジェクトを生成します。	CreateCallableStatement
DBPreparedStatement オブジェクトを生成します。	CreatePreparedStatement
DBStatement オブジェクトを生成します。	CreateStatement
Transaction オブジェクトとの関連付けを解消します。	EraseTransaction
検索結果を必要としない，SQL 文を実行します。	ExecuteDirect
DBRArrayDataFactory オブジェクトを生成します。	GetArrayDataFactory
SQL 文の非同期実行時のエラー情報を得るために，DBSQLCA オブジェクトへのポインタを取得します。	GetErrorStatus
DBDatabaseMetaData オブジェクトを生成します。	GetMetaData
DBConnection オブジェクトの名称を取得します。	GetName
DBConnection オブジェクトで実行待ち，又は実行中のステートメントがあるかどうかを確認します。	InWaitForDataSource
データベースが切断されているかどうかを確認します。	IsClosed
対象となるオブジェクトを生成した DBDriver オブジェクトのポインタを取得します。	Parent
DBTransaction オブジェクトと DBConnection オブジェクトを関連付けます。	RegisterTransactions
DBConnection オブジェクト(自分自身)を削除します。	Remove
DBCallableStatement オブジェクトを削除します。	RemoveCallableStatement
DBPreparedStatement オブジェクトを削除します。	RemovePreparedStatement
DBStatement オブジェクトを破棄します。	RemoveStatement

機 能	メソッド名
RegisterTransactions メソッドで関連付けた DBTransaction オブジェクトのポインタを取得します。	Transaction
DBConnection オブジェクトで要求した実行待ち，及び実行中の非同期処理がすべて終了するまで待ちます。	WaitForDataSource

Close メソッド

機能

データベースとの接続を切り離します。

形式

```
void Close(void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

DBConnection オブジェクトに対して次の処理をして，データベースとの接続を切り離します。トランザクションが完了していないコネクションの場合には，ロールバック後，データベースとの接続を切り離します。接続していないときは何もしません。

- すべてのステートメントオブジェクトの削除
- DBDatabaseMetaData オブジェクトの削除
- Transaction オブジェクトへの関連付けを解消

発生する例外

DBSQLCA(RetCode)

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

DB_ERROR_DAB_ACCESS_ERROR
　DABroker でエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR
　システムコールでエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

Connect メソッド

機能

Close メソッドを使って切断したデータベースと再接続します。

形式

```
void Connect (LPCTSTR lpctUID,
              LPCTSTR lpctPWD,
              LPCTSTR lpctDBN,
              LPCTSTR lpctOPT = NULL,
              LPCTSTR lpctAbstractName = NULL,
              UINT16 swWait = LOCK_OPT_NOWAIT,
              UINT16 swTimeout = 0,
              UINT16 swSync = STMT_SYNC)        throw DBSQLCA
```

引数

lpctUID

接続先データベースのユーザ ID を指定します。

SQL/K，又は XDM/SD の場合，Database Connection Server の認証属性の設定によっては，不正な値を設定してもデータベースに接続できるので注意してください。認証属性の詳細については，マニュアル「Database Connection Server」を参照してください。

lpctPWD

接続先データベースのパスワードを指定します。

SQL/K，又は XDM/SD の場合，Database Connection Server の認証属性の設定によっては，不正な値を設定してもデータベースに接続できるので注意してください。認証属性の詳細については，マニュアル「Database Connection Server」を参照してください。

lpctDBN

NULL を指定します。

lpctOPT

通常は NULL を指定します。以降の引数をすべて省略する場合はこの引数の指定も省略できます。

- SQL/K の場合

  lpctAbstractName に NULL を指定して，lpctOPT で次に示す形式で接続先を指定すると，接続先データファイルを使用しないで接続できます。

  `DBID=xxx1;DBHOST=xxx2;SNDBUFSZ=xxx3;RCVBUFSZ=xxx4`

  xxx1：分散定義名を指定します。

  xxx2：コネクションマルチ名を指定します。

  xxx3：通信バッファサイズを指定します。この指定は省略できます。

  xxx4：受信バッファサイズを調整します。この指定は省略できます。

  各項目の詳細については，マニュアル「DABroker」を参照してください。

- XDM/SD の場合

  lpctAbstractName に NULL を指定して，lpctOPT で次に示す形式で接続先を指定します。

  `SDNODE=xxxxx;DBHOST=xxxxx;SNDBUFSZ=xxxxx;RCVBUFSZ=xxxxx`

lpctAbstractName

接続先データベース定義ファイル中のデータベース名を指定します。

- SQL/K，又は XDM/SD の場合
  NULL を指定して，lpctOPT で接続先を指定することで，接続先データファイルを使用しないで接続できます。

swWait

検索対象のレコードがほかのトランザクションによってロックされている場合の動作を指定します。ここでの処理が，DBConnection 内でのデフォルト値になります。DBStatement クラスの Execute，DBPreparedStatement クラスの Execute，及び DBCallableStatement クラスの SetProcedure メソッドの実行時にも指定できます。

SQL/K の場合は，どの値を指定しても，ロックが解除されるまで待ち状態になります。

XDM/SD の場合は，どの値を指定しても，ロックが解除されるまで待たないで，すぐにエラーを返します。

- LOCK_OPT_NOWAIT，又は LOCK_OPT_WITH_ROLLBACK
  ロックの解除を待たないで，すぐにエラーを返します。HiRDB 又は XDM/RD の場合は同時にロールバックを実行します。それ以外の DBMS の場合は，ロールバックは実行しません。
- LOCK_OPT_WITHOUT_ROLLBACK
  ロックの解除を待たないで，すぐにエラーを返します。ロールバックは実行しません。
  ただし，現在のバージョンでは，XDM/RD ではロールバックを実行します。また，HiRDB の DELETE,INSERT,UPDATE 文では，このオプションは無効で，LOCK_OPY_WAIT と同じ動作になります。
- LOCK_OPT_WAIT
  ロックが解除されるまで待ちます。

swTimeout

現在のバージョンでは値を指定しても意味を持ちません。指定を省略する又は 0 を指定します。

swSync

コネクションごとに，データベースアクセスを（SQL 文）を非同期，又は同期処理のどちらで実行するかを指定します。ただし，TPBroker の OTS インタフェースでトランザクションを制御する場合は，非同期処理の STMT_ASYNC は指定できません。

STMT_SYNC：SQL 文を同期処理で実行します。

STMT_ASYNC：SQL 文を非同期処理で実行します。

戻り値

なし

機能詳細

Close メソッドを呼び出してデータベースとの接続を切断した後, 再度データベースへ接続する場合に呼び出します。

各引数の詳細な説明については，「5.4 DBDriver クラスの詳細」の Connect メソッドを参照してください。

このメソッドは，非同期実行可能なメソッドです。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_CONNECTSTRING_INVALID
　接続文字列の形式が間違っています。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_ERROR_DAB_ACCESS_ERROR
　DABroker でエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR
　システムコールでエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_ILLEGAL_VALUE
　指定した引数が不正です。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_TIMEOUT
　タイムアウトが発生したため制御を戻します。

DB_ERROR_NOT_EXECUTE_ASYNC
　XA インターフェース使用時に非同期処理は指定できません。

CreateCallableStatement メソッド

機能

DBCallableStatement オブジェクトを生成します。

DBCallableStatement オブジェクトはストアドプロシジャの実行を管理します。

形式

```
DBCallableStatement *CreateCallableStatement
                        (LPCTSTR lpctName = NULL) throw DBSQLCA
```

引数

lpctName

生成する DBCallableStatement オブジェクトの名前，又は NULL を指定します。

NULL を指定する，又は指定を省略すると，DABroker が名前を自動生成して，DBCallableStatement オブジェクト生成し，オブジェクトへのポインタを返します。

指定した名前を持つオブジェクトが既にある場合は，そのオブジェクトのポインタを返します。

戻り値

データ型：DBCallableStatement*

DBCallableStatement オブジェクトへのポインタ。

機能詳細

DBCallableStatement オブジェクトを生成し，オブジェクトのポインタを返します。

ストアドプロシジャを実行するためには，この DBCallableStatement オブジェクトを使います。

ストアドプロシジャの実行方法については「2.6.6 ストアドプロシジャの利用」を参照してください。

生成できる DBCallableStatement オブジェクトの個数には制限があります。詳細については「3.3.2 データベースアクセスリソース数の制限」を参照してください。

自動的に生成されるオブジェクト名については「3.3.1 オブジェクトの生成と削除」を参照してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_NOT_CONNECTED
　データベースに接続していません。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_DRV_ERROR_STAT_COUNT
　ステートメントの個数が上限値を超えました。

CreatePreparedStatement メソッド

機能

DBPreparedStatement オブジェクトを生成します。

DBPreparedStatement オブジェクトは，?パラメタを使った SQL 文を実行するためのオブジェクトです。

形式

```
DBPreparedStatement *CreatePreparedStatement
                         (LPCTSTR lpctStatement,
                          LPCTSTR lpctName = NULL) throw DBSQLCA
```

引数

lpctStatement

0 個以上の?パラメタを含んだ SQL 文を指定します。

lpctName

生成する DBPreparedStatement オブジェクトの名前，又は NULL を指定します。

NULL を指定する，又は指定を省略すると，DABroker が名前を自動生成して，DBPreparedStatement オブジェクトを生成し，オブジェクトへのポインタを返します。

指定した名前を持つオブジェクトが既にあった場合，そのオブジェクトのポインタが返ります。

戻り値

データ型：DBPreparedStatement*

DBPreparedStatement オブジェクトへのポインタ。

機能詳細

？パラメタを含む SQL 文を組み立て, このメソッドを呼び出して DBPreparedStatement オブジェクトを生成します。

なお，引数 lpctStatement で NULL 以外の値を指定して，既にあるオブジェクトと同じ名称を指定した場合はエラーをスローします。SQL の実行方法については「2.6 詳細版クラスのデータベースアクセス」を参照してください。

自動的に生成されるオブジェクト名については，「3.3.1 オブジェクトの生成と削除」を参照してください。

生成できる DBPreparedStatement オブジェクトの個数には制限があります。詳細については「3.3.2 データベースアクセスリソース数の制限」を参照してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_NOT_CONNECTED
　DB に接続していません。

DB_ERROR_INVALID_ARGUMENT
　新規生成時に，既に使われているオブジェクト名称を指定しています。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_ERROR_NAME_ALREADY_USED
　指定した名称は既に使用されています（非同期実行時）。

DB_ERROR_CANNOT_FIND_OBJECT
　指定した名前を持つオブジェクトがありません。

DB_DRV_ERROR_STAT_COUNT
　ステートメントの個数が上限値を超えました。

CreateStatement メソッド

機能

DBStatement オブジェクトを生成します。

DBStatement オブジェクトでは，SQL 文の実行を管理します。

形式

```
DBStatement *CreateStatement(LPCTSTR lpctName=NULL)
                                              throw DBSQLCA
```

引数

lpctName

生成する DBStatement オブジェクトの名前，又は NULL を指定します。

NULL を指定する，又は指定を省略すると，DABroker が名前を自動生成して，DBStatement オブジェクトを生成し，オブジェクトへのポインタを返します。

指定した名前を持つオブジェクトが既にある場合は，そのオブジェクトのポインタを返します。

戻り値

データ型：DBStatement*

DBStatement オブジェクトへのポインタ。

機能詳細

？パラメタを使わない SQL 文を実行する場合，このメソッドで DBStatement オブジェクトを生成します。？パラメタを含む SQL 文を実行したい場合は，CreatePreparedStatement メソッドを呼び出して DBPreparedStatement オブジェクトを生成してください。

SQL の実行方法については「2.6 詳細版クラスのデータベースアクセス」を参照してください。

自動的に生成されるオブジェクト名については，「3.3.1　オブジェクトの生成と削除」を参照してください。

オブジェクトの個数には制限があります。詳細については，「3.3.2 データベースアクセスリソース数の制限」を参照してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_ERROR_NOT_CONNECTED
　データベースに接続していません。
DB_DRV_ERROR_STAT_COUNT
　ステートメントの個数が上限値を超えました。

EraseTransaction メソッド

機能

Transaction オブジェクトとの関連付けを解消します。

形式

```
void EraseTransaction (void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

DBTransaction オブジェクトに対する DBConnection オブジェクトの関連付けを解消します。

トランザクション実行中の場合, 関連付けの解消時にロールバックします。自動コミットが設定されている場合は何もしません。

TPBroker の OTS インタフェースを使ったトランザクション制御の場合

登録されている DBConnection オブジェクトが OTS インタフェースを使用している場合でもこのクラスの各メソッドは呼び出せます。しかし，関連付けの解消時には，ロールバックは行われないため，注意してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NO_CONNECTION_OBJECT
　DBConnection オブジェクトが DBTransaction オブジェクトに関連付けられていません。
DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中のステートメントがあります。
DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。
DB_ERROR_DAB_ACCESS_ERROR
　DABroker でエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。
DB_ERROR_DAB_SYSTEMCALL_ERROR
　システムコールでエラーが発生しました。

DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_TIMEOUT
タイムアウトが発生したため制御を戻します。

ExecuteDirect メソッド

機能

検索結果を必要としない，SQL 文を実行します。

形式

```
void ExecuteDirect(LPCTSTR lpctStatement) throw DBSQLCA
```

引数

lpctStatement

実行する SQL 文を指定します。

戻り値

なし

機能詳細

引数 lpctStatement で指定された SQL 文を実行します。

このメソッドでは，?パラメタを使った SQL 文を指定できません。また，SELECT 文を実行しても，検索結果は取得できません。

SQL の実行方法については「2.6 詳細版クラスのデータベースアクセス」を参照してください。

非同期に実行できる SQL 文の個数には制限があります。詳細については，「3.3.2 データベースアクセスリソース数の制限」を参照してください。

このメソッドは，非同期実行可能なメソッドです。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
メモリ容量が不足しています。

DB_ERROR_NOT_CONNECTED
データベースに接続していません。

DB_ERROR_CANNOT_BE_NULL
引数に NULL を指定しています。

DB_ERROR_DRIVER_ERROR
DBMS でエラーが発生しました。

DB_ERROR_DAB_ACCESS_ERROR
DABroker でエラーが発生しました。

DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR
システムコールでエラーが発生しました。
DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_TIMEOUT
タイムアウトが発生したため制御を戻します。

DB_DRV_ERROR_STAT_COUNT
ステートメントの個数が上限値を超えました。(非同期実行時)

GetArrayDataFactory メソッド

機能

繰り返し列を扱うために，DBRArrayDataFactory オブジェクトを生成し，ポインタを取得します。

形式

```
DBRArrayDataFactory * GetArrayDataFactory(void) throw DBSQLCA
```

引数

なし

戻り値

データ型：DBRArrayDataFactory *

DBRArrayDataFactory オブジェクトのポインタ。

機能詳細

GetArrayDataFactory メソッドは，DBRArrayData オブジェクトを管理するための
DBRArrayDataFactory オブジェクトを生成し，ポインタを取得するメソッドです。

この DBRArrayDataFactory オブジェクトの生成後，CreateArrayData メソッドで DBRArrayData オブジェクトを生成します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_DRIVER_NOT_LOADED
データベースとの接続時に非同期処理を設定していますが，接続に失敗した上に，接続する DBMS 種別も不明です。

GetErrorStatus メソッド

機能

SQL 文の非同期実行時のエラー情報を得るために，DBSQLCA オブジェクトへのポインタを取得します。

形式

```
DBSQLCA *GetErrorStatus(void)
```

引数

なし

戻り値

データ型：DBSQLCA*

DBSQLCA オブジェクトへのポインタ。

機能詳細

DBSQLCA オブジェクトへのポインタを取得します。

非同期処理のときは，エラーが発生しても直にエラーを取得できません。このため，ユーザは任意の時点でエラーを確認する必要があります。

非同期実行時にエラーが発生した場合，DBSQLCA オブジェクトにエラー情報が設定されます。

DBSQLCA オブジェクトでは，非同期実行中に発生したエラー情報を最大 100 回分保存できます。エラー情報は，新しいエラー情報から 100 回分保存されるため，エラー情報を確認したあとは DBSQLCA クラスの Delete メソッドを呼び出して，不要なエラー情報をクリアしておいてください。

発生する例外

なし

GetMetaData メソッド

機能

DBDatabaseMetaData オブジェクトを生成します。

DBDatabaseMetaData オブジェクトでは，データベース情報を管理します。

形式

```
DBDatabaseMetaData *GetMetaData(void) throw DBSQLCA
```

引数

なし

戻り値

データ型：DBDatabaseMetaData*

DBDatabaseMetaData オブジェクトへのポインタ。

機能詳細

DBDatabaseMetaData オブジェクトを生成し，そのオブジェクトへのポインタを返します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_NOT_CONNECTED
　データベースに接続されていません。

GetName メソッド

機能

DBConnection オブジェクトの名称を取得します。

形式

```
LPCTSTR GetName(void)
```

引数

なし

戻り値

データ型：LPCTSTR

DBConnection オブジェクトの名前を返します。

機能詳細

DBConnection オブジェクトの名前を取得します。

発生する例外

なし

InWaitForDataSource メソッド

機能

DBConnection オブジェクトで実行待ち，又は実行中のステートメントがあるかどうかを確認します。同じ DBConnection オブジェクトで実行しているすべてのステートメントが対象になります。

形式

```
BOOLEAN InWaitForDataSource(void)
```

引数

なし

戻り値

データ型：BOOLEAN

TRUE：非同期処理の継続中です。

FALSE：非同期処理は終了しています。

機能詳細

子スレッドで実行待ち又は実行中の SQL 文があれば TRUE を，実行待ち又は実行中の SQL 文がなければ FALSE を返します。

厳密には，SQL を実行する子スレッドがあれば TRUE を，なければ FALSE を返します。同期実行接続時には常に FALSE を返します。

なお，各クラスの InExecute メソッドと，InWaitForDataSource メソッドの違いは，前者が InExecute メソッドを実行したオブジェクトの非同期処理が実行中かどうかを確認するのに対して，後者は DBConnection オブジェクトで生成したすべてのオブジェクトの非同期処理が実行中かどうかを確認できます（対象となるコネクション内のすべての非同期処理が終わるのを確認できます）。

発生する例外

なし

IsClosed メソッド

機能

データベースが切断されているかどうかを確認します。

形式

```
BOOLEAN  IsClosed(void)
```

引数

なし

戻り値

データ型：BOOLEAN

TRUE：データベースと接続されていません。

FALSE：データベースと接続しています。

機能詳細

なし

発生する例外

なし

Parent メソッド

機能

対象となるオブジェクトを生成した DBDriver オブジェクトのポインタを取得します。

形式

```
LPOBJECT Parent(void)
```

引数

なし

戻り値

データ型：LPOBJECT

対象となるオブジェクトを生成した DBDriver オブジェクトへのポインタ。

機能詳細

なし

発生する例外

なし

RegisterTransactions メソッド

機能

DBTransaction オブジェクトと DBConnection オブジェクトを関連付けます。

トランザクションの開始前, コネクションとトランザクションを対応させる必要があります。この関連付けによって，そのコネクションに対してトランザクション制御ができるようになります。

形式

```
void RegisterTransactions (DBTransaction* pTransaction)
                                                throw DBSQLCA
```

引数

pTransaction

トランザクションを管理する DBTransaction オブジェクトのポインタを指定します。

戻り値

なし

機能詳細

トランザクションをアプリケーションから制御する場合，DBConnection オブジェクトを引数 pTransaction に指定した DBTransaction オブジェクトに関連付けます。

DBTransaction オブジェクトへの関連付けはデータベースへの接続単位でします。1 トランザクションでは，1 コネクションを管理します。

対象とする DBTransaction オブジェクトで自動コミットが設定されている場合，DBConnection オブジェクト(コネクション)は関連付けられた時に自動コミット設定になります。また，DBTransaction オブジェクトで自動コミットが設定されていない場合，DBConnection オブジェクトを関連付け，BeginTrans メソッド(DBTransaction クラス)を呼び出した時点からトランザクションが開始されます。

また，RegisterTransactions メソッドを指定しない場合は，SQL 文を 1 文実行するごとにコミットされるので注意してください。

TPBroker の OTS インタフェースを使ったトランザクション制御の場合

登録されている DBConnection オブジェクトが OTS インタフェースを使用している場合でもこのクラスの各メソッドは呼び出せますが，このような場合，DBTransaction オブジェクトではトランザクション制御はしないため，このメソッドの機能は無効になります。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_CANNOT_BE_NULL
　引数に NULL を指定しています。

DB_ERROR_ALREADY_REGISTERED
　既に登録されています。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中ステートメントがあります。

Remove メソッド

機能

DBConnection オブジェクト(自分自身)を削除します。

形式

```
void Remove(void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

データベースとの接続を切り離し，DBConnection オブジェクトから作られたステートメントと一覧オブジェクトをすべて削除します。また，DBTransaction オブジェクトとの関連付けを解消した上で，自分自身を削除します。

コミットされていないトランザクションはロールバックされます。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中ステートメントがあります。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_ERROR_DAB_ACCESS_ERROR
　DABroker でエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR
　システムコールでエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

RemoveCallableStatement メソッド

機能

DBCallableStatement オブジェクトを削除します。

形式

```
void RemoveCallableStatement(LPCTSTR lpctName = NULL)
                                                throw DBSQLCA
```

引数

lpctName

削除する DBCallableStatement オブジェクトの名前，又は NULL を指定します。

NULL を指定した場合，この DBConnection オブジェクトが持っているすべての DBCallableStatement オブジェクトを削除します。

戻り値

なし

機能詳細

引数 lpctName で指定された名前を持つ DBCallableStatement オブジェクトを削除します。また，削除するステートメントで生成された DBResultSet オブジェクトも削除します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

DB_ERROR_CANNOT_FIND_OBJECT
　指定した名前を持つ DBCallableStatement オブジェクトがありません。

RemovePreparedStatement メソッド

機能

DBPreparedStatement オブジェクトを削除します。

形式

```
void RemovePreparedStatement(LPCTSTR lpctName = NULL)
                                                throw DBSQLCA
```

引数

lpctName

削除する DBPreparedStatement オブジェクトの名前，又は NULL を指定します。

NULL を指定した場合, この DBConnection オブジェクトが持っているすべての DBPreparedStatement オブジェクトを削除します。

戻り値

なし

機能詳細

引数 lpctName で指定された名前を持つ DBPreparedStatement オブジェクトを削除します。また，削除する DBPreparedStatement オブジェクトで生成された DBResultSet オブジェクトも削除します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

DB_ERROR_CANNOT_FIND_OBJECT
　指定した名前を持つ DBPreparedStatement オブジェクトがありません。

RemoveStatement メソッド

機能

DBStatement オブジェクトを削除します。

形式

```
void RemoveStatement(LPCTSTR lpctName = NULL) throw DBSQLCA
```

引数

lpctName

削除する DBStatement オブジェクトの名前，又は NULL を指定します。

NULL を指定した場合，この DBConnection オブジェクトが持っているすべての DBStatement オブジェクトを削除します。

戻り値

なし

機能詳細

引数 lpctName で指定された名前を持つ DBStatement オブジェクトを削除します。また，削除する DBStatement オブジェクトで生成した DBResultSet オブジェクトも削除します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

DB_ERROR_CANNOT_FIND_OBJECT
　指定した名前を持つ DBStatement オブジェクトがありません。

Transaction メソッド

機能

RegisterTransactions メソッドで関連付けた DBTransaction オブジェクトのポインタを取得します。

形式

```
DBTransaction *Transaction (void)
```

引数

なし

戻り値

データ型：DBTransaction*

DBTransaction オブジェクトのポインタ。

機能詳細

RegisterTransactions メソッドで関連付けた DBTransaction オブジェクトのポインタを取得します。関連付けられていない場合は，NULL が返ります。

発生する例外

なし

WaitForDataSource メソッド

機能

DBConnection オブジェクトで要求した実行待ち，及び実行中の非同期処理がすべて終了するまで待ちます。非同期処理中の同期を取りたい場合に利用できます。

形式

```
BOOLEAN WaitForDataSource(UINT32 dwWaitTime = DBR_INFINITE)
```

引数

dwWaitTime

非同期処理の終了を待つ最大時間（単位はミリ秒），又は DBR_INFINITE を指定します。

指定した時間内に非同期処理が終了した場合，又は非同期実行中の SQL がない場合は，TRUE が返ります。指定した値を経過しても非同期処理が終了しない場合，FALSE が返ります。

DBR_INFINITE を指定した場合はタイムアウト時間を設定しません。非同期処理がすべて終了するまで待ち続けます。

戻り値

データ型：BOOLEAN

TRUE：すべての非同期処理が終了しました。

FALSE：タイムアウト時間が経過しました。

機能詳細

非同期処理の実行待ち及び実行中の SQL が，終了するのを待ちます。

なお，各クラスの WaitForDataSource メソッドと，このクラスの WaitForDataSource メソッドの違いは，前者が WaitForDataSource メソッドを実行したオブジェクトの非同期処理の終了を待つのに対して，後者は DBConnection オブジェクトで生成したすべてのオブジェクトの非同期処理の終了を待ちます（対象となるコネクション内のすべての非同期処理が終わるのを待ちます）。

同期実行時はすぐに TRUE を返します。

発生する例外

なし

5.6　DBStatement クラスの詳細

SQL 文の実行を管理します。このクラスは，SQL 文の実行やその結果の取得などのメソッドを提供するクラスです。SQL 文の実行でも，?パラメタ（プレースホルダ）を利用して動的に検索条件を変更したい場合は DBPreparedStatement クラスを使います。

検索

DBStatement オブジェクトを生成した後，SELECT 文を指定した Execute メソッドを呼び出します。次に，検索結果を取得するために，GetResultSet メソッドを呼び出して，DBResultSet オブジェクトを取得します。

検索以外

DBStatement オブジェクトを生成した後，SQL 文を指定した Execute メソッドを呼び出します。

機 能	メソッド名
SQL 文の情報をデータベースに通知します。	Execute
SQL 文の非同期実行時のエラー情報を得るために, DBSQLCA オブジェクトへのポインタを取得します。	GetErrorStatus
検索結果のフィールドの数を取得します。	GetFieldCount
SetMaxFieldSize メソッドで設定したデータ長を取得します。	GetMaxFieldSize
ResultSet に検索できるレコード数の最大値を取得します。	GetMaxRows
DBStatement オブジェクトの名前を取得します。	GetName
SQL 文を実行し，検索したレコードから ResultSet オブジェクトを生成します。	GetResultSet
DBResultSetMetaData オブジェクトを生成します。	GetResultSetMetaData
SQL 文を使って更新，追加，又は削除したレコード数を取得します。	GetUpdateRows
DBStatement オブジェクトに非同期実行中（又は実行待ち）のステートメントがあるかどうかを確認します。	InExecute
対象となるオブジェクトを生成した DBConnection オブジェクトのポインタを取得します。	Parent
DBStatement オブジェクト(自分自身)を削除します。	Remove
DBResultSet オブジェクトを削除します。	RemoveResultSet
アプリケーションで受け取るフィールドの長さを設定します。	SetMaxFieldSize
ResultSet に検索するレコード数の最大値を指定します。	SetMaxRows
DBResultSet オブジェクトの生成オプションを指定します。	SetResultSetType
DBStatement オブジェクトで要求した実行待ち, 及び実行中の非同期処理が終了するまで待ちます。	WaitForDataSource

Execute メソッド

機能

引数で指定した SQL 文の情報をデータベースに通知します。

SELECT 文によって検索結果を取得する場合，Execute メソッドを呼び出した後，GetResultSet メソッドを呼び出して DBResultSet オブジェクトを生成します。

形式

```
void Execute (LPCTSTR lpctStatement,
              UINT16 swWait = LOCK_OPT_DEFAULT)      throw DBSQLCA
```

引数

lpctStatement

SQL 文を指定します。

swWait

実行する SQL 単位に，検索対象のレコードがほかのトランザクションによってロックされていた場合の動作を指定します。

SQL/K の場合は，どの値を指定しても，ロックが解除されるまで待ち状態になります。

XDM/SD の場合は，どの値を指定しても，ロックが解除されるまで待たないで，すぐにエラーを返します。

次の値のうち，どれか一つを指定します。

- LOCK_OPT_DEFAULT
  DBConnection オブジェクトの設定を引き継ぎます。（DBDriver クラスの Connect メソッドの引数 swWait に指定した値）
- LOCK_OPT_NOWAIT，又は LOCK_OPT_WITH_ROLLBACK
  ロックの解除を待たないで，すぐにエラーを返します。HiRDB 又は XDM/RD の場合は同時にロールバックを実行します。それ以外の DBMS の場合は，ロールバックは実行しません。
- LOCK_OPT_WITHOUT_ROLLBACK
  ロックの解除を待たないで，すぐにエラーを返します。ロールバックは実行しません。
  ただし，現在のバージョンでは，XDM/RD ではロールバックを実行します。また，HiRDB の DELETE,INSERT,UPDATE 文では，このオプションは無効で，LOCK_OPY_WAIT と同じ動作になります。
- LOCK_OPT_WAIT
  ロックが解除されるまで待ちます。

戻り値

なし

機能詳細

引数 lpctStatement で指定された SQL 文の情報をデータベースに通知します。また，その SQL 文ごとに排他エラーの処理形態を変更できます。

このメソッドは，非同期実行可能なメソッドです。(SELECT 文以外の実行時)

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_ERROR_DAB_ACCESS_ERROR
　DABroker でエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR
　システムコールでエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_TIMEOUT
　タイムアウトが発生したため制御を戻します。

DB_DRV_ERROR_INVALID_SQL_EXCLUSIVE
　WITH 句で始まる SQL 文の指定時には，DBResultSet オブジェクトの生成オプションを TYPE_EXCLUSIVE 以外に設定してください。

GetErrorStatus メソッド

機能

SQL 文の非同期実行時のエラー情報を得るために，DBSQLCA オブジェクトへのポインタを取得します。

形式

```
DBSQLCA *GetErrorStatus(void)
```

引数

なし

戻り値

データ型：DBSQLCA*

DBSQLCA オブジェクトへのポインタ。

機能詳細

DBSQLCA オブジェクトへのポインタを取得します。

非同期処理のときは，エラーが発生しても直にエラーを取得できません。このため，ユーザは任意の時点でエラーを確認する必要があります。

非同期実行時にエラーが発生した場合，DBSQLCA オブジェクトにエラー情報が設定されます。

DBSQLCA オブジェクトでは，非同期実行中に発生したエラー情報を最大 100 回分保存できます。エラー情報は，新しいエラー情報から 100 回分保存されるため，エラー情報を確認したあとは DBSQLCA クラスの Delete メソッドを呼び出して，不要なエラー情報をクリアしておいてください。

発生する例外

なし

GetFieldCount メソッド

機能

検索結果のフィールドの数を取得します。

形式

```
UINT32 GetFieldCount(void)
```

引数

なし

戻り値

データ型：UINT32

フィールドの個数。

機能詳細

Execute メソッドで指定された SELECT 文の，検索結果中に含まれるフィールドの個数を取得します。

GetFieldCount メソッドを呼び出す前に，Execute メソッドを呼び出しておく必要があります。

発生する例外

なし

GetMaxFieldSize メソッド

機能

SetMaxFieldSize メソッドで設定したデータ長を取得します。

形式

フィールド名で指定する場合

```
UINT32 GetMaxFieldSize(LPCTSTR lpctFieldName=NULL) throw DBSQLCA
```

インデクス番号で指定する場合

```
UINT32 GetMaxFieldSize(UINT32 dwIndex) throw DBSQLCA
```

引数

lpctFieldName

フィールドの名称，又は NULL を指定します。

NULL が指定された場合，すべてのフィールド長の合計値をバイト数で取得します。

dwIndex

1 から始まるフィールドの番号を指定します。

戻り値

データ型：UINT32

SetMaxFieldSize メソッドで設定したデータ長。

機能詳細

SetMaxFieldSize メソッドで設定したデータの長さを取得します。データベース中のフィールドの定義長ではないことに注意してください。

このメソッドを呼び出す前に，Execute メソッドを呼び出しておく必要があります。

不定長のフィールドの場合は，0 を返します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_BEFORE_EXECUTE
　Execute が実行されていません。

DB_ERROR_NOT_FOUND
　指定されたフィールドがありません。

GetMaxRows メソッド

機能

ResultSet に検索できるレコード数の最大値を取得します。

形式

```
UINT32 GetMaxRows(void)
```

引数

なし

戻り値

データ型：UINT32

SetMaxRows メソッドで設定した，レコード数の最大値を取得します。

機能詳細

SetMaxRows メソッドで設定した，DBResultSet オブジェクトで取得できるレコード数の最大値を取得します。SetMaxRows メソッドを一度も呼び出していない場合はデフォルト値の 100 を返します。

発生する例外

なし

GetName メソッド

機能

DBStatement オブジェクトの名前を取得します。

形式

```
LPCTSTR GetName(void)
```

引数

なし

戻り値

データ型：LPCTSTR

DBStatement オブジェクトの名前。

機能詳細

DBStatement オブジェクトの名前を取得します。DBStatement オブジェクトの名称は，DBConnection クラスの CreateStatement メソッドの引数から生成されます。

発生する例外

なし

GetResultSet メソッド

機能

SQL 文を実行し，検索したレコードから ResultSet オブジェクトを生成します。

形式

```
DBResultSet *GetResultSet (void) throw DBSQLCA
```

引数

なし

戻り値

データ型：DBResultSet*

DBResultSet オブジェクトへのポインタ。

機能詳細

Execute メソッドで指定された SELECT 文の検索結果を取得します。

データベースから SetMaxRows メソッドで指定されたレコード数分のレコードを検索後，DBResultSet オブジェクトを生成し，生成したオブジェクトへのポインタを返します。

Execute メソッドを実行していない場合や，Execute メソッドで実行した SQL が SELECT 文でない場合は NULL が返ります。

同じ SQL に対する 2 度目以降の GetResultSet の呼び出しは，既にある DBResultSet オブジェクトへのポインタが返ります。

BLOB 型データを扱う場合の制限事項については，SetResultSetType メソッド，及び「3.3.4 BLOB 型データの取得方法についての制限」を参照してください。

このメソッドは，非同期実行可能なメソッドです。

DBResultSet オブジェクトのポインタの有効期間

GetResultSet メソッドを呼び出して取得した DBResultSet オブジェクトのポインタは，次に Execute メソッドを実行した時点で無効になります。このため，Execute メソッド実行後に検索結果を取得したい場合は，必ず GetResultSet メソッドを実行して新しい DBResulSet オブジェクトのポインタを取得してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_ERROR_DAB_ACCESS_ERROR
　DABroker でエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR
　システムコールでエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_TIMEOUT
　タイムアウトが発生したため制御を戻します。

DB_DRV_ERROR_REQUEST_RESULTSET_ROWS
　ResultSet を生成するために指定されたレコード数が不正です。SetMaxRows メソッドで正しいレコード数を指定してください。

GetResultSetMetaData メソッド

機能

DBResultSetMetaData オブジェクトを生成します。

DBResultSetMetaData オブジェクトからフィールドの名称，数，属性などが取得できます。

形式

```
DBResultSetMetaData *GetResultSetMetaData(void) throw DBSQLCA
```

引数

なし

戻り値

データ型：DBResultSetMetaData*

DBResultSetMetaData オブジェクトへのポインタ。

機能詳細

Execute メソッドで指定された SELECT 文の検索結果に対する情報を取得します。

DBResultSetMetaData オブジェクトを生成し，生成したオブジェクトへのポインタを返します。取得したポインタは，次に Execute メソッドが呼ばれた時点で無効になります。

また，Execute メソッドで実行した SQL が SELECT 文でない場合は NULL を返します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

GetUpdateRows メソッド

機能

SQL 文を使って更新，追加，又は削除したレコード数を取得します。

形式

```
UINT32 GetUpdateRows(void)
```

引数

なし

戻り値

データ型：UINT32

処理されたレコード件数を取得します。

機能詳細

Execute メソッドで指定した UPDATE，DELETE，INSERT 文によって処理されたレコード数を取得します。

発生する例外

なし

InExecute メソッド

機能

DBStatement オブジェクトに非同期実行中（又は実行待ち）のステートメントがあるかどうかを確認します。

形式

```
BOOLEAN  InExecute(void)
```

引数

なし

戻り値

データ型：BOOLEAN

TRUE：ステートメントが非同期実行中(又は実行待ち)です。

FALSE：非同期実行中(又は実行待ち)ではありません。

機能詳細

InExecute メソッドを実行したオブジェクト内のステートメントが非同期実行中（又は実行待ち）かどうかを確認します。

非同期実行中（又は実行待ち）の場合は TRUE を，非同期実行中(又は実行待ち）でない場合は FALSE を返します。

同期実行接続時は FALSE を返します。

コネクション内のすべてのステートメントを対象に非同期実行中かどうかを確認するには，InWaitForDataSource メソッドを呼び出します(DBConnection クラス)。

発生する例外

なし

Parent メソッド

機能

対象となるオブジェクトを生成した DBConnection オブジェクトのポインタを取得します。

形式

```
LPOBJECT Parent(void)
```

引数

なし

戻り値

データ型：LPOBJECT

対象となるオブジェクトを生成した DBConnection オブジェクトへのポインタ。

機能詳細

なし

発生する例外

なし

Remove メソッド

機能

DBStatement オブジェクト(自分自身)を削除します。

形式

```
void Remove(void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

DBStatement オブジェクト(自分自身)を削除します。

非同期処理中に実行待ち，及び実行中のステートメントがある場合は削除できません。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中ステートメントがあります。

RemoveResultSet メソッド

機能

DBResultSet オブジェクトを削除します。

形式

```
void RemoveResultSet(void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

GetResultSet メソッドで生成した DBResultSet オブジェクトを削除します。

非同期処理中に実行待ち，又は実行中のステートメントがある場合は削除できません。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行中です。

SetMaxFieldSize メソッド

機能

アプリケーションで受け取るフィールドの長さを設定します。

形式

フィールド名で指定する場合

```
void SetMaxFieldSize(LPCTSTR lpctFieldName,  UINT32 dwMaxSize=0)
                                                    throw DBSQLCA
```

インデクス番号で指定する場合

```
void SetMaxFieldSize(UINT32 dwIndex, UINT32 dwMaxSize=0)
                                                    throw DBSQLCA
```

引数

lpctFieldName

フィールド名を指定します。

dwIndex

1 から始まるフィールドの番号を指定します。

dwMaxSize

アプリケーションで受け取るフィールドの長さを, 0 以上のバイト数で指定します。この引数で指定された値が，データベースでの定義長よりも大きい場合，引数で指定した値は無視されます。

0 を指定すると，データベース中の定義長が仮定されます。

戻り値

なし

機能詳細

アプリケーションで受け取るフィールドの長さをバイト数で指定します。アプリケーションで必要とするデータがフィールド値全体でなく, フィールド値の一部である場合に利用します。データベース中のフィールドの定義長ではないことに注意してください。

このメソッドを呼び出す前に，Execute メソッドを呼び出しておく必要があります。

指定したフィールドが繰り返し列の場合は，設定した値は無視されます。繰り返し列に対しては，取得するフィールド長は指定できません。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_BEFORE_EXECUTE
　Execute が実行されていません。

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数 dwIndex が 1 よりも小さいです。

DB_ERROR_CANNOT_BE_NULL
　引数 lpctFieldName に NULL を指定しています。

DB_DRV_ERROR_INVALID_ARGUMENT
　指定した引数の範囲が不正です。

DB_DRV_ERROR_INVALID_MAX_SIZE
　バッファの最大長の指定値が不正です。

SetMaxRows メソッド

機能

ResultSet に検索するレコード数の最大値を指定します。

形式

```
void SetMaxRows(UINT32 dwMaxSize=MAX_ROWS_DEFAULT) throw DBSQLCA
```

引数

dwMaxSize

ResultSet に検索するレコード数の最大値を指定します。必ず 1 以上を指定してください。

システムデフォルト値は，MAX_ROWS_DEFAULT です(=100 が仮定されます)。

SQL/K の場合，指定できる範囲は 4096 レコード以下です。

戻り値

なし

機能詳細

DBResultSet オブジェクトで取得できるレコード数の最大値を指定します。

一度の読み込み（GetResultSet メソッド，又は PageNext メソッドの呼び出し）で，データベースから取得するレコード数の最大値を設定します。

指定できるレコード数の範囲

指定できるレコード数の最大値は使用している DBMS によって異なります。各 DBMS ごとに，指定できるレコード数の最大値を次に示します。

- ORACLE を使用している場合：32767 レコード以下
- ORACLE 以外の DBMS の場合：4096 レコード以下

1 レコードのフィールド数は，DBResultSetMetaData オブジェクトの GetColumnCount メソッドを呼び出して取得できます。

更新可能な DBResultSet オブジェクトでの扱い

DBResultSet オブジェクトが更新可能なオブジェクトとして生成された場合(SetResultSetType メソッドの引数 swType で TYPE_EXCLUSIVE を指定した場合)，SetMaxRows メソッドによる指定は無視され，常に 1 が仮定されます。更新可能な DBResultSet は常に 1 レコードだけを読み込みます。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数 dwMaxSize が 1 より小さい値です。

SetResultSetType メソッド

機能

DBResultSet オブジェクトの生成オプションを指定します。

DBResultSet オブジェクトを生成するときに，参照専用，又は更新可能のどちらで生成するか選択します。検索レコードに対するロック方法を指定します。

検索時にバッファを幾つ使うかも指定できます。

また，BLOB 型データを扱う場合，その取得方法についても指定します。

形式

```
void SetResultSetType (UINT16
                       swType = TYPE_EXCLUSIVE|BUFFER_TYPE_SINGLE,
                       DBR_BLOB_TYPE nBLOBType = TYPE_BLOB_MEMORY,
                       LPCTSTR lpctBLOBFileName = NULL)
                                              throw DBSQLCA
```

引数

swType

次の三つの値を指定します。

- 検索したレコードに対するロック方法
- 検索時に使用するバッファ数
- VARCHAR データを DBR_BINARY 型で取得した場合の Length メンバ，RealLength メンバの意味

指定する値は，上記三つの値の論理和になります。

このメソッドを実行しなかった場合の動作は，TYPE_EXCLUSIVE | BUFFER_TYPE_SINGLE | VARCHAR_LENGTH_DEF を指定した場合と同じになります。

検索したレコードに対するロック方法

次に示す TYPE_EXCLUSIVE，TYPE_EXCLUSIVE2，TYPE_NONE，TYPE_WAIT，TYPE_NOWAIT，TYPE_SHARED のどれかの値で，検索したレコードに対するロック方法を指定します。

- TYPE_EXCLUSIVE 又は TYPE_EXCLUSIVE2

  DBResultSet オブジェクトからレコードの追加，更新，削除をする場合は，このオプションを指定します。検索したレコードに排他ロックが掛かり，他のトランザクションからの参照・更新を制限できます。実際に排他ロックが掛かるのは，このメソッドを実行した時点からです。

  このオプションで生成した DBResultSet オブジェクトでは，一度に複数のレコードを読み込めないため，SetMaxRows で指定したレコード数は無視されます。また，検索条件に一致する次のレコードをアクセスするには，Next メソッドを使用します。TYPE_EXCLUSIVE，又は TYPE_EXCLUSIVE2 で生成したオブジェクトの場合，カーソル制御のメソッドとしては Next メソッドだけが指定できます。Next メソッドについては，「5.7 DBResultSet クラスの詳細」を参照してください。

なお，この引数の値は，実行時に SQL 文のオプション文字列に変換されて実行されます。DBMS 別に，どのオプション文字列へ変換されて実行されるのかを次に示します。なお，HiRDB 以外の DBMS では TYPE_EXCLUSIVE と TYPE_EXCLUSIVE2 で同じオプション文字列へ変換されて実行されます。

- ORACLE：FOR UPDATE
- SQL Anywhere,Adaptive Server Anywhere：オプション文字列は付加しません。
- HiRDB
  TYPE_EXCLUSIVE：FOR UPDATE
  TYPE_EXCLUSIVE2：WITH EXCLUSIVE LOCK FOR UPDATE
- XDM/RD：WITH EXCLUSIVE LOCK FOR UPDATE
- SQL Server：(テーブル名の後に)UPDLOCK
- SQL/K：FOR UPDATE
  詳細については，マニュアル「SQL/K」を参照してください。
- XDM/SD：LOCK SU

- TYPE_NONE
  参照専用の DBResultSet オブジェクトを生成します。検索するレコードに対して，参照ロックを掛けます。
  このオプションで生成した DBResultSet オブジェクトでは，複数レコードを読み込めます。
  なお，この引数の値は，実行時に SQL 文のオプション文字列に変換されて実行されます。DBMS 別に，どのオプション文字列へ変換されて実行されるのかを次に示します。
  - ORACLE：付加しません。
  - SQL Anywhere，Adaptive Server Anywhere：付加しません。
  - HiRDB：付加しません。
  - XDM/RD：付加しません。
  - SQL Server：付加しません。
  - SQL/K：付加しません。
  - XDM/SD：付加しません(SDEXCLUSIVE 値が仮定されます)。
    SDEXCLUSIVE 値の詳細については，マニュアル「Database Connection Server」のコントロール空間起動制御文を参照してください。
- TYPE_WAIT(HiRDB，XDM/RD，SQL Server，SQL/K，XDM/SD の場合だけ有効です)
  参照専用の DBResultSet オブジェクトを生成します。
  検索したレコードに対して参照ロックを掛けますが，見終わった行から排他制御を解除します。このため，DABroker for C++では，ResultSet にレコードを読み込んだ時点，つまり，GetResultSet メソッド，又は PageNext メソッドが完了した時点で，ロックが解除されています。
  このオプションで生成した DBResultSet オブジェクトでは，複数レコードを読み込めます。
  なお，この引数の値は，実行時に SQL 文のオプション文字列に変換されて実行されます。DBMS 別に，どのオプション文字列へ変換されて実行されるのかを次に示します。
  - HiRDB：WITHOUT LOCK WAIT
  - XDM/RD：WITHOUT LOCK WAIT
  - SQL Server：(テーブル名の後に)HOLDLOCK
  - SQL/K：付加しません。

- XDM/SD：LOCK SR

- TYPE_NOWAIT(HiRDB，XDM/RD，SQL Server，SQL/K，XDM/SD の場合だけ有効です)
  参照専用の DBResultSet オブジェクトを生成します。該当するレコードをほかのトランザクションが更新中だったり，TYPE_EXCLUSIVE で検索している場合でも読み込めます。排他制御を全くしません。
  このオプションで生成した DBResultSet オブジェクトでは，複数レコードを読み込めます。
  なお，この引数の値は，実行時に SQL 文のオプション文字列に変換されて実行されます。DBMS 別に，どのオプション文字列へ変換されて実行されるのかを次に示します。
  - HiRDB：WITHOUT LOCK NOWAIT
  - XDM/RD：WITHOUT LOCK NOWAIT
  - SQL Server：(テーブル名の後に)NOLOCK
  - SQL/K：付加しません。
  - XDM/SD：LOCK NR
- TYPE_SHARED(HiRDB，XDM/RD，SQL/K，XDM/SD の場合だけ有効です)
  検索するレコードに共有ロックを掛けます。共有ロックの掛かったレコードは，ほかのトランザクションから参照できますが，更新はできません。
  - HiRDB：WITH SHARE LOCK
  - XDM/RD：WITH SHARE LOCK
  - SQL/K：付加しません。
  - XDM/SD：LOCK SR

検索時に使用するバッファ数

次に示す BUFFER_TYPE_SINGLE，又は BUFFER_TYPE_DOUBLE のどちらかの値で，検索時に使用するバッファ数を指定します。指定を省略した場合は，BUFFER_TYPE_SINGLE が仮定されます。
なお，バッファの利用方法については，「3.3.3 検索性能の向上策」を参照してください。

- BUFFER_TYPE_SINGLE
  検索結果を保持するバッファを一つだけ使います。
- BUFFER_TYPE_DOUBLE
  検索結果を保持するバッファを二つ使います。

VARCHAR データを DBR_BINARY 型で取得したときの Length メンバ，RealLength メンバの意味

次に示す値のどれか一つを指定します。

- VARCHAR_LENGTH_DEF
  - GetField メソッドの形式 2 を使用した場合，又は形式 3 を使用してもデータの切り捨てが発生しなかった場合
    Length：カラムの定義長※
    RealLength：0
  - GetField メソッドの形式 3 を使用してデータの切り捨てが発生した場合
    Length：切り捨てたデータ長
    RealLength：カラムの定義長※
    注※　SetMaxFieldSize メソッドで取得データ長を指定している場合は，「カラムの定義長」ではなく「SetMaxFieldSize メソッドで指定したサイズ」となります。

- VARCHAR_LENGTH_REAL
  - GetField メソッドの形式 2 を使用した場合，又は形式 3 を使用してもデータの切り捨てが発生しなかった場合

    Length：実際のデータ長※

    RealLength：0
  - GetField メソッドの形式 3 を使用してデータの切り捨てが発生した場合

    Length：切り捨てたデータ長

    RealLength：実際のデータ長※

    注※　SetMaxFieldSize メソッドでカラムの定義長よりも小さいサイズを指定している場合は，データベースから取得したデータは実際のデータよりも小さい場合があるため，「実際のデータ長」ではなく「取得データ長」となります。

nBLOBType

BLOB 型データの取得方法として，次のどちらかを指定します。

- TYPE_BLOB_MEMORY

  メモリ上に一括して取得します。

  ResultSet のレコード数が 1 の場合，BLOB 型データをすべて取得します。

  ResultSet のレコード数が複数の場合，先頭から 32K バイトまでを取得します。
- TYPE_BLOB_FILE

  ファイルを経由して取得します。

  引数 lpctBLOBFileName に指定された名称が，BLOB 型データを格納するファイルのプレフィックスとして使用されます。

  なお，引数 swType で BUFFER_TYPE_DOUBLE を指定している場合，検索の途中で終了しても先読みしている ResultSet 分のファイルは作成されます。

lpctBLOBFileName

BLOB 型データを格納するファイルのプレフィックスを指定します。

引数 nBLOBType で TYPE_BLOB_FILE を指定した場合，BLOB 型データを格納するファイルのプレフィックスには，この引数で指定した名称が使われます。TYPE_BLOB_FILE 以外の値を指定した場合は，この引数での指定は無視されます。

BLOB 型データを格納したファイル名称は，「プレフィックス+レコード番号+フィールド番号」で表されます。NULL が指定された場合，「DBConnection の名称+DBStatement の名称」が仮定されます。

戻り値

なし

機能詳細

検索レコードに対するロック方法，検索時にバッファを幾つ使うか，及び BLOB 型データの取得方法について指定します。

指定した各引数の値は，次に GetResultSet メソッドを実行したときから有効になります。

ファイル上で BLOB 型データを扱う場合の制限事項

データの格納されているファイルは，DABroker が終了しても削除されません。アプリケーション自身で削除してください。

作成されたファイルのアクセス権限は，アプリケーションの実行ユーザ及びデフォルトの権限に依存します。例えば，HP-UX 上で"user1"が実行しているアプリケーションで，デフォルト umask が 0666 の場合，取得した BLOB 型データが格納されるファイルも，"user1"の所有する 0666 のアクセス権限が付与されたファイルとして生成されます。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

WaitForDataSource メソッド

機能

DBStatement オブジェクトで要求した実行待ち，及び実行中の非同期処理が終了するまで待ちます。非同期処理中に同期を取りたい場合に利用できます。

形式

```
BOOLEAN WaitForDataSource(UINT32 swWaitTime = DBR_INFINITE)
```

引数

swWaitTime

非同期処理の終了を待つための最大時間（単位：ミリ秒），又は DBR_INFINITE を指定します。

指定した時間内に非同期処理が終了した場合，TRUE が返ります。指定した時間を経過しても非同期処理が終了しない場合，FALSE が返ります。非同期実行中の SQL がない場合は，TRUE が返ります。

DBR_INFINITE を指定した場合はタイムアウト時間を設定しません。非同期処理が終了するまで待ち続けます。

同期実行時はすぐに TRUE を返します。

戻り値

データ型：BOOLEAN

TRUE：すべての非同期処理が終了しました。

FALSE：タイムアウト時間が経過しました。

機能詳細

DBStatement オブジェクトで要求した実行待ち，及び実行中の非同期処理(SQL)が，終了するのを待ちます。

DBConnection クラスの WaitForDataSource メソッドとの違い

DBStatement オブジェクトの WaitForDataSource メソッドでは, WaitForDataSource メソッドを実行した DBStatement オブジェクトの非同期実行処理が終了するまで待ちます。ほかのオブジェクトの非同期実行処理については待ちません。そのため，ほかのオブジェクトの非同期実行処理が実行中でも，WaitForDataSource メソッドを呼び出したオブジェクトの非同期実行処理が終了すれば，WaitForDataSource メソッドは TRUE を返します。

同じ DBConnection オブジェクトで実行しているすべての非同期実行処理が終了するのを待ちたい場合は，DBConnection クラスの WaitForDataSource メソッドを実行してください。

発生する例外

なし

5.7　DBResultSet クラスの詳細

検索結果を管理するには，DBResultSet クラスを利用します。このクラスは，検索結果の取得，そのためのカーソルの移動，更新などのメソッドを提供するクラスです。

DBResultSet オブジェクトが生成されると，フィールド値を参照するための GetField メソッドや，更新のための SetField メソッドではレコード単位にデータを扱います。どのレコードを処理対象にするかはカーソルを使います。カーソルの位置付いたレコードをカレントレコードと呼び，複数のレコードを処理するためにはカーソルを順次移動して処理対象を変更していきます。

参照専用

DBStatement オブジェクト，DBPreparedStatement オブジェクト又は DBCallableStatement オブジェクトの SetResultSetType メソッドで TYPE_NONE などを指定して生成します。

参照専用の DBResultSet オブジェクトは，一度に複数のレコードを読み込めます。1 度に読み込める最大レコードの数は，SetMaxRows メソッド(DBStatement オブジェクト，DBPreparedStatement オブジェクト，又は DBCallableStatement オブジェクト)で指定できます。また，読み込んだレコードは，GetField メソッドを呼び出して参照します。

更新可能

DBStatement オブジェクト，DBPreparedStatement オブジェクト，又は DBCallableStatement オブジェクトの SetResultSetType メソッドで TYPE_EXCLUSIVE を指定して生成します（検索するレコードに対してロックをかけます)。

更新可能な DBResultSet オブジェクトは，一度に一つのレコードしか読み込めませんが，読み込んだレコードに対して，Edit メソッドや Update メソッドなどの更新用のメソッドを使って直接データを変更できます。このため，個別に UPDATE 文などの SQL 文を使わなくても，検索したレコードを直接更新するアプリケーションを作成できます。

機 能	メソッド名
カーソルを ResultSet の先頭から n 番目のレコードに移動します。	Absolute
カーソルを ResultSet の最後のレコードへ移動します。	Bottom
カレントレコードを削除します。	Delete
カレントレコードを更新するための準備をします。	Edit
フィールド名称に対応するフィールドのインデクスを取得します。	FindColumn
検索したすべてのレコードの先頭から数えた，カレントレコードの位置を取得します。	GetCurrent
ResultSet の先頭レコードから数えた，カレントレコードの位置を取得します。	GetCurrentOfResultSet
SQL 文の非同期実行時のエラー情報を得るために，DBSQLCA オブジェクトへのポインタを取得します。	GetErrorStatus
カレントレコードのフィールドの値を取得します。	GetField
DBResultSetMetaData オブジェクトを生成します。	GetMetaData
ResultSet のレコードの総数を取得します。	GetRowCount
DBResultSet オブジェクトに非同期実行中（又は実行待ち）のステートメントがあるかどうかを確認します。	InExecute

機 能	メソッド名
カーソルの位置が，ResultSet のレコードの最後を超えたかどうかを確認します。	IsEOF
フィールドの値が NULL かどうかを確認します。	IsNull
カーソルを次のレコードへ移動します。	Next
カーソルを次の ResultSet の先頭へ移動します。	PageNext
対象となるオブジェクトを生成した DBStatement オブジェクト，DBPreparedStatement オブジェクト，又は DBCallableStatement オブジェクトのポインタを取得します。	Parent
カーソルを前のレコードへ移動します。	Previous
先頭のレコードから再読み込みをします。	Refresh
カーソルを現在のレコードの位置から n 個分移動します。	Relative
DBResultSet オブジェクト(自分自身)を削除します。	Remove
指定されたフィールドにデータを設定します。	SetField
指定されたフィールドに NULL を設定します。	SetNull
カーソルを ResultSet の先頭のレコードへ移動します。	Top
フィールドの更新結果をデータベースへ通知します。	Update
DBResultSet オブジェクトで要求した実行待ち，及び実行中の非同期処理が終了するまで待ちます。	WaitForDataSource

Absolute メソッド

機能

カーソルを ResultSet の先頭から n 番目のレコードに移動します。

形式

```
void Absolute(UINT32 dwCount) throw DBSQLCA
```

引数

dwCount

カーソルをどのレコードに位置付けるか，レコードのインデクスを指定します。

ResultSet の範囲を超えるような番号は指定できません。

戻り値

なし

機能詳細

カーソルを ResultSet の先頭から引数 dwCount で指定した番号のレコードに移動します。

カーソルがレコードの最後を超えている場合（IsEOF メソッドで TRUE が返る場合），このメソッドは呼び出せません。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_RESULTSET_NOT_EXIST
　非同期実行時に，ResultSet がまだ取得できていません。

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

DB_ERROR_OUT_OF_RESULTSET
　ResultSet の範囲を超えるような呼び出しです。

Bottom メソッド

機能

カーソルを ResultSet の最後のレコードへ移動します。

形式

```
void Bottom(void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

カーソルを ResultSet の最後のレコードへ移します。

カーソルがレコードの最後を超えている場合（IsEOF メソッドで TRUE が返る場合），このメソッドは呼び出せません。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_RESULTSET_NOT_EXIST
　非同期実行時に，ResultSet がまだ取得できていません。

DB_ERROR_OUT_OF_RESULTSET
　ResultSet の範囲を超えるような呼び出しです。

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

Delete メソッド

機能

カレントレコードを削除します。

形式

```
void Delete(void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

カレントレコードを削除します。

トランザクションを設定したアプリケーションの場合，コミットされるまでデータベースに反映されません。

このメソッドは，非同期実行可能なメソッドです。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_NOT_IN_EDIT
　Edit メソッドが実行されていません。

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行中です。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_ERROR_DAB_ACCESS_ERROR
　DABroker でエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR
　システムコールでエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_TIMEOUT
　タイムアウトが発生したため制御を戻します。

Edit メソッド

機能

カレントレコードを更新するための準備をします。

形式

```
void Edit(void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

カーソルを位置付けたレコードに対する更新の準備をします。

実際にカレントレコードを更新するためには，Edit メソッドを呼び出した後，SetField メソッドを呼び出してフィールドの値を更新します。必要なデータを更新した後に Update メソッドを呼び出すと，カレントレコードの更新情報はデータベースへ送信されます。

トランザクションを設定したアプリケーションの場合, 送信されたデータはコミットされるまでデータベースに反映されません。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_CURRENT_RECORD_DELETED
　カレントレコードが削除されています。

DB_ERROR_RESULTSET_NOT_EXIST
　非同期実行時に，検索結果がまだ取得できていません。

DB_ERROR_CANNOT_EDIT_LAST_RECORD
　カレントレコードがありません(検索結果の最後がカレントレコードの場合)。

DB_ERROR_CANT_UPDATE
　検索結果が読み込み専用です。

FindColumn メソッド

機能

フィールド名称に対応するフィールドのインデクスを取得します。

形式

```
UINT32 FindColumn(LPCTSTR lpctFieldName) throw DBSQLCA
```

引数

lpctFieldName

フィールド名を指定します。

なお，SELECT 文でフィールド演算や，COUNT などの関数を使用した場合，フィールド名はデータベースによって決められます。このような場合にフィールド名を調べるためには，DBResultSetMetaData オブジェクトの GetColumnName メソッドを呼び出します。

戻り値

データ型：UINT32

フィールドの番号を取得します。

機能詳細

指定されたフィールド名に対応したフィールド番号を取得します。

フィールド番号は，SQL 文中での出現順に割り当てられます。

引数に指定されたフィールド名が DBResultSet オブジェクト中で重複していた場合，FindColumn メソッドは最初に検索したフィールド番号（フィールド番号の最も若いもの）を返します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_RESULTSET_NOT_EXIST
　非同期実行時に，検索結果がまだ取得できていません。

DB_ERROR_NOT_FOUND
　指定したフィールドが見つかりません。

DB_ERROR_CANNOT_BE_NULL
　引数 lpctFieldName に NULL を指定しています。

GetCurrent メソッド

機能

検索したすべてのレコードの先頭から数えた，カレントレコードの位置を取得します。

形式

```
UINT32 GetCurrent(void) throw DBSQLCA
```

引数

なし

戻り値

データ型：UINT32

カレントレコードの先頭レコードからの位置を取得します。

機能詳細

検索したすべてのレコードの先頭を 1 として数えた，カレントレコードの位置を取得します。

レコードの終端（EOF）で GetCurrent メソッドを呼び出すと，検索レコード数+1 の値を返します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_RESULTSET_NOT_EXIST
　非同期実行時に，検索結果がまだ取得できていません。

GetCurrentOfResultSet メソッド

機能

ResultSet の先頭レコードから数えた，カレントレコードの位置を取得します。

形式

```
UINT32 GetCurrentOfResultSet(void) throw DBSQLCA
```

引数

なし

戻り値

データ型：UINT32

カレントレコードの ResultSet の先頭レコードからの位置を取得します。

機能詳細

ResultSet のレコードの先頭を 1 して数えた，カレントレコードの位置を取得します。

ResultSet が 0 レコードの場合も 1 を返します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_RESULTSET_NOT_EXIST
　非同期実行時に，ResultSet がまだ取得できていません。

GetErrorStatus メソッド

機能

SQL 文の非同期実行時のエラー情報を得るために，DBSQLCA オブジェクトへのポインタを取得します。

形式

```
DBSQLCA *GetErrorStatus(void)
```

引数

なし

戻り値

データ型：DBSQLCA*

DBSQLCA オブジェクトへのポインタ。

機能詳細

DBSQLCA オブジェクトへのポインタを取得します。

非同期処理のときは，エラーが発生しても直にエラーを取得できません。このため，ユーザは任意の時点でエラーを確認する必要があります。

非同期実行時にエラーが発生した場合，DBSQLCA オブジェクトにエラー情報が設定されます。

DBSQLCA オブジェクトでは，非同期実行中に発生したエラー情報を最大 100 回分保存できます。エラー情報は，新しいエラー情報から 100 回分保存されるため，エラー情報を確認したあとは DBSQLCA クラスの Delete メソッドを呼び出して，不要なエラー情報をクリアしておいてください。

発生する例外

なし

GetField メソッド

機能

カレントレコードのフィールドの値を取得します。

形式１：ユーザ確保領域へコピーする場合

インデクス番号指定

```
void GetField(UINT32 dwIndex, INT16  *psData) throw DBSQLCA
void GetField(UINT32 dwIndex, INT32  *plData) throw DBSQLCA
void GetField(UINT32 dwIndex, UINT16 *pswData) throw DBSQLCA
void GetField(UINT32 dwIndex, UINT32 *pdwData) throw DBSQLCA
void GetField(UINT32 dwIndex, SINGLE *psfData) throw DBSQLCA
void GetField(UINT32 dwIndex, DOUBLE *pdfData) throw DBSQLCA
void GetField(UINT32 dwIndex, DBR_DATETIME *pdtData)
                                                  throw DBSQLCA
void GetField(UINT32 ui32Index, DBRArrrayDataConstPtr*
                                     cparData)throw DBSQLCA
```

フィールド名指定

```
void GetField(LPCTSTR lpctFieldName, INT16  *psData)
                                                  throw DBSQLCA
void GetField(LPCTSTR lpctFieldName, INT32  *plData)
                                                  throw DBSQLCA
void GetField(LPCTSTR lpctFieldName, UINT16 *pswData)
                                                  throw DBSQLCA
void GetField(LPCTSTR lpctFieldName, UINT32 *pdwData)
                                                  throw DBSQLCA
void GetField(LPCTSTR lpctFieldName, SINGLE *psfData)
                                                  throw DBSQLCA
void GetField(LPCTSTR lpctFieldName, DOUBLE *pdfData)
                                                  throw DBSQLCA
void GetField(LPCTSTR lpctFieldName, DBR_DATETIME *pdtData)
                                                  throw DBSQLCA
void GetField(LPCTSTR lpctFieldName, DBRArrrayDataConstPtr*
                                    cparData) throw DBSQLCA
```

形式２：DABroker が用意した領域を参照する場合

インデクス番号指定

```
void GetField(UINT32 dwIndex, LPTSTR *lptData) throw DBSQLCA
void GetField(UINT32 dwIndex, DBR_BINARY *pblobData)
                                                  throw DBSQLCA
```

フィールド名指定

```
void GetField(LPCTSTR lpctFieldName, LPTSTR *lptData)
                                                  throw DBSQLCA
void GetField(LPCTSTR lpctFieldName, DBR_BINARY *pblobData)
                                                  throw DBSQLCA
```

形式３：ユーザ確保領域へコピーする場合

インデクス番号指定

```
UINT32 GetField(UINT32 dwIndex, LPTSTR lptData, UINT32 dwSize)
                                                  throw DBSQLCA
UINT32 GetField(UINT32 dwIndex, DBR_BINARY *pblobData,
                                 UINT32 dwSize) throw DBSQLCA
```

フィールド名指定

```
UINT32 GetField(LPCTSTR lpctFieldName, LPTSTR lptData,
                                 UINT32 dwSize) throw DBSQLCA
UINT32 GetField(LPCTSTR lpctFieldName, DBR_BINARY *pblobData,
                                 UINT32 dwSize) throw DBSQLCA
```

引数

dwIndex, ui32Index

1 から始まるフィールドの番号を指定します。指定するフィールドの番号は，SQL 文中での出現順に割り当てられます。

lpctFieldName

フィールド名を指定します。

なお，SELECT 文でフィールド演算や，COUNT などの関数を使用した場合，フィールド名はデータベースによって決められます。このような場合にフィールド名を調べるためには，DBResultSetMetaData オブジェクトの GetColumnName メソッドを呼び出します。取得するフィールド名は, 重複することもあります。このため，SELECT 文で JOIN を指定したときなどに，フィールド名が重複している可能性がある場合や，SELECT 文でのフィールド演算などでフィールド名が不定の場合は，フィールドの番号を使用してください。

*xxData（形式 1，3 の第 2 引数）

データをコピーする領域のポインタを指定します。

*xxData（形式 2 の第 2 引数）

ポインタ変数を指定します。

dwSize

コピー先の領域サイズをバイト数で指定します。

戻り値

データ型：UINT32

コピーした文字列の長さが返ります。フィールドの値を指定された型にキャストして取得します（形式 3 の場合）。

機能詳細

形式 1 の処理

フィールド値を，引数に指定されたポインタが指す領域にコピーします。

形式 2 の処理

DABroker が確保した領域にフィールド値をコピーし，その先頭アドレスをポインタ変数に設定します。このため，LPTSTR または DBR_BINARY の変数の確保だけで，フィールド値をコピーする領域を確保する必要はありません。

- フィールド値の有効期間
  フィールド値は，GetResultSet メソッドや Next メソッドなどで次の ResultSet 取得時まで，又は ResultSet を削除するまで有効です。前の ResultSet のフィールド値を残したい場合には，ユーザが確保した領域にデータを取得してください。

形式３の処理

- LPTSTR 型
  引数 lptData の指す領域は，引数 dwSize バイト分確保されている必要があります。
  - 引数 dwSize がデータベースから取得したデータの長さ以下の場合
    データベースのデータ型が可変長文字列型の場合，引数 dwSize-1 分のデータ（最後には NULL 終端文字が入る）がコピーされ，残りのデータは切り捨てられます。
    データベースのデータ型が固定長文字列の場合，引数 dwSize 分のデータ（最後に NULL 終端文字が入らない）がコピーされ，残りのデータは切り捨てられます。
  - 引数 dwSize がデータベースから取得したデータの長さよりも大きい場合
    文字列はすべてコピーします。固定長文字列の場合，残りの領域には NULL 終端文字が埋められます。
- DBR_BINARY 型
  フィールド値をコピーする領域（LPTSTR 型）を確保し，DBR_BINARY 型変数（構造体）の Data メンバに設定し，この変数を GetField メソッドの引数に指定します。引数 dwSize はこの領域のサイズを指定します。
  引数 dwSize がデータベースから取得したデータの長さ以下の場合，引数 dwSize 分のデータ（最後に NULL 終端文字が入らない）がコピーされ，残りのデータは切り捨てられます。引数 dwSize がデータベースから取得したデータの長さより大きい場合，データはすべてコピーされ，残りの領域には NULL 終端文字が埋められます。

データ変換について

データベースのデータ型と GetField の引数に指定したデータ型とが異なる場合は，値の変換できるものについては引数で指定したデータ型に変換して返します。

文字列から数値データ型への変換に失敗した場合は 0 を返します。データ型の変換規則については，「7.1 クラスライブラリで扱うデータ型と変換規則」の GetField，GetParam メソッドでのデータ型変換規則を参照してください。

NULL 値の扱い

NULL であるフィールドに対して GetField メソッドを呼び出すと，戻される値は意味のない値(0，空文字列，要素がすべて 0，空文字列からなる構造体，又は不定値)となります。

このため，NULL が格納されている可能性のあるフィールドでは，IsNull メソッドを呼び出して，NULL 値かどうかを確認してください。

DBR_BINARY 型で値を取得した場合

各メンバの値は次のようになります。

- SetMaxFieldSize メソッドで指定した長さよりも実データ長が長い場合
  Length：取得したデータ長
  RealLength：実データ長
  Data：Length の長さ分のデータ（データの切り捨てあり）
- SetMaxFieldSize メソッドで指定した長さよりも実データ長が短い，又は等しい場合
  Length：取得したデータ長
  RealLength：0

Data：Length の長さ分のデータ（データの切り捨てなし）

TYPE_BLOB_FILE について

SetResultSetType メソッドで，TYPE_BLOB_FILE を指定した場合，BLOB 型のフィールドに対する GetField メソッドの引数にはファイル名称が返ります。この場合，GetField メソッドの引数の型には，文字列と同じ LPTSTR を使います。

SQL/K の場合

データが設定されていない列を検索しても，GetField メソッドを実行すると，VOS K/FS のファイル定義で定義されたデータを返します。ファイル定義の詳細は，マニュアル「ファイル運用」を参照してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_RESULTSET_NOT_EXIST
　非同期実行時に，検索結果がまだ取得できていません。

DB_ERROR_NOT_FOUND
　引数 dwIndex の範囲が不正です。

DB_ERROR_CANNOT_BE_NULL
　引数 lpctFieldName，又は引数 xxData に NULL を指定しています。

DB_ERROR_DATA_TRUNCATED
　取得した値を指定した型に変換できません。

DB_DRV_ERROR_INVALID_ARGUMENT
　引数に指定したフィールド名は不正です。

DB_ERROR_CONVERT_ARRAY_TO_SCALAR
　引数 cparData で繰り返し列以外のフィールド値を取得しました，又は引数 cparData 以外の引数で繰り返し列の値を取得しようとしました。

GetMetaData メソッド

機能

DBResultSetMetaData オブジェクトを生成します。

DBResultSetMetaData オブジェクトから，ResultSet のフィールド数や，属性などが取得できます。

形式

```
DBResultSetMetaData *GetMetaData(void) throw DBSQLCA
```

引数

なし

戻り値

データ型：DBResultSetMetaData*

DBResultSetMetaData オブジェクトへのポインタ。

機能詳細

DBResultSetMetaData オブジェクトを生成し，生成したオブジェクトへのポインタを返します。

発生する例外

なし

GetRowCount メソッド

機能

ResultSet のレコードの総数を取得します。

形式

```
UINT32 GetRowCount(void) throw DBSQLCA
```

引数

なし

戻り値

データ型：UINT32

ResultSet 中のレコードの総数を取得します。

機能詳細

DBResultSet オブジェクト中に読み込まれたレコード数を取得します。

GetRowCount メソッドの返す値が GetMaxRows メソッドの返す値よりも少ない場合，データベース中のすべての検索結果が読み込まれたことを示します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_RESULTSET_NOT_EXIST
　非同期実行時に，検索結果がまだ取得できていません。

InExecute メソッド

機能

DBResultSet オブジェクトに非同期実行中（又は実行待ち）のステートメントがあるかどうかを確認します。

形式

```
BOOLEAN  InExecute(void)
```

引数

なし

戻り値

データ型：BOOLEAN

TRUE：ステートメントが非同期実行中(又は実行待ち)です。

FALSE：非同期実行中(又は実行待ち)ではありません。

機能詳細

InExecute メソッドを実行したオブジェクト内のステートメントが非同期実行中(又は実行待ち）かどうかを確認します。

非同期実行中(又は実行待ち）の場合は TRUE を，非同期実行中(又は実行待ち）でない場合は FALSE を返します。

同期実行接続時は FALSE を返します。

コネクション内のすべてのステートメントを対象に非同期実行中かどうかを確認するには，InWaitForDataSource メソッドを呼び出します(DBConnection クラス)。

発生する例外

なし

IsEOF メソッド

機能

カーソルの位置が，ResultSet のレコードの最後を超えたかどうかを確認します。

形式

```
BOOLEAN  IsEOF(void) throw DBSQLCA
```

引数

なし

戻り値

データ型：BOOLEAN

TRUE：カーソルの位置が，ResultSet のレコードの最後を超えました。

FALSE：カーソルの位置は，まだ ResultSet の途中のレコードにあります。

機能詳細

Next メソッドで次のレコードへカーソルを移動した後，まだレコードがあるかどうかを判断するときに使います。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_RESULTSET_NOT_EXIST
　非同期実行時に，検索結果がまだ取得できていません。

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

IsNull メソッド

機能

フィールドの値が NULL かどうかを確認します。

形式

インデクス番号で指定する場合

```
BOOLEAN  IsNull(UINT32   dwIndex) throw DBSQLCA
```

フィールド名称で指定する場合

```
BOOLEAN  IsNull(LPCTSTR lpctName) throw DBSQLCA
```

引数

dwIndex

1 から始まるフィールドの番号を指定します。

lpctName

フィールド名を指定します。

戻り値

データ型：BOOLEAN

TRUE：フィールドの値が NULL です。

FALSE：フィールドの値は NULL ではありません。

機能詳細

指定されたフィールドの値が NULL 値かどうかを確認します。

NULL は C 言語で使用する NULL ポインタの意味ではなく，データベースシステムでは「値がない」ことを意味します。

NULL であるフィールドに対して GetField メソッドを呼び出すと，戻される値は意味のない値（0，NULL 文字列，要素がすべて 0，NULL 文字列からなる構造体，又は不定値）となるため注意してください。

SQL/K の場合，データが設定されていない列を検索しても，IsNull メソッドの戻り値は常に FALSE です。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_RESULTSET_NOT_EXIST
　非同期実行時に，検索結果がまだ取得できていません。

DB_ERROR_NOT_FOUND
　引数 dwIndex の範囲が不正です。

DB_ERROR_CANNOT_BE_NULL
　引数 lpctName に NULL を指定しています。

DB_DRV_ERROR_INVALID_ARGUMENT
　引数に指定したフィールド名は不正です。

Next メソッド

機能

カーソルを次のレコードへ移動します。

形式

```
void Next(void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

次のレコードへカーソルを移します。

更新可能な ResultSet の場合，検索条件に一致する次のレコードを ResultSet に読み込みます。参照専用の ResultSet の場合，カーソルを次のレコードへ移します。ResultSet の最後のレコードにカーソルがある場合，Next メソッドが呼び出されると，次のレコードを読み込みます。

次のレコードを ResultSet に読み込む場合は，非同期実行可能です。単にカーソルを次のレコードに移す場合は，非同期に実行しません。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_RESULTSET_NOT_EXIST
　非同期実行時に，検索結果がまだ取得できていません。

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

DB_ERROR_OUT_OF_RANGE
　レコード終端を超えてメソッドが呼び出されました。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_ERROR_DAB_ACCESS_ERROR
　DABroker でエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR
　システムコールでエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_TIMEOUT
　タイムアウトが発生したため制御を戻します。

PageNext メソッド

機能

カーソルを次の ResultSet の先頭へ移動します。

形式

```
void PageNext(void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

次のレコード群(SetMaxRows で設定したレコード数)を読み込み，ResultSet 中のレコードを置き換えます。カーソルは次のレコード群の先頭のレコードに位置付けられます。

すべてのレコードが読み込まれている（IsEOF メソッドで TRUE が返る）ときにこのメソッドは呼び出せません。

このメソッドは，非同期実行可能なメソッドです。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_RESULTSET_NOT_EXIST
　非同期実行時に，検索結果がまだ取得できていません。

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

DB_ERROR_OUT_OF_RANGE
　レコード終端を超えてメソッドが呼び出されました。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_ERROR_DAB_ACCESS_ERROR
　DABroker でエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR
　システムコールでエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_TIMEOUT
　タイムアウトが発生したため制御を戻します。

Parent メソッド

機能

対象となるオブジェクトを生成した DBStatement オブジェクト，DBPreparedStatement，又は DBCallableStatement オブジェクトオブジェクトのポインタを取得します。

形式

```
LPOBJECT Parent(void)
```

引数

なし

戻り値

データ型：LPOBJECT

対象となるオブジェクトを生成した DBStatement オブジェクト，DBPreparedStatement オブジェクト，又は DBCallableStatement オブジェクトへのポインタ。

機能詳細

なし

発生する例外

なし

Previous メソッド

機能

カーソルを前のレコードへ移動します。

形式

```
void Previous(void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

カーソルを一つ前のレコードへ移します。

ResultSet の先頭で Previous メソッドは呼び出せません。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_RESULTSET_NOT_EXIST
　非同期実行時に，検索結果がまだ取得できていません。

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

DB_ERROR_OUT_OF_RESULTSET
　ResultSet の範囲を超えるような呼び出しです。

Refresh メソッド

機能

先頭レコードから再読み込みをします。

形式

```
void Refresh(void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

データベースからレコードを再検索して，先頭レコードから再読み込みをします。

DBResultSet オブジェクトが DBPreparedStatement オブジェクトから生成された場合，Refresh メソッドを呼び出す前に DBPreparedStatement オブジェクトの SetParam メソッドを呼び出して?パラメタの値を変更できます。この場合，Refresh メソッドによって再検索された結果は，新しく設定された?パラメタの値を反映したものになります。

例えば「商品番号」を?パラメタとして使用している場合，Refresh メソッドを呼び出す前に SetParam メソッドで新しい商品番号を設定しておくと，Refresh メソッドを呼び出した結果再検索されたデータは，新しい商品番号のデータとなります。

このメソッドは非同期実行可能なメソッドです。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_RESULTSET_NOT_EXIST
　非同期実行時に，検索結果がまだ取得できていません。

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_ERROR_DAB_ACCESS_ERROR
　DABroker でエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR
　システムコールでエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_TIMEOUT
　タイムアウトが発生したため制御を戻します。

Relative メソッド

機能

カーソルを現在のレコードの位置から n 個分移動します。

形式

```
void Relative(INT32  lCount) throw DBSQLCA
```

引数

lCount

指定した数分現在のレコードの位置からカーソルを移動します。

戻り値

なし

機能詳細

現在の位置から数えて，引数 lCount 番目のレコードにカーソルを移します。

正の値を指定した場合現在の位置より指定した値分後ろのレコードへ, 負の値を指定した場合は指定した値分前のレコードへカーソルを移動します。

すべてのレコードが読み込まれている（IsEOF メソッドで TRUE が返る）ときにこのメソッドは呼び出せません。

また，引数 lCount に検索結果の範囲を超えるような値は指定できません。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_RESULTSET_NOT_EXIST
　非同期実行時に，検索結果がまだ取得できていません。

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

DB_ERROR_OUT_OF_RESULTSET
　ResultSet の範囲を超えるような呼び出しです。

Remove メソッド

機能

DBResultSet オブジェクト(自分自身)を削除します。

形式

```
void Remove(void)
```

引数

なし

戻り値

なし

機能詳細

DBResultSet オブジェクト(自分自身)を削除します。

非同期処理中に実行待ち，又は実行中のステートメントがある場合は削除できません。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中ステートメントがあります。

DB_ERROR_RESULTSET_NOTEXIST
　非同期実行時に検索結果がまだ取得できません。

SetField メソッド

機能

指定されたフィールドにデータを設定します。

形式

インデクスで指定する場合

```
void SetField(UINT32 dwIndex, INT16 sData) throw DBSQLCA
void SetField(UINT32 dwIndex, INT32 lData) throw DBSQLCA
void SetField(UINT32 dwIndex, UINT16 swData) throw DBSQLCA
void SetField(UINT32 dwIndex, UINT32 dwData) throw DBSQLCA
void SetField(UINT32 dwIndex, SINGLE sfData) throw DBSQLCA
void SetField(UINT32 dwIndex, DOUBLE dfData) throw DBSQLCA
void SetField(UINT32 dwIndex, LPCTSTR lpctData)
                                             throw DBSQLCA
void SetField(UINT32 dwIndex, const DBR_DATETIME& dtData)
                                             throw DBSQLCA
void SetField(UINT32 dwIndex, const DBR_BINARY& blobData)
                                             throw DBSQLCA
void SetField(UINT32 ui32Index, const DBRArrayDataConstPtr&
                                  cparData) throw DBSQLCA
```

フィールド名で指定する場合

```
void SetField(LPCTSTR lpctFieldName, INT16 sData)
                                             throw DBSQLCA
void SetField(LPCTSTR lpctFieldName, INT32 lData)
                                             throw DBSQLCA
void SetField(LPCTSTR lpctFieldName, UINT16 swData)
                                             throw DBSQLCA
void SetField(LPCTSTR lpctFieldName, UINT32 dwData)
                                             throw DBSQLCA
void SetField(LPCTSTR lpctFieldName, SINGLE sfData)
                                             throw DBSQLCA
void SetField(LPCTSTR lpctFieldName, DOUBLE dfData)
                                             throw DBSQLCA
void SetField(LPCTSTR lpctFieldName, LPCTSTR lpctData)
                                             throw DBSQLCA
void SetField(LPCTSTR lpctFieldName, const DBR_DATETIME&
                                    dtData) throw DBSQLCA
void SetField(LPCTSTR lpctFieldName, const DBR_BINARY&
                                  blobData) throw DBSQLCA
void SetField(LPCTSTR lpctFieldName, const DBRArrayDataConstPtr&
                                  cparData) throw DBSQLCA
```

引数

dwIndex，ui32Index

1 から始まるフィールドの番号を指定します。指定するフィールドの番号は，SQL 文中での出現順に割り当てられます。

lpctFieldName

フィールド名を指定します。

xxData（データ型：形式に示す第 2 引数を参照してください）

設定する値を指定します。

戻り値

なし

機能詳細

カレントレコードのフィールドに値を設定します。

更新可能な DBResultSet オブジェクトに対して指定できます。Edit メソッドを呼び出した後に実行してください。

値はデータベースのデータ型に合わせて変換されます。データ型の変換規則については，「7.1 クラスライブラリで扱うデータ型と変換規則」の SetField メソッドでのデータ型変換規則を参照してください。

BLOB 型データをファイル経由で設定する場合

SetResultSetType メソッドで TYPE_BLOB_FILE を指定した場合，BLOB 型のフィールドに対する SetField メソッドの引数にはファイル名称を指定します。この場合，SetField メソッドの引数の型には，文字列と同じ LPCTSTR を使います。

DBR_BINARY 型を指定する場合

DBR_BINARY 型の各メンバには次の値を設定します。

Length：設定するデータの長さ（単位はバイト）

RealLength：設定不要

Data：実際のデータの領域を指すポインタ

繰り返し列を扱う場合

引数 cparData に DBArrayData オブジェクトを指定します。DBRArrayData オブジェクトは DBRArrayDataFactory クラスを利用して事前に生成しておく必要があります。なお，これ以外の引数には，繰り返し列は指定できません。

また，引数 cparData に指定されたオブジェクトが，DBRArrayData クラスのインスタンスを持っていない場合は，エラーになります。

SetField メソッドで値を設定した後，DBRArrayData オブジェクトの SetData メソッドを実行すると，SetField メソッド実行時点のデータは失われて，SetData メソッドで設定したデータに置き換わるため注意が必要です。

XDM/RD で更新できる型の制限

LONG VARCHAR 型，LONG NVARCHAR 型，LONG MVARCHAR 型のフィールドについては更新できません。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_NOT_FOUND
　引数 dwIndex の範囲，又は引数 lpctFieldName が不正です。

DB_ERROR_CANNOT_BE_NULL
　引数 lpctFieldName に NULL を指定しています。

DB_ERROR_NOT_IN_EDIT
　Edit メソッドが実行されていません。

DB_ERROR_DATA_TRUNCATED
　取得した値を指定した型に変換できません。

DB_ERROR_NO_INSTANCE
　引数 cparData に指定されたオブジェクトが，DBRArrayData クラスのインスタンスを持っていません。

DB_ERROR_CONVERT_ARRAY_TO_SCALAR
　引数 cparData に繰り返し列以外のフィールド値を設定しようとしました，又は引数 cparData 以外の引数に繰り返し列の値を設定しようとしました。

DB_ERROR_CANNOT_ACCESS_WHILE_EXECUTED
　Update メソッドを実行中に，DBRArrayData オブジェクトの SetData メソッドを呼び出しました。

SetNull メソッド

機能

指定されたフィールドに NULL を設定します。

形式

インデクス番号で指定する場合

```
void SetNull(UINT32 dwIndex) throw DBSQLCA
```

フィールド名で指定する場合

```
void SetNull(LPCTSTR lpctFieldName) throw DBSQLCA
```

引数

dwIndex

1 から始まるフィールドの番号を指定します。

lpctFieldName

フィールド名を指定します。

戻り値

なし

機能詳細

指定されたフィールドに NULL を設定します。

NULL は C 言語で使用する NULL ポインタの意味ではなく，データベースシステムでは「値がない」ことを意味します。

SetField メソッドと同様に Edit メソッドを呼び出した後に実行してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_FOUND
　引数 dwIndex の範囲が不正です。

DB_ERROR_CANNOT_BE_NULL
　引数 lpctFieldName に NULL を指定しています。

DB_ERROR_NOT_IN_EDIT
　Edit メソッドが実行されていません。

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

DB_DRV_ERROR_INVALID_ARGUMENT
　引数に指定したフィールド名が不正です。

Top メソッド

機能

カーソルを ResultSet の先頭のレコードへ移動します。

形式

```
void Top(void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

カーソルを ResultSet の先頭のレコードへ移動します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_RESULTSET_NOT_EXIST
　非同期実行時に，検索結果がまだ取得できていません。

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

Update メソッド

機能

フィールドの更新結果をデータベースへ通知します。

形式

```
void Update(void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

検索したレコードを更新する場合，Edit メソッドの後，SetField メソッドでフィールドの値を更新し，Update メソッドを呼び出すと，更新結果がデータベースへ通知されます。

トランザクションを設定したアプリケーションの場合，通知されたデータはコミットされるまでデータベースに反映されません。

SetField メソッドで更新したあと，Update メソッドを呼び出さないで Next，又は PageNext メソッドを呼び出した場合，更新したフィールドの内容は無効になります。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_NOT_IN_EDIT
　Edit メソッドが実行されていません。

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行中です。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。
DB_ERROR_DAB_ACCESS_ERROR
　DABroker でエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。
DB_ERROR_DAB_SYSTEMCALL_ERROR
　システムコールでエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。
DB_ERROR_TIMEOUT
　タイムアウトが発生したため制御を戻します。

WaitForDataSource メソッド

機能

DBResultSet オブジェクトで要求した実行待ち，及び実行中の非同期処理が終了するまで待ちます。非同期処理中に同期を取りたい場合に利用できます。

形式

```
BOOLEAN WaitForDataSource(UINT32 swWaitTime = DBR_INFINITE)
```

引数

swWaitTime

非同期処理の終了を待つための最大時間（単位：ミリ秒），又は DBR_INFINITE を指定します。

指定した時間内に非同期処理が終了した場合，TRUE が返ります。指定した時間を経過しても非同期処理が終了しない場合，FALSE が返ります。非同期実行中の SQL がない場合は，TRUE が返ります。

DBR_INFINITE を指定した場合はタイムアウト時間を設定しません。非同期処理が終了するまで待ち続けます。

同期実行時はすぐに TRUE を返します。

戻り値

データ型：BOOLEAN

TRUE：すべての非同期処理が終了しました。

FALSE：タイムアウト時間が経過しました。

機能詳細

DBResultSet オブジェクトで要求した実行待ち，及び実行中の非同期処理(SQL)が，終了するのを待ちます。

DBConnection クラスの WaitForDataSource メソッドとの違い

DBResultSet オブジェクトの WaitForDataSource メソッドでは，WaitForDataSource メソッドを実行した DBResultSet オブジェクトの非同期実行処理が終了するまで待ちます。ほかのオブジェクトの非同期実行処理については待ちません。そのため，ほかのオブジェクトの非同期実行処理が実行中でも，WaitForDataSource メソッドを呼び出したオブジェクトの非同期実行処理が終了すれば，WaitForDataSource メソッドは TRUE を返します。

同じ DBConnection オブジェクトで実行しているすべての非同期実行処理が終了するのを待ちたい場合は，DBConnection クラスの WaitForDataSource メソッドを実行してください。

発生する例外

なし

5.8　DBResultSetMetaData クラスの詳細

ResultSet の情報を管理します。

このクラスは，フィールドの数，名称，属性などを取得するためのメソッドを提供するクラスです。

機 能	メソッド名
繰り返し列の要素の数を取得します。	GetArraySize
ResultSet のフィールド数を取得します。	GetColumnCount
フィールドのデータ型を C++の型で取得します。	GetColumnCType
フィールドのデータ型を DBMS の型で取得します。	GetColumnDBType
フィールド名称を取得します。	GetColumnName
フィールドの精度（桁数）を取得します。	GetColumnPrecision
フィールドの小数点以下の桁数を取得します。	GetColumnScale
フィールドのデータ型を取得します。	GetColumnType
対象となるオブジェクトを生成した DBStatement オブジェクト, DBPreparedStatement オブジェクト，又は DBResultSet オブジェクトのポインタを取得します。	Parent

GetArraySize メソッド

機能

繰り返し列の要素の数を取得します。

形式

インデクス番号で指定する場合

```
UINT32 GetArraySize(UINT32 ui32Index) throw DBSQLCA
```

フィールド名で指定する場合

```
UINT32 GetArraySize(LPCTSTR lpctFieldName) throw DBSQLCA
```

引数

ui32Index

フィールド番号を，１～フィールド数の範囲の値で指定します。

lpctFieldName

フィールド名を指定します。

戻り値

データ型：UINT32

繰り返し列の要素数を返します。

機能詳細

フィールド番号又はフィールド名によって特定した繰り返し列の，要素の数を取得します。

指定したフィールドが繰り返し列でない場合は，0 が返ります。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_CANNOT_BE_NULL
　引数 lpctFieldName に NULL を指定しています。

DB_ERROR_NOT_FOUND
　インデックス指定の場合は，指定したフィールド番号が 1 未満の値，又はフィールド数より大きい値を指定しています。フィールド名指定の場合は，指定したフィールド名が間違っています(NULL を除く)。

GetColumnCount メソッド

機能

ResultSet のフィールド数を取得します。

形式

```
UINT32 GetColumnCount(void)
```

引数

なし

戻り値

データ型：UINT32

ResultSet 中のフィールド数。

機能詳細

ResultSet のフィールド数を取得します。

必ず 1 以上の数値が返ります。

発生する例外

なし

GetColumnCType メソッド

機能

指定したフィールドのデータ型を C++の型で取得します。

形式

インデクス番号で指定する場合

```
DBR_CTYPE GetColumnCType(UINT32 dwIndex) throw DBSQLCA
```

フィールド名で指定する場合

```
DBR_CTYPE GetColumnCType(LPCTSTR lpctFieldName) throw DBSQLCA
```

引数

dwIndex

1 から始まるフィールドの番号を指定します。

lpctFieldName

フィールドの名称を指定します。

戻り値

データ型：DBR_CTYPE

指定したフィールドのデータ型を C++のデータ型で取得します。

COL_CTYPE_ INT16：INT16

COL_CTYPE_ INT32：INT32

COL_CTYPE_ UINT16：UINT16

COL_CTYPE_UINT32：UINT32

COL_CTYPE_DOUBLE：DOUBLE

COL_CTYPE_SINGLE：SINGLE

COL_CTYPE_CHAR：TCHAR，LPTSTR，LPCTSTR

COL_CTYPE_OFFSET：OFFSET

COL_CTYPE_DATETIME：DBR_DATETIME

COL_CTYPE_BINARY：DBR_BINARY

COL_CTYPE_BOOLEAN：BOOLEAN

機能詳細

指定したフィールドのデータ型を C++のデータ型で取得します。

それぞれのデータ型は，DBResultSet オブジェクト中に読み込まれたときに，デフォルトの C++データ型に変換されます。

変換されたデフォルトのデータ型がどのような型になるかは，使用する DBMS によって異なります。型変換の詳細については，「7.1 クラスライブラリで扱うデータ型と変換規則」を参照してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_FOUND
　指定したフィールドがありません。

DB_ERROR_CANNOT_BE_NULL
　引数 lpctFieldName に NULL を指定できません。

GetColumnDBType メソッド

機能

指定したフィールドのデータ型を DBMS の型で取得します。

形式

インデクス番号で指定する場合

```
UINT16 GetColumnDBType(UINT32 dwIndex) throw DBSQLCA
```

フィールド名で指定する場合

```
UINT16 GetColumnDBType(LPCTSTR lpctFieldName) throw DBSQLCA
```

引数

dwIndex

1 から始まるフィールドの番号を指定します。

lpctFieldName

フィールドの名称を指定します。

戻り値

データ型：UINT16

指定したフィールドのデータ型を DBMS のデータ型で取得します。

機能詳細

指定したフィールドの属性を DBMS のデータ型で取得します。

それぞれのデータ型は，DBMS によって異なります。データ型の詳細については，使用する DBMS の仕様をご確認ください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_FOUND
　指定したフィールドがありません。

DB_ERROR_CANNOT_BE_NULL
　引数 lpctFieldName に NULL を指定できません。

GetColumnName メソッド

機能

フィールド名称を取得します。

形式

```
LPCTSTR GetColumnName(UINT32 dwIndex) throw DBSQLCA
```

引数

dwIndex

１から始まるフィールドの番号を指定します。指定するフィールド番号は，SQL 文中での出現順に割り当てられます。

戻り値

データ型：LPCTSTR

指定されたフィールドの名称を取得します。

機能詳細

指定されたフィールドのフィールド名を取得します。

SELECT 文でフィールド演算や，COUNT などの関数を使用した場合，フィールド名はデータベースによって決められます。このような場合にこのメソッドを使用します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_FOUND
　指定したフィールドがありません。

GetColumnPrecision メソッド

機能

フィールドの精度（桁数）を取得します。

形式

インデクス番号で指定する場合

```
UINT32 GetColumnPrecision(UINT32 dwIndex) throw DBSQLCA
```

フィールド名で指定する場合

```
UINT32 GetColumnPrecision(LPCTSTR lpctFieldName) throw DBSQLCA
```

引数

dwIndex

1から始まるフィールドの番号を指定します。

lpctFieldName

フィールドの名称を指定します。

戻り値

データ型：UINT32

フィールドの精度（桁数）を取得します。COL_TYPE_NUMERIC の場合，小数点以下を含めた数値の有効桁数を取得します。

機能詳細

指定されたフィールドのデータの精度（桁数）を取得します。

指定されたフィールドのデータ型によって,戻り値の持つ意味が異なるものがあります。各データ型で返す値（Precision,Scale）を，表 5-2 から 5-9 に示します。

表 5-2　Precision,Scale で返す値（ORACLE）

データ型	Precision	Scale
NUMBER(固定小数点数)	定義長	小数点以下の桁数
NUMBER（浮動小数点数）	45	0
VARCHAR2	定義長	0
RAW	定義長	0
LONG	0	0
LONG RAW	0	0
DATE	32※	0
ROWID	18	0

注※ この値は，sizeof(DBR_DATETIME)で取得した値です。

表 5-3　Precision,Scale で返す値（SQL Anywhere）

データ型	Precision	Scale
CHAR CHARACTER VARCHAR CHARACTER VARYING	定義長	0
SYSNAME	30	0
LONG VARCHAR TEXT	0	0

データ型	Precision	Scale
SMALLINT BIT TINYINT	6	0
INT INTEGER	11	0
DECIMAL NUMERIC	定義長	定義長
MONEY	19	4
SMALLMONEY	10	4
REAL	9	0
DOUBLE	17	0
DATE	32※	0
TIME	32※	0
TIMESTAMP DATETIME SMALLDATETIME	32※	0
BINARY	定義長	0
LONG BINARY IMAGE	0	0

注※ この値は，sizeof(DBR_DATETIME)で取得した値です。

表 5-4　Precision,Scale で返す値(Adaptive Server Anywhere)

データ型	Precision	Scale
CHAR CHARACTER VARCHAR CHARACTER VARYING	定義長	0
SYSNAME	30	0
LONG VARCHAR TEXT	0	0
SMALLINT BIT TINYINT OLDBIT	6	0
INT INTEGER	11	0

データ型	Precision	Scale
DECIMAL NUMERIC	定義長	定義長
MONEY	19	4
SMALLMONEY	10	4
REAL	9	0
DOUBLE	17	0
DATE	32※	0
TIME	32※	0
TIMESTAMP DATETIME SMALLDATETIME	32※	0
BINARY VARBINARY	定義長	0
LONG BINARY IMAGE java serialization java.lang.Object	0	0
BIGINT	20	0

注※ この値は，sizeof(DBR_DATETIME)で取得した値です

表 5-5　Precision,Scale で返す値(SQL Server)

データ型	Precision	Scale
CHAR VARCHAR	定義長	0
TEXT	0	0
SMALLINT TINYINT BIT	6	0
INT	11	0
DECIMAL NUMERIC	定義長	定義長
MONEY	19	4
SMALLMONEY	10	4
REAL	9	0
DOUBLE	17	0

データ型	Precision	Scale
DATETIME SMALLDATETIME	32※	0
TIMESTAMP	8	0
BINARY VARBINARY	定義長	0
IMAGE	0	0

注※ この値は，sizeof(DBR_DATETIME)で取得した値です

表 5-6　Precision,Scale で返す値（HiRDB）

データ型	Precision	Scale
SMALLINT	6	0
INT	11	0
SMALLFLT	9	0
FLOAT	17	0
DECIMAL	定義長	小数点以下の桁数
CHAR	定義長	0
VARCHAR	定義長	0
MCHAR	定義長	0
MVARCHAR	定義長	0
NCHAR	定義長×2	0
NVARCHAR	定義長×2	0
DATE	32※	0
TIME	32※	0
INTERVAL YEAR TO DAY	32※	0
INTERVAL HOUR TO SECOND	32※	0
BLOB	定義長	0
BINARY	定義長	0

注※ この値は，sizeof(DBR_DATETIME)で取得した値です。

表 5-7　Precision,Scale で返す値(XDM/RD)

データ型	Precision	Scale
DECIMAL	定義長	小数点以下の桁数
LARGE DECIMAL	定義長	小数点以下の桁数

データ型	Precision	Scale
INT	11	0
SMALLINT	6	0
FLOAT	17	0
SMALLFLT	9	0
CHAR	定義長	0
VARCHAR	定義長	0
LONG VARCHAR	定義長	0
MCHAR	定義長	0
MVARCHAR	定義長	0
LONG MVARCHAR	定義長	0
NCHAR	定義長×2	0
NVARCHAR	定義長×2	0
LONG NVARCHAR	定義長×2	0
DATE	32※	0

注※ この値は，sizeof(DBR_DATETIME)で取得した値です。

表 5-8　Precision,Scale で返す値（SQL/K）

データ型	Precision	Scale
SMALLINT	2	0
INTEGER	4	0
DECIMAL	定義長	定義長
NUMERIC TRAILING	定義長	定義長
NUMERIC UNSIGNED	定義長	定義長
CHAR	定義長	0
NCHAR	定義長×2	0
MCHAR	定義長	0
XCHAR	定義長	0

表 5-9　Precision,Scale で返す値（XDM/SD）

データ型	Precision	Scale
BIT	定義長	0
$DBK(データベースキー)	定義長	0

データ型	Precision	Scale
NUMERIC TRAILING	定義長	定義長
DECIMAL	定義長	定義長
INTEGER	4	0
SMALLFLT	2	0
NCHAR	定義長×2	0
CHAR	定義長	0

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_FOUND
　指定したフィールドがありません。

DB_ERROR_CANNOT_BE_NULL
　引数 lpctFieldName に NULL を指定しています。

GetColumnScale メソッド

機能

フィールドの小数点以下の桁数を取得します。

形式

インデクス番号で指定する場合

```
INT32 GetColumnScale(UINT32 dwIndex) throw DBSQLCA
```

フィールド名で指定する場合

```
INT32 GetColumnScale(LPCTSTR lpctFieldName) throw DBSQLCA
```

引数

dwIndex

1 から始まるフィールドの番号を指定します。

lpctFieldName

フィールドの名称を指定します。

戻り値

データ型：INT32

COL_TYPE_NUMERIC の場合，小数点の以下の桁数を取得します。

0 の場合は整数となります。また，マイナスの場合は桁上がりとなります。

機能詳細

指定されたフィールドが数値データの場合，データの小数点以下の桁数を取得します。そのほかのデータの場合，返される値は意味を持ちません。

負の値が戻された場合，戻された値分の桁が上がります。

また，GetColumnPrecision メソッドで取得した値よりも大きな正の値が戻された場合，戻された値分の桁が下がります。

指定されたフィールドのデータ型によって，戻り値の持つ意味が異なるものがあります。各データ型で返す値（Scale）については，GetColumnPrecision メソッドの表 5-2 から 5-7 を参照してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_FOUND
　指定したフィールドがありません。

DB_ERROR_CANNOT_BE_NULL
　引数 lpctFieldName に NULL を指定しています。

GetColumnType メソッド

機能

フィールドのデータ型を取得します。

形式

インデクス番号で指定する場合

```
UINT16 GetColumnType(UINT32 dwIndex) throw DBSQLCA
```

フィールド名で指定する場合

```
UINT16 GetColumnType(LPCTSTR lpctFieldName) throw DBSQLCA
```

引数

dwIndex

1 から始まるフィールドの番号を指定します。

lpctFieldName

フィールドの名称を指定します。

戻り値

データ型：UINT16

指定したフィールドのデータ型を取得します。

COL_TYPE_INT16：2 バイト長符号付き 2 進整数

COL_TYPE_INT32：4 バイト長符号付き 2 進整数

COL_TYPE_UINT16：2 バイト長符号なし 2 進整数

COL_TYPE_UINT32：4 バイト長符号なし 2 進整数

COL_TYPE_NUMERIC：符号付き 10 進数

COL_TYPE_SINGLE：4 バイト長浮動小数点数

COL_TYPE_DOUBLE：8 バイト長浮動小数点数

COL_TYPE_CHAR：固定長文字列

COL_TYPE_VARCHAR：可変長文字列（最大長指定あり）

COL_TYPE_LONGVARCHAR：可変長文字列（最大長指定なし）

COL_TYPE_DATE：日付型

COL_TYPE_TIME：時間型

COL_TYPE_TIMESTAMP：日付・時間型

COL_TYPE_INTERVAL：時間間隔型

COL_TYPE_BOOLEAN：ブーリアン型

COL_TYPE_BINARY：固定長バイナリ型

COL_TYPE_VARBINARY：可変長バイナリ型（最大長指定あり）

COL_TYPE_LONGVARBINARY：可変長バイナリ型（最大長指定なし）

COL_TYPE_MONEY：金額型

COL_TYPE_SERIAL：自動インクリメント型

COL_TYPE_ROWID：ROWID

COL_TYPE_BIGINT：8 バイト長符号付き整数

COL_TYPE_BIT：1 バイト長符号なし 2 進整数

機能詳細

指定されたフィールドのデータベース中でのデータ型を取得します。

それぞれのデータ型は，DBResultSet オブジェクト中に読み込まれたときに，デフォルトの C++データ型に変換されます。

変換されたデフォルトのデータ型がどのような型になるかは，使用する DBMS によって異なります。

型変換の詳細については，「7.1 クラスライブラリで扱うデータ型と変換規則」を参照してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_FOUND
　指定したフィールドがありません。

DB_ERROR_CANNOT_BE_NULL
　引数 lpctFieldName に NULL を指定しています。

Parent メソッド

機能

対象となるオブジェクトを生成した DBStatement オブジェクト，DBPreparedStatement オブジェクト，又は DBResultSet オブジェクトのポインタを取得します。

形式

```
LPOBJECT Parent(void)
```

引数

なし

戻り値

データ型：LPOBJECT

対象となるオブジェクトを生成した DBStatement オブジェクト，DBPreparedStatement オブジェクト，又は DBResultSet オブジェクトのポインタ。

機能詳細

対象となるオブジェクトを生成した DBStatement オブジェクト，DBPreparedStatement オブジェクト，又は DBResultSet オブジェクトのポインタを取得します。どのオブジェクトのポインタを取得するかは，DBResultSetMetaData オブジェクトを取得したメソッドによって異なります。

- DBResultSetMetaData オブジェクトを GetResultSetMetaData メソッドで取得した場合

GetResultSetMetaData メソッド 呼び出し元クラス	Parent メソッドの戻り値
DBStatement	呼び出し元(DBStatement)オブジェクトのポインタ
DBPreparedStatement	呼び出し元(DBPreparedStatement)オブジェクトのポインタ

- DBResultSet::GetMetaData メソッドで取得した場合

DBResultSet オブジェクトを 作成したクラス	Parent メソッドの戻り値
DBStatement	DBResultSet オブジェクトを作成した DBStatement オブジェクトのポインタ
DBPreparedStatement	DBResultSet オブジェクトを作成した DBPreparedStatement オブジェクトのポインタ

DBResultSet オブジェクトを作成したクラス	Parent メソッドの戻り値
DBCallableStatement	GetMetaData メソッドを呼び出した DBResultSet オブジェクトのポインタ

発生する例外

なし

5.9　DBPreparedStatement クラスの詳細

実行時に，動的に設定できる?パラメタを利用する SQL 文を管理します。このクラスは，?パラメタを含む SQL 文の実行やその結果の取得などのメソッドを提供するクラスです。

DBPreparedStatement クラスを使えば，?の値を除いて実行する SQL 文が固定されるため，DBMS の SQL 文シンタックスチェックが最初以外必要なくなります。例えば，同じテーブルに対してデータを追加する場合，INSERT 文の形式は固定され，データだけが変化します。このような場合，?パラメタは効果的に利用できます。

上記の場合，Execute メソッドは最初だけ呼び出せばよく，その後の更新処理では,SetParam, ExecuteUpdate，SetParam，ExecuteUpdate…と繰り返し実行することで異なるデータを追加できます。指定方法の詳細については，「2.6 詳細版クラスのデータベースアクセス」を参照してください。

ORACLE を使用している場合,？パラメタはプレースホルダに相当します。？パラメタとプレースホルダのどちらでも指定できますが,一つの SQL 文中に混在はできません。

SQL/K の場合，？パラメタを含む SQL の実行時は，必ず？パラメタに値を設定してください。SetNull, 及びデフォルト値(NULL)を使用した場合，ExecuteUpdate，及び GetResultSet でエラーになります。NULL 値を指定したい場合は，？パラメタの代わりに NULL 述語を指定してください。

機 能	メソッド名
データベースに SQL 実行の情報を通知します。	Execute
INSERT, DELETE, UPDATE の SQL 文を実行します。	ExecuteUpdate
SQL 文の非同期実行時のエラー情報を得るために，DBSQLCA オブジェクトへのポインタを取得します。	GetErrorStatus
検索結果のフィールドの数を取得します。	GetFieldCount
SetMaxFieldSize メソッドで設定したデータ長を取得します。	GetMaxFieldSize
ResultSet に検索できるレコード数の最大値を取得します。	GetMaxRows
DBPreparedStatement オブジェクトの名前を取得します。	GetName
SetParam メソッドで指定したパラメタの値を取得します。	GetParam
パラメタの数を取得します。	GetParamCount
SQL 文を実行し，検索したレコードから，ResultSet オブジェクトを生成します。	GetResultSet
DBResultSetMetaData オブジェクトを生成します。	GetResultSetMetaData
SQL 文を使って更新，追加，又は削除したレコード数を取得します。	GetUpdateRows
DBPreparedStatement オブジェクトに非同期実行中（又は実行待ち）のステートメントがあるかどうかを確認します。	InExecute
パラメタの値が NULL（欠損値）かどうか確認します。	IsNull
対象となるオブジェクトを生成した DBConnection オブジェクトのポインタを取得します。	Parent
DBPreparedStatement オブジェクトを削除します。	Remove

機 能	メソッド名
DBResultSet オブジェクトを削除します。	RemoveResultSet
SQL の INSERT 文使用時,1 度に何レコード挿入するかを指定します。	SetInsertRows
アプリケーションで受け取るフィールドの長さを指定します。	SetMaxFieldSize
ResultSet に検索するレコード数の最大値を指定します。	SetMaxRows
指定されたパラメタに NULL（欠損値)を設定します。	SetNull
?パラメタに値を設定します。	SetParam
?パラメタの属性を指定します。	SetParamType
DBResultSet オブジェクトの生成オプションを指定します。	SetResultSetType
DBPreparedStatement オブジェクトで要求した実行待ち，及び実行中の非同期処理が終了するまで待ちます。	WaitForDataSource

Execute メソッド

機能

SQL 文実行の前処理として，レコードがロックされていたときの動作を指定し，データベースに SQL 実行の情報を通知します。

検索結果を取得する場合，Execute メソッドを呼び出した後，GetResultSet を呼び出して DBResultSet オブジェクトを取得します。

形式

```
void Execute (UINT16 swWait = LOCK_OPT_DEFAULT) throw DBSQLCA
```

引数

swWait

実行する SQL 単位に，検索対象のレコードがほかのトランザクションによってロックされていた場合の動作を指定します。

SQL/K の場合は，どの値を指定しても，ロックが解除されるまで待ち状態になります。

XDM/SD の場合は，どの値を指定しても，ロックが解除されるまで待たないで，すぐにエラーを返します。

次の値のうち，どれか一つを指定します。

- LOCK_OPT_DEFAULT
  DBConnection オブジェクトの設定を引き継ぎます。(DBDriver クラスの Connect メソッドの引数 swWait に指定した値)
- LOCK_OPT_NOWAIT，又は LOCK_OPT_WITH_ROLLBACK
  ロックの解除を待たないで，すぐにエラーを返します。HiRDB 又は XDM/RD の場合は同時にロールバックを実行します。それ以外の DBMS の場合は，ロールバックは実行しません。
- LOCK_OPT_WITHOUT_ROLLBACK

ロックの解除を待たないで，すぐにエラーを返します。ロールバックは実行しません。

ただし，現在のバージョンでは，XDM/RD ではロールバックを実行します。また，HiRDB の DELETE,INSERT,UPDATE 文では，このオプションは無効で，LOCK_OPY_WAIT と同じ動作になります。

- LOCK_OPT_WAIT
  ロックが解除されるまで待ちます。

戻り値

なし

機能詳細

DBPreparedStatement オブジェクトの生成時に設定されている SQL 文実行の情報をデータベースに通知します（DBPreparedStatement オブジェクトは，DBConnection クラスの CreatePreparedStatement メソッドで生成します。）

このステートメントの ResultSet が既にある場合，Execute 実行時に ResultSet は削除されます。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
メモリ容量が不足しています。

DB_ERROR_IN_ASYNC_EXECUTE
ステートメントが非同期実行処理中です。

DB_ERROR_DRIVER_ERROR
DBMS でエラーが発生しました。

DB_ERROR_DAB_ACCESS_ERROR
DABroker でエラーが発生しました。
DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR
システムコールでエラーが発生しました。
DBSQLCA クラスで詳細コードを確認してください。

DB_DRV_ERROR_PHOLDER_AND_QP
プレースホルダ，及び？パラメタが混在する SQL 文は指定できません。

DB_ERROR_TIMEOUT
タイムアウトが発生したため制御を戻します。

DB_DRV_ERROR_INVALID_SQL_EXCLUSIVE
WITH 句で始まる SQL 文の指定時には，DBResultSet オブジェクトの生成オプションを TYPE_EXCLUSIVE 以外に設定してください。

DB_ERROR_DAB_ILLEGAL_VALUE
指定した引数が不正です。
DBSQLCA クラスで詳細コードを確認してください。

ExecuteUpdate メソッド

機能

INSERT，DELETE，UPDATE の SQL 文を実行します。

形式

```
void ExecuteUpdate(void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

追加・更新・削除のための SQL 文を実行します。

検索のために SELECT 文を実行する場合は，このメソッドではなく，GetResultSet メソッドを呼び出します。

なお，GetUpdateRows メソッドを呼び出して，処理したレコード数を取得できます。

このメソッドは，非同期実行可能なメソッドです。

XDM/RD を使用している場合

ExecuteUpdate を呼び出す前に，すべてのパラメタについて SetParamType メソッドを呼び出し，データ型を設定しておく必要があります。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_BEFORE_EXECUTE
　Execute が実行されていません。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_ERROR_DAB_ACCESS_ERROR
　DABroker でエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR
　システムコールでエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_DRV_ERROR_RDA_PARAM_TYPE
　パラメタの型が未設定か，又は設定された型が不正です。
DB_ERROR_TIMEOUT
　タイムアウトが発生したため制御を戻します。

GetErrorStatus メソッド

機能

SQL 文の非同期実行時のエラー情報を得るために，DBSQLCA オブジェクトへのポインタを取得します。

形式

```
DBSQLCA *GetErrorStatus(void)
```

引数

なし

戻り値

データ型：DBSQLCA*

DBSQLCA オブジェクトへのポインタ。

機能詳細

DBSQLCA オブジェクトへのポインタを取得します。

非同期実行時にエラーが発生した場合，DBSQLCA オブジェクトにエラー情報が設定されます。

非同期処理のときは，エラーが発生しても直にエラーを取得できません。このため，ユーザは任意の時点でエラーを確認する必要があります。

DBSQLCA オブジェクトでは，非同期実行中に発生したエラー情報を最大 100 回分保存できます。エラー情報は，新しいエラー情報から 100 回分保存されるため，エラー情報を確認したあとは DBSQLCA クラスの Delete メソッドを呼び出して，不要なエラー情報をクリアしておいてください。

発生する例外

なし

GetFieldCount メソッド

機能

検索結果のフィールドの数を取得します。

形式

```
UINT32 GetFieldCount(void)
```

引数

なし

戻り値

データ型：UINT32

フィールドの個数。

機能詳細

DBPreparedStatement オブジェクトの生成時に指定した SELECT 文の，検索結果中に含まれるフィールドの個数を取得します。

GetFieldCount メソッドを呼び出す前に，Execute メソッドを呼び出しておく必要があります。

発生する例外

なし

GetMaxFieldSize メソッド

機能

SetMaxFieldSize メソッドで設定したデータ長を取得します。

形式

フィールド名で指定する場合

```
UINT32   GetMaxFieldSize(LPCTSTR lpctFieldName = NULL) throw DBSQLCA
```

インデクス番号で指定する場合

```
UINT32   GetMaxFieldSize(UINT32 dwIndex) throw DBSQLCA
```

引数

lpctFieldName

フィールドの名称，又は NULL を指定します。

NULL が指定された場合，すべてのフィールド長の合計値をバイト数で取得します。

dwIndex

1 から始まるフィールドの番号を指定します。

戻り値

データ型：UINT32

SetMaxFieldSize メソッドで設定したデータ長。

機能詳細

SetMaxFieldSize メソッドで設定したデータの長さを指定します。データベース中のフィールドの定義長ではないことに注意してください。

取得するデータがフィールドの値を格納する領域の最大長よりも大きい場合，データは切り捨てられます。

このメソッドを呼び出す前に，Execute メソッドを呼び出しておく必要があります。

不定長のフィールドの場合は，0 を返します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_BEFORE_EXECUTE
　Execute メソッドが実行されていません。

DB_ERROR_NOT_FOUND
　指定されたフィールドがありません。

GetMaxRows メソッド

機能

ResultSet に検索できるレコード数の最大値を取得します。

形式

```
UINT32 GetMaxRows(void)
```

引数

なし

戻り値

テータ型：UINT32

SetMaxRows メソッドで設定した，レコード数の最大値を取得します。

機能詳細

SetMaxRows メソッドで設定した，DBResultSet オブジェクトで取得できるレコード数の最大値を取得します。SetMaxRows メソッドを一度も呼び出していない場合はデフォルト値の 100 を返します。

発生する例外

なし

GetName メソッド

機能

DBPreparedStatement オブジェクトの名前を取得します。

形式

```
LPCTSTR GetName(void)
```

引数

なし

戻り値

データ型：LPCTSTR

DBPreparedStatement オブジェクトの名前が返ります。

機能詳細

DBPreparedStatement オブジェクトの名前を取得します。DBPreparedStatement オブジェクトの名称は，DBConnection クラスの CreatePreparedStatement メソッドの引数から生成されます。

発生する例外

なし

GetParam メソッド

機能

SetParam メソッドで指定したパラメタの値を取得します。

形式１：ユーザ確保領域へコピーする場合

インデクス番号指定

```
void GetParam(UINT32 dwIndex, INT16* sParam) throw DBSQLCA
void GetParam(UINT32 dwIndex, INT32* lParam) throw DBSQLCA
void GetParam(UINT32 dwIndex, UINT16* swParam) throw DBSQLCA
void GetParam(UINT32 dwIndex, UINT32 * dwParam) throw DBSQLCA
void GetParam(UINT32 dwIndex, SINGLE* sfParam) throw DBSQLCA
void GetParam(UINT32 dwIndex, DOUBLE* dfParam) throw DBSQLCA
void GetParam(UINT32 dwIndex, DBR_DATETIME* dtParam)
                                                  throw DBSQLCA
void GetParam(UINT32 ui32Index, DBRArrayDataPtr* parParam)
                                                  throw DBSQLCA
```

レコード番号・インデクス番号指定

```
void GetParam(UINT32 dwRow, UINT32 dwIndex, INT16* sParam)
                                                     throw DBSQLCA
void GetParam(UINT32 dwRow, UINT32 dwIndex, INT32* lParam)
                                                     throw DBSQLCA
void GetParam(UINT32 dwRow, UINT32 dwIndex, UINT16* swParam)
                                                     throw DBSQLCA
void GetParam(UINT32 dwRow, UINT32 dwIndex, UINT32 * dwParam)
                                                     throw DBSQLCA
void GetParam(UINT32 dwRow, UINT32 dwIndex, SINGLE* sfParam)
                                                     throw DBSQLCA
void GetParam(UINT32 dwRow, UINT32 dwIndex, DOUBLE* dfParam)
                                                     throw DBSQLCA
void GetParam(UINT32 dwRow, UINT32 dwIndex, DBR_DATETIME*
                                        dtParam) throw DBSQLCA
void GetParam(UINT32 ui32Row, UINT32 ui32Index,
                    DBRArrayDataPtr* parParam) throw DBSQLCA
```

形式２：DABroker が用意した領域を参照する場合

インデクス番号指定

```
void GetParam(UINT32 dwIndex, LPTSTR* lptParam) throw DBSQLCA
void GetParam(UINT32 dwIndex, DBR_BINARY* blobParam)
                                                throw DBSQLCA
```

レコード番号・インデクス番号指定

```
void GetParam(UINT32 dwRow, UINT32 dwIndex, LPTSTR* lptParam)
                                                throw DBSQLCA
void GetParam(UINT32 dwRow, UINT32 dwIndex,
                          DBR_BINARY* blobParam) throw DBSQLCA
```

形式３：ユーザ確保領域へコピーする場合

インデクス番号指定

```
UINT32 GetParam(UINT32 dwIndex, LPTSTR lptParam,
                                 UINT32 dwSize) throw DBSQLCA
UINT32 GetParam(UINT32 dwIndex, DBR_BINARY* blobParam,
                                 UINT32 dwSize) throw DBSQLCA
```

レコード番号・インデクス番号指定

```
UINT32 GetParam(UINT32 dwRow, UINT32 dwIndex, LPTSTR lptParam,
                                   UINT32 dwSize) throw DBSQLCA
UINT32 GetParam(UINT32 dwRow, UINT32 dwIndex,
          DBR_BINARY* blobParam, UINT32 dwSize) throw DBSQLCA
```

引数

dwIndex ,ui32Index

1 から始まるパラメタの番号を指定します。指定するパラメタの番号は，SQL 文中での出現順に割り当てられます。

dwRow

1 から始まる，レコード番号を指定します。

*xxParam（形式 1，3 の第 2 引数）

データをコピーする領域のポインタを指定します。

*xxParam（形式 2 の第 2 引数）

ポインタ変数を指定します。

dwSize

コピー先の領域サイズをバイト数で指定します。

戻り値

データ型：UINT32

コピーした文字列の長さが返ります。パラメタの値を指定された型にキャストして取得します(形式３の場合)。

機能詳細

SetParam で設定した値を取得します。このメソッドは，Execute を呼び出し，SetParam メソッドでパラメタを渡した後に実行してください。SetParam を実行する前に GetParam を実行した場合, NULL 文字, 又は 0 が返ります。

形式 1 の処理

パラメタ値を，引数に指定されたポインタが指す領域にコピーします。

繰り返し列がある場合は，引数 parParam に設定します。

形式 2 の処理

DABroker が確保した領域にパラメタ値をコピーし，その先頭アドレスをポインタ変数に設定します。このため，LPTSTR または DBR_BINARY の変数の確保だけで，フィールド値をコピーする領域を確保する必要はありません。

- パラメタ値の有効期間

  パラメタ値は，Execute メソッドや SetProcedure メソッドを実行するまで，又は取得したパラメタに対して SetParam を実行するまで有効です。

形式 3 の処理

- LPTSTR 型

  引数 lptParam の指す領域は，引数 dwSize バイト分確保されている必要があります。

  引数 dwSize がデータベースから取得したデータの長さ以下の場合，引数 dwSize-1 分のデータ（最後には NULL 終端文字が入る）がコピーされ，残りのデータは切り捨てられます。

  引数 dwSize がデータベースから取得したデータの長さよりも大きい場合，文字列はすべてコピーします。

- DBR_BINARY 型

  パラメタ値をコピーする領域（LPTSTR 型）を確保し，DBR_BINARY 型変数（構造体）の Data メンバに設定し，この変数を GetField メソッドの引数に指定します。引数 dwSize はこの領域のサイズを指定します。

  引数 dwSize がデータベースから取得したデータの長さ以下の場合，引数 dwSize 分のデータがコピーされ，残りのデータは切り捨てられます。この場合，NULL 終端文字は入りません。引数 dwSize がデータベースから取得したデータの長さより大きい場合，データはすべてコピーされ，残りの領域には NULL 終端文字が埋められます。

データ変換について

データベースのデータ型と GetParam の引数に指定したデータ型とが異なる場合は，値の変換できるものについては引数で指定したデータ型に変換して返します。

文字列から数値データ型への変換に失敗した場合は 0 を返します。データ型の変換規則については，「7.1 クラスライブラリで扱うデータ型と変換規則」の GetField，GetParam メソッドでのデータ型変換規則を参照してください。

NULL 値の扱い

NULL であるパラメタに対して GetParam メソッドを呼び出すと，戻される値は意味のない値(0，空文字列，要素がすべて 0，空文字列からなる構造体，又は不定値)となります。

このため，パラメタの値が NULL である可能性がある場合は，IsNull メソッドを呼び出して，NULL 値かどうかを確認してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_CANNOT_BE_NULL
　引数 lpctParamName，又は引数 xxParam に NULL を指定しています。

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数 dwRow，又は引数 dwIndex の範囲が不正です（Execute 実行前に実行した場合を含みます）。

DB_ERROR_DATA_TRUNCATED
　パラメタの値を指定した型に変換できません。

DB_ERROR_CONVERT_ARRAY_TO_SCALAR
　引数 parData で繰り返し列以外のフィールド値を取得しようとしました，又は引数 parData 以外の引数で繰り返し列の値を取得しようとしました。

GetParamCount メソッド

機能

パラメタの数を取得します。

形式

```
UINT32 GetParamCount(void)
```

引数

なし

戻り値

データ型：UINT32

パラメタの個数。

機能詳細

SQL 文中に含まれる？パラメタの個数を取得します。

GetParamCount メソッドを呼び出す前に，Execute メソッドを呼び出しておく必要があります。

発生する例外

なし

GetResultSet メソッド

機能

SQL 文を実行し，検索したレコードから，ResultSet オブジェクトを生成します。

形式

```
DBResultSet *GetResultSet (void) throw DBSQLCA
```

引数

なし

戻り値

データ型：DBResultSet*

DBResultSet オブジェクトへのポインタ。

機能詳細

?パラメタを付けた SELECT 文の検索結果を取得します。SELECT 文は DBPreparedStatement オブジェクトの生成時に指定したものです。

データベースから SetMaxRows メソッドで指定されたレコード数分レコードを検索後，DBResultSet オブジェクトを生成し，生成したオブジェクトへのポインタを返します。

GetResultSet メソッドを呼び出す前に，Execute メソッドを呼び出し，SetParam メソッドを呼び出してパラメタの値を設定しておきます。SetParam メソッドで値を設定していないパラメタは NULL 値と解釈して実行します。

Execute メソッドで実行した SQL が SELECT 文でない場合は NULL を返します。

BLOB 型データを扱う場合の制限事項については，SetResultSetType メソッド，及び「3.3.4 BLOB 型データの取得方法についての制限」を参照してください。

このメソッドは，非同期実行可能なメソッドです。

XDM/RD の場合

GetResultSet メソッドを呼び出す前に，すべてのパラメタについて SetParamType メソッドでデータ型を設定しておいてください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しました。

DB_ERROR_BEFORE_EXECUTE
　Execute メソッドが実行されていません。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_ERROR_DAB_ACCESS_ERROR
DABroker でエラーが発生しました。
DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR
システムコールでエラーが発生しました。
DBSQLCA クラスで詳細コードを確認してください。

DB_DRV_ERROR_RDA_PARAM_TYPE
パラメタのデータ型が設定されていないか，又は設定されたデータ型が不正です。

DB_ERROR_TIMEOUT
タイムアウトが発生したため制御を戻します。

DB_DRV_ERROR_REQUEST_RESULTSET_ROWS
ResultSet を生成するために指定されたレコード数が不正です。SetMaxRows メソッドで正しいレコード数を指定してください。

GetResultSetMetaData メソッド

機能

DBResultSetMetaData オブジェクトを生成します。

DBResultSetMetaData オブジェクトからフィールドの名称，数，属性などが取得できます。

形式

```
DBResultSetMetaData *GetResultSetMetaData(void) throw DBSQLCA
```

引数

なし

戻り値

データ型：DBResultSetMetaData*

DBResultSetMetaData オブジェクトへのポインタ。

機能詳細

Execute メソッドを実行した後，SELECT 文の検索結果に対する情報を取得します。

DBResultSetMetaData オブジェクトを生成し，生成したオブジェクトへのポインタを返します。取得したポインタは，次回の Execute メソッドが呼ばれた時点で無効になります。

また，実行した SQL 文が SELECT 文でない場合は NULL を返します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
メモリ容量が不足しました。

GetUpdateRows メソッド

機能

SQL 文を使って更新，追加，又は削除したレコード数を取得します。

形式

```
UINT32 GetUpdateRows(void)
```

引数

なし

戻り値

データ型：UINT32

処理されたレコード件数を取得します。

機能詳細

ExecuteUpdate メソッドで実行された UPDATE，DELETE，INSERT 文によって処理されたレコード数を取得します。

発生する例外

なし

InExecute メソッド

機能

DBPreparedStatement オブジェクトに非同期実行中（又は実行待ち）のステートメントがあるかどうかを確認します。

形式

```
BOOLEAN  InExecute(void)
```

引数

なし

戻り値

データ型：BOOLEAN

TRUE：ステートメントが非同期実行中(又は実行待ち)です。

FALSE：非同期実行中(又は実行待ち)ではありません。

機能詳細

InExecute メソッドを実行したオブジェクト内のステートメントが非同期実行中(又は実行待ち）かどうかを確認します。

非同期実行中(又は実行待ち）の場合は TRUE を，非同期実行中(又は実行待ち）でない場合は FALSE を返します。

同期実行接続時は FALSE を返します。

コネクション内のすべてのステートメントを対象に非同期実行中かどうかを確認するには，InWaitForDataSource メソッドを呼び出します(DBConnection クラス)。

発生する例外

なし

IsNull メソッド

機能

パラメタの値が NULL（欠損値）かどうかを確認します。

形式

インデクス番号で指定する場合

```
BOOLEAN IsNull(UINT32 dwIndex) throw DBSQLCA
```

レコード番号・インデクス番号で指定する場合

```
BOOLEAN IsNull(UINT32 dwRow, UINT32 dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まるパラメタの番号を指定します。

dwRow

1 から始まるパラメタのレコード番号を指定します。

戻り値

データ型：BOOLEAN

TRUE：パラメタの値が NULL です。

FALSE：パラメタの値は NULL ではありません。

機能詳細

指定されたパラメタが NULL かどうかを確認します。

NULL は C 言語で使用する NULL ポインタの意味ではなく，データベースシステムでは「値がない」ことを意味します。

NULL であるパラメタに対して GetParam メソッドを呼び出すと，戻される値は意味のない値（0，NULL 文字列，要素がすべて 0，NULL 文字列からなる構造体，又は不定値）となります。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しました。

DB_ERROR_RESULTSET_NOT_EXIST
　非同期実行時に，検索結果がまだ取得できていません。

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数 dwIndex，引数 dwRow の範囲が不正です。

Parent メソッド

機能

対象となるオブジェクトを生成した DBConnection オブジェクトのポインタを取得します。

形式

```
LPOBJECT Parent(void)
```

引数

なし

戻り値

データ型：LPOBJECT

対象となるオブジェクトを生成した DBConnection オブジェクトへのポインタ。

機能詳細

なし

発生する例外

なし

Remove メソッド

機能

DBPreparedStatement オブジェクト(自分自身)を削除します。

形式

```
void Remove(void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

DBPreparedStatement オブジェクト（自分自身）を削除します。

非同期処理中に実行待ち，及び実行中のステートメントがある場合は削除できません。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中ステートメントがあります。

RemoveResultSet メソッド

機能

DBResultSet オブジェクトを削除します。

形式

```
void RemoveResultSet(void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

GetResultSet メソッドで生成した DBResultSet オブジェクトを削除します。

非同期処理中に実行待ち，及び実行中のステートメントがある場合は削除できません。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行中です。

SetInsertRows メソッド

機能

SQL の INSERT 文使用時，1 度に何レコード挿入するかを指定します。

形式

```
void SetInsertRows(UINT32 dwCount = 1) throw DBSQLCA
```

引数

dwCount

INSERT するレコード数を指定します。1 以上を指定してください。

戻り値

なし

機能詳細

1 回の INSERT で挿入するレコード数を指定します。

指定できるレコード数の範囲

指定できるレコード数の最大値は使用している DBMS によって異なります。各 DBMS ごとに指定できるレコード数の最大数を次に示します。

- ORACLE を使用している場合：最大 32767 レコード
- ORACLE 以外の DBMS を使用している場合：
  「1 レコードのパラメタ数×INSERT するレコード数 ≦ 32767」を満たす値

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数 dwCount で指定した値が不正（1 未満）です。

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_ERROR_DAB_ACCESS_ERROR
　DABroker でエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR
　システムコールでエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

SetMaxFieldSize メソッド

機能

アプリケーションで受け取るフィールドの長さを指定します。

形式

フィールド名で指定する場合

```
void SetMaxFieldSize(LPCTSTR lpctFieldName,
                     UINT32 dwMaxSize=0)    throw DBSQLCA
```

インデクス番号で指定する場合

```
void SetMaxFieldSize(UINT32 dwIndex,
                     UINT32 dwMaxSize=0)    throw DBSQLCA
```

引数

lpctFieldName

フィールド名を指定します。

dwIndex

1 から始まるフィールドの番号を指定します。

dwMaxSize

アプリケーションで受け取るフィールドの長さを, 0 以上のバイト数で指定します。この引数で指定された値が，データベースでの定義長よりも大きい場合，引数で指定した値は無視されます。

0 を指定すると，データベース中の定義長が仮定されます。

戻り値

なし

機能詳細

アプリケーションで受け取るフィールドの長さを, バイト数で指定します。アプリケーションで必要とするデータがフィールド値全体でなく, フィールド値の一部である場合に利用します。データベース中のフィールドの定義長ではないことに注意してください。

このメソッドを呼び出す前に，Execute メソッドを呼び出しておく必要があります。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_BEFORE_EXECUTE
: Execute メソッドが実行されていません。

DB_ERROR_IN_ASYNC_EXECUTE
: ステートメントが非同期実行処理中です。

DB_ERROR_ARGUMENT_OUT_OF_RANGE
: 引数 dwIndex が 1 より小さい値です。

DB_ERROR_CANNOT_BE_NULL
: 引数 lpctFieldName に NULL が指定されています。

DB_DRV_ERROR_INVALID_ARGUMENT
: 指定した引数は不正です。

DB_DRV_ERROR_INVALID_MAX_SIZE
　バッファの最大長の指定値が不正です。

SetMaxRows メソッド

機能

ResultSet に検索するレコード数の最大値を指定します。

形式

```
void SetMaxRows(UINT32 dwMaxSize=MAX_ROWS_DEFAULT) throw DBSQLCA
```

引数

dwMaxSize

ResultSet に検索するレコード数の最大値を指定します。必ず 1 以上を指定してください。

システムデフォルト値は，MAX_ROWS_DEFAULT です（=100 が仮定されます）。

戻り値

なし

機能詳細

DBResultSet オブジェクトで一度に取得できる最大レコード数を指定します。

一度の読み込み（GetResultSet メソッド，又は PageNext メソッドの呼び出し）で，データベースから取得するレコード数の最大値を設定します。

指定できるレコード数の範囲

指定できるレコード数の最大値は使用している DBMS によって異なります。各 DBMS ごとに，指定できるレコード数の最大値を次に示します。

- ORACLE を使用している場合：32767 レコード以下
- ORACLE 以外の DBMS を使用している場合：4096 レコード以下

1 レコードのフィールド数は，DBResultSetMetaData オブジェクトの GetColumnCount メソッドを呼び出して取得できます。

更新可能な DBResultSet オブジェクトでの扱い

DBResultSet オブジェクトが更新可能なオブジェクトとして生成された場合(SetResultSetType メソッドの引数 swType で TYPE_EXCLUSIVE を指定した場合)，SetMaxRows メソッドによる指定は無視され，常に 1 が仮定されます。更新可能な DBResultSet は常に 1 レコードだけを読み込みます。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数 dwMaxSize に範囲外（1 より小さい）が指定されています。

SetNull メソッド

機能

指定されたパラメタに NULL（欠損値)を設定します。

形式

インデクス番号で指定する場合

```
void SetNull(UINT32 dwIndex) throw DBSQLCA
```

レコード番号・インデクス番号で指定する場合

```
void SetNull(UINT32 dwRow, UINT32 dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まるフィールドの番号を指定します。

dwRow

1 から始まるパラメタのレコード番号を指定します。

戻り値

なし

機能詳細

指定されたパラメタに NULL を設定します。

NULL は C 言語で使用する NULL ポインタの意味ではなく，データベースシステムでは「値がない」ことを意味します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数 dwIndex，引数 dwRow の範囲が不正です。
DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

SetParam メソッド

機能

？パラメタに値を設定します。

形式

インデクス番号で指定する場合

```
void SetParam(UINT32 dwIndex, LPCTSTR lpctParam) throw DBSQLCA
void SetParam(UINT32 dwIndex, INT16 sParam) throw DBSQLCA
void SetParam(UINT32 dwIndex, INT32 lParam) throw DBSQLCA
void SetParam(UINT32 dwIndex, UINT16 swParam) throw DBSQLCA
void SetParam(UINT32 dwIndex, UINT32 dwParam) throw DBSQLCA
void SetParam(UINT32 dwIndex, SINGLE sfParam) throw DBSQLCA
void SetParam(UINT32 dwIndex, DOUBLE dfParam) throw DBSQLCA
void SetParam(UINT32 dwIndex, const DBR_DATETIME& dtParam)
                                              throw DBSQLCA
void SetParam(UINT32 dwIndex, const DBR_BINARY& blobParam)
                                              throw DBSQLCA
void SetParam(UINT32 ui32Index, const DBRArrayDataConstPtr&
                                    cpArray) throw DBSQLCA
```

レコード番号・インデクス番号で指定する場合

```
void SetParam(UINT32 dwRow, UINT32 dwIndex, LPCTSTR lpctParam)
                                              throw DBSQLCA
void SetParam(UINT32 dwRow, UINT32 dwIndex, INT16 sParam)
                                              throw DBSQLCA
void SetParam(UINT32 dwRow, UINT32 dwIndex, INT16 sParam)
                                              throw DBSQLCA
void SetParam(UINT32 dwRow, UINT32 dwIndex, UINT16 swParam)
                                              throw DBSQLCA
void SetParam(UINT32 dwRow, UINT32 dwIndex, UINT32 dwParam)
                                              throw DBSQLCA
void SetParam(UINT32 dwRow, UINT32 dwIndex, SINGLE sfParam)
                                              throw DBSQLCA
void SetParam(UINT32 dwRow, UINT32 dwIndex, DOUBLE dfParam)
                                              throw DBSQLCA
void SetParam(UINT32 dwRow, UINT32 dwIndex, DOUBLE dfParam)
                                              throw DBSQLCA
void SetParam(UINT32 dwRow, UINT32 dwIndex,
                const DBR_DATETIME& dtParam) throw DBSQLCA
void SetParam(UINT32 dwRow, UINT32 dwIndex,
                const DBR_BINARY& blobParam) throw DBSQLCA
void SetParam(UINT32 ui32Row, UINT32 ui32Index,
         const DBRArrayDataConstPtr& cpArray) throw DBSQLCA
```

引数

指定されたパラメタを，データベースの型にキャストして設定します。

dwIndex

1 から始まるパラメタの番号を指定します。指定するパラメタの番号は，SQL 文中での出現順に割り当てられます。

dwRow

1 から始まるパラメタを設定するレコード番号を指定します。

xxParam（データ型：インデクス番号で指定する場合の第 2 引数，レコード番号・インデクス番号で指定する場合の第 3 引数を参照してください）

パラメタに設定する値を指定します。

cpArray

インスタンスを保持している DBRArrayDataConstPtr オブジェクトを指定します。

戻り値

なし

機能詳細

指定されたパラメタの値を設定します。

値は指定するデータ型からデータベース固有の型に変換されます。データ型変換規則については，「7.1 クラスライブラリで扱うデータ型と変換規則」の SetParam メソッドでのデータ型変換規則を参照してください。

SetParam メソッドは ExecuteUpdate メソッドの前，Execute と GetResultSet の間に実行してください。

パラメタに欠損値を設定したい場合は，SetNull メソッドを使用してください。

引数 lpctParam が NULL（LPCTSTR 型の変数に NULL を代入して引数に渡した場合，又は NULL を LPCTSTR 型にキャストした場合）の時も欠損値が設定されます。キャストしないで引数に NULL だけ指定した場合は欠損値とはみなしません。

BLOB 型データをファイル経由で取得する場合

SetResultSetType メソッドで TYPE_BLOB_FILE_xx を指定した場合，BLOB 型のフィールドに対する SetParam メソッドの引数にはファイル名称を指定します。SetParamType メソッドは SetParam メソッドの前に実行してください。

DBR_BINARY 型を指定する場合

DBR_BINARY 型の各メンバには次の値を設定します。

Length：設定するデータの長さ（単位はバイト）

RealLength：設定不要

Data：実際のデータの領域を指すポインタ

レコード番号の指定

1 レコードだけを挿入する場合には，インデクス番号と設定するデータだけを指定します。複数のレコードを一度に挿入する場合，DABroker で確保された領域のどのレコード番号に対する処理なのかをアプリケーションから指定する必要があります。引数 dwRow には SetParam を実行するレコード番号を，SetInsertRows メソッドで設定した範囲内の値で指定します。

繰り返し列を扱う場合の注意

引数 cpArray で指定する DBRArrayDataConstPtr オブジェクトがインスタンスを保持していない場合は，エラーになります。インスタンスを保持する DBRArrayDataPtr オブジェクトは，次の手順で生成し，引数へ設定します。

1. DBRArrayDataFactory オブジェクトの CreateArrayData メソッドで，DBRArrayData オブジェクトを生成します。
2. 新しく繰り返し列を作成する場合だけ，DBRArrayData オブジェクトの Create メソッドで要素の格納領域を確保します。
3. DBRArrayData オブジェクトの SetData メソッドで要素の値を設定します。

4. 引数 pArray に DBRArrayDataPtr オブジェクトを設定し，SetParam メソッドを呼び出します。
5. ExecuteUpdate メソッドで実行します。

なお，SetParam メソッドで値を設定した後に，DBRArrayData オブジェクトの SetData メソッドを実行すると，SetParam メソッド実行時点のデータは失われ，SetData メソッドに指定した値が設定されるため注意してください。

また，Update メソッドを実行中に，DBRArrayData オブジェクトの SetData メソッドは呼び出せません。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数 dwRow，又は引数 dwIndex の範囲が不正です（Execute 実行前に実行した場合を含みます）。

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

DB_ERROR_CANNOT_BE_NULL
　引数に NULL は指定できません。

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しました。

DB_ERROR_NO_INSTANCE
　引数 cparArray に指定されたオブジェクトが, DBRArrayDataPtr クラスのインスタンスを持っていません。

DB_ERROR_CANNOT_ACCESS_WHILE_EXECUTED
　Update メソッドを実行中に，DBRArrayData オブジェクトの SetData メソッドを呼び出しました。

SetParamType メソッド

機能

？パラメタの属性を指定します。

形式

```
void SetParamType(UINT32 dwIndex,
                  UINT16 ui16Type = TYPE_BLOB_MEMORY) throw DBSQLCA
```

引数

dwIndex

1 から始まるパラメタの番号を指定します。

ui16Type

パラメタの属性を指定します。 指定できる値を次に示します。

- TYPE_BLOB_MEMORY：メモリ上のデータを長大データとして読み込みます。
- TYPE_BLOB_FILE_TEXT：テキストファイルを長大データとして読み込みます。

- TYPE_BLOB_FILE_BINARY：バイナリファイルを長大データとして読み込みます。

RDA Link for Gateway を使用している場合にだけ指定できる値を次に示します。

- RDA_DT_XCHAR
- RDA_DT_NUM_TRAILING
- RDA_DT_NUM_UNSIGNED
- RDA_DT_FLOAT
- RDA_DT_SMALLFLT
- RDA_DT_DECIMAL
- RDA_DT_LARGE_DECIMAL
- RDA_DT_INTEGER
- RDA_DT_SMALLINT
- RDA_DT_MVARCHAR
- RDA_DT_MCHAR
- RDA_DT_LONG_MVARCHAR
- RDA_DT_NVARCHAR
- RDA_DT_NCHAR
- RDA_DT_LONG_NVARCHAR
- RDA_DT_VARCHAR
- RDA_DT_CHAR
- RDA_DT_LONG_VARCHAR
- RDA_DT_DATE

～LONG_(M,N)VARCHAR はメモリ上のデータを長大データとして読み込みます。

戻り値

なし

機能詳細

パラメタの属性を指定します。

BLOB 型のフィールドに対してファイル経由でのパラメタ設定をしたい場合だけ，SetParamType メソッドを TYPE_BLOB_FILE_TEXT，又は TYPE_BLOB_FILE_BINARY オプションを指定して呼び出します。この場合，SetParam では「値」として「ファイル名」を指定する必要があります。詳細については，SetParam メソッドを参照してください。

SetParamType メソッドは SetParam メソッドを呼び出す前に実行してください。SetParam メソッドで値を設定した後に SetParamType メソッドを実行した場合，そのパラメタの値はクリアされます。

HiRDB の Binary 型の場合

TYPE_BLOB_FILE_TEXT および TYPE_BLOB_FILE_BINARY は指定できません。メモリ経由でのパラメタ設定だけ実行できます。

XDM/RD の場合

ExecuteUpdate メソッド，又は GetResultSet メソッドを呼び出す前に，すべてのパラメタについてこのメソッドを呼び出してパラメタのデータ型を指定してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数 dwIndex の範囲が不正です（Execute メソッド実行前に実行した場合を含みます)。

DB_DRV_ERROR_SQL_NOT_FOUND
　SQL 文が設定されていません。

SetResultSetType メソッド

機能

DBResultSet オブジェクトの生成オプションを指定します。

DBResultSet オブジェクトを生成するときに，参照専用，又は更新可能のどちらで生成するか選択します。検索レコードに対するロック方法を指定します。

検索時にバッファを幾つ使うかも指定できます。

また，BLOB 型データを扱う場合，その取得方法についても指定します。

形式

```
void SetResultSetType (UINT16
                  swType = TYPE_EXCLUSIVE|BUFFER_TYPE_SINGLE,
                  DBR_BLOB_TYPE nBLOBType = TYPE_BLOB_MEMORY,
                  LPCTSTR lpctBLOBFileName = NULL) throw DBSQLCA
```

引数

swType

次の三つの値を指定します。

- 検索したレコードに対するロック方法
- 検索時に使用するバッファ数
- VARCHAR データを DBR_BINARY 型で取得した場合の Length メンバ，RealLength メンバの意味

指定する値は，上記三つの値の論理和になります。

このメソッドを実行しなかった場合の動作は，TYPE_EXCLUSIVE | BUFFER_TYPE_SINGLE | VARCHAR_LENGTH_DEF を指定した場合と同じになります。

検索したレコードに対するロック方法
　次に示す TYPE_EXCLUSIVE，TYPE_EXCLUSIVE2，TYPE_NONE，TYPE_WAIT，TYPE_NOWAIT，TYPE_SHARED のどれかの値で，検索したレコードに対するロック方法を指定します。

- TYPE_EXCLUSIVE 又は TYPE_EXCLUSIVE2

  DBResultSet オブジェクトからレコードの追加，更新，削除をする場合は，このオプションを指定します。検索したレコードに排他ロックが掛かり，他のトランザクションからの参照・更新を制限できます。実際に排他ロックが掛かるのは，GetResultSet メソッドを実行した時点からです。

  このオプションで生成した DBResultSet オブジェクトでは，一度に複数のレコードを読み込めないため，SetMaxRows で指定したレコード数は無視されます。また，検索条件に一致する次のレコードをアクセスするには，Next メソッドを使用します。TYPE_EXCLUSIVE，又は TYPE_EXCLUSIVE2 で生成したオブジェクトの場合，カーソル制御のメソッドとしては Next メソッドだけが指定できます。Next メソッドについては，「5.7 DBResultSet クラスの詳細」を参照してください。

  なお，この引数の値は，実行時に SQL 文のオプション文字列に変換されて実行されます。DBMS 別に，どのオプション文字列へ変換されて実行されるのかを次に示します。なお，HiRDB 以外の DBMS では TYPE_EXCLUSIVE と TYPE_EXCLUSIVE2 で同じオプション文字列へ変換されて実行されます。

  - ORACLE：FOR UPDATE
  - SQL Anywhere,Adaptive Server Anywhere：オプション文字列は付加しません。
  - HiRDB

    TYPE_EXCLUSIVE：FOR UPDATE

    TYPE_EXCLUSIVE2：WITH EXCLUSIVE LOCK FOR UPDATE
  - XDM/RD：WITH EXCLUSIVE LOCK FOR UPDATE
  - SQL Server：UPDLOCK
  - SQL/K：FOR UPDATE

    詳細については，マニュアル「SQL/K」を参照してください。
  - XDM/SD：LOCK SU
- TYPE_NONE

  参照専用の DBResultSet オブジェクトを生成します。検索するレコードに対して，参照ロックを掛けます。

  このオプションで生成した DBResultSet オブジェクトでは，複数レコードを読み込めます。

  なお，この引数の値は，実行時に SQL 文のオプション文字列に変換されて実行されます。DBMS 別に，どのオプション文字列へ変換されて実行されるのかを次に示します。

  - ORACLE：付加しません。
  - SQL Anywhere，Adaptive Server Anywhere：付加しません。
  - HiRDB：付加しません。
  - XDM/RD：付加しません。
  - SQL Server：付加しません。
  - SQL/K：付加しません。
  - XDM/SD：付加しません(SDEXCLUSIVE 値が仮定されます)。

    SDEXCLUSIVE 値の詳細については，マニュアル「Database Connection Server」のコントロール空間起動制御文を参照してください。
- TYPE_WAIT(HiRDB，XDM/RD，SQL Server，SQL/K，XDM/SD の場合に有効です)

  参照専用の DBResultSet オブジェクトを生成します。

  検索したレコードに対して参照ロックを掛けますが，見終わった行から排他制御を解除します。このため，DABroker for C++では，ResultSet にレコードをすべて読み込んだ時点，つまり，GetResultSet メソッド，又は PageNext メソッドが完了した時点で，ロックが解除されています。

このオプションで生成した DBResultSet オブジェクトでは，複数レコードを読み込めます。

なお，この引数の値は，実行時に SQL 文のオプション文字列に変換されて実行されます。DBMS 別に，どのオプション文字列へ変換されて実行されるのかを次に示します。

- HiRDB：WITHOUT LOCK WAIT
- XDM/RD：WITHOUT LOCK WAIT
- SQL Server：HOLDLOCK
- SQL/K：付加しません。
- XDM/SD：LOCK SR

- TYPE_NOWAIT(HiRDB，XDM/RD，SQL Server，SQL/K，XDM/SD の場合に有効です)

  参照専用の DBResultSet オブジェクトを生成します。該当するレコードをほかのトランザクションが更新中だったり，TYPE_EXCLUSIVE で検索している場合でも読み込めます。排他制御を全くしません。

  このオプションで生成した DBResultSet オブジェクトでは，複数レコードを読み込めます。

  なお，この引数の値は，実行時に SQL 文のオプション文字列に変換されて実行されます。DBMS 別に，どのオプション文字列へ変換されて実行されるのかを次に示します。

  - HiRDB：WITHOUT LOCK NOWAIT
  - XDM/RD：WITHOUT LOCK NOWAIT
  - SQL Server：NOLOCK
  - SQL/K：付加しません。
  - XDM/SD：LOCK NR

- TYPE_SHARED(HiRDB，XDM/RD，SQL/K，XDM/SD の場合だけ有効です)

  検索するレコードに共有ロックを掛けます。共有ロックの掛かったレコードは，ほかのトランザクションから参照できますが，更新はできません。

  - HiRDB：WITH SHARE LOCK
  - XDM/RD：WITH SHARE LOCK
  - SQL/K：付加しません。
  - XDM/SD：LOCK SR

検索時に使用するバッファ数

次に示す BUFFER_TYPE_SINGLE，又は BUFFER_TYPE_DOUBLE のどちらかの値で，検索時に使用するバッファ数を指定します。指定を省略した場合は，BUFFER_TYPE_SINGLE が仮定されます。なお，バッファの利用方法については，「3.3.3 検索性能の向上策」を参照してください。

- BUFFER_TYPE_SINGLE

  検索結果を保持するバッファを一つだけ使います。

- BUFFER_TYPE_DOUBLE

  検索結果を保持するバッファを二つ使います。

VARCHAR データを DBR_BINARY 型で取得したときの Length メンバ，RealLength メンバの意味

次に示す値のどれか一つを指定します。

- VARCHAR_LENGTH_DEF

- GetField メソッドの形式 2 を使用した場合，又は形式 3 を使用してもデータの切り捨てが発生しなかった場合

  Length：カラムの定義長※

  RealLength：0

- GetField メソッドの形式 3 を使用してデータの切り捨てが発生した場合

  Length：切り捨てたデータ長

  RealLength：カラムの定義長※

  注※　SetMaxFieldSize メソッドで取得データ長を指定している場合は，「カラムの定義長」ではなく「SetMaxFieldSize メソッドで指定したサイズ」となります。

- VARCHAR_LENGTH_REAL
  - GetField メソッドの形式 2 を使用した場合，又は形式 3 を使用してもデータの切り捨てが発生しなかった場合

    Length：実際のデータ長※

    RealLength：0

  - GetField メソッドの形式 3 を使用してデータの切り捨てが発生した場合

    Length：切り捨てたデータ長

    RealLength：実際のデータ長※

    注※　SetMaxFieldSize メソッドでカラムの定義長よりも小さいサイズを指定している場合は，データベースから取得したデータは実際のデータよりも小さい場合があるため，「実際のデータ長」ではなく「取得データ長」となります。

nBLOBType

BLOB 型データの取得方法を次のどちらかで指定します。

- TYPE_BLOB_MEMORY

  メモリ上に一括して取得します。

  ResultSet のレコード数が 1 の場合，BLOB 型データをすべて取得します。

  ResultSet のレコード数が複数の場合，先頭から 32K バイトまでを取得します。

- TYPE_BLOB_FILE

  ファイルを経由して取得します。

  引数 lpctBLOBFileName に指定された名称が，BLOB 型データを格納するファイルのプレフィックスとして使用されます。

  なお，引数 swType で BUFFER_TYPE_DOUBLE を指定している場合，検索の途中で終了しても先読みしている ResultSet 分のファイルは作成されます。

lpctBLOBFileName

BLOB 型データを格納するファイルのプレフィックスを指定します。

引数 nBLOBType で TYPE_BLOB_FILE を指定した場合，BLOB 型データを格納するファイルのプレフィックスには，この引数で指定した名称が使われます。TYPE_BLOB_FILE 以外の値を指定した場合は，この引数での指定は無視されます。

BLOB 型データを格納したファイル名称は，「プレフィックス+レコード番号+フィールド番号」で表されます。NULL が指定された場合，「DBConnection の名称+DBPreparedStatement の名称」が仮定されます。

戻り値

なし

機能詳細

検索レコードに対するロック方法，検索時にバッファを幾つ使うか，及び BLOB 型データの取得方法について指定します。

指定した各引数の値は，次に GetResultSet メソッドを実行したときから有効になります。

ファイル上で BLOB 型データを扱う場合の制限事項

データの格納されているファイルは，DABroker が終了しても削除されません。アプリケーション自身で削除してください。

作成されたファイルのアクセス権限は，アプリケーションの実行ユーザ及びデフォルトの権限に依存します。例えば，HP-UX 上で"user1"が実行しているアプリケーションで，デフォルト umask が 0666 の場合，取得した BLOB 型データが格納されるファイルも，"user1"の所有する 0666 のアクセス権限が付与されたファイルとして生成されます。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

WaitForDataSource メソッド

機能

DBPreparedStatement オブジェクトで要求した実行待ち，及び実行中の非同期処理が終了するまで待ちます。非同期処理中に同期を取りたい場合に利用できます。

形式

```
BOOLEAN WaitForDataSource(UINT32 swWaitTime = DBR_INFINITE)
```

引数

swWaitTime

非同期処理の終了を待つための最大時間（単位：ミリ秒），又は DBR_INFINITE を指定します。

指定した時間内に非同期処理が終了した場合，TRUE が返ります。指定した時間を経過しても非同期処理が終了しない場合，FALSE が返ります。非同期実行中の SQL がない場合は，TRUE が返ります。

DBR_INFINITE を指定した場合はタイムアウト時間を設定しません。非同期処理が終了するまで待ち続けます。

同期実行時はすぐに TRUE を返します。

戻り値

データ型：BOOLEAN

TRUE：すべての非同期処理が終了しました。

FALSE：タイムアウト時間が経過しました。

機能詳細

DBPreparedStatement オブジェクトで要求した実行待ち，及び実行中の非同期処理(SQL)が，終了するのを待ちます。

DBConnection クラスの WaitForDataSource メソッドとの違い

DBPreparedStatement オブジェクトの WaitForDataSource メソッドでは，WaitForDataSource メソッドを実行した DBPreparedStatement オブジェクトの非同期実行処理が終了するまで待ちます。ほかのオブジェクトの非同期実行処理については待ちません。そのため，ほかのオブジェクトの非同期実行処理が実行中でも，WaitForDataSource メソッドを呼び出したオブジェクトの非同期実行処理が終了すれば，WaitForDataSource メソッドは TRUE を返します。

同じ DBConnection オブジェクトで実行しているすべての非同期実行処理が終了するのを待ちたい場合は，DBConnection クラスの WaitForDataSource メソッドを実行してください。

発生する例外

なし

5.10　DBCallableStatement クラスの詳細

ストアドプロシジャの実行を管理します。

このクラスは，ストアドプロシジャの実行やその結果の取得などのメソッドを提供するクラスです。ストアドプロシジャについては，「1.6.11 ストアドプロシジャの利用」，又は「2.6.6 ストアドプロシジャの利用」を参照してください。

ストアドプロシジャを指定できる DBMS

- ORACLE
- HiRDB
- SQLAnywhere，Adaptive Server Anywhere
- SQL Server

ストアドプロシジャへの引数設定と実行結果の受け取り

引数を持つプロシジャでは，IN，OUT，IN/OUT のどの属性の引数に対しても？パラメタを使えます。IN，又は IN/OUT の引数に対しては，Execute メソッドで実行する前に，SetParam メソッドで値を設定します。

プロシジャの実行後，IN/OUT，又は OUT の引数に格納された結果を取得するためには GetParam メソッドを使います。

SQL Anywhere，Adaptive Server Anywhere では,実行結果を GetResultSet メソッドで生成した DBResultSet オブジェクトとして受け取ることもできます。

なお，GetParam メソッドは，DBPreparedStatement クラスでは SetParam メソッドで設定した値を取得するメソッドであるのに対し，DBCallableStatement クラスでは，IN/OUT，又は OUT 引数に出力されたストアドプロシジャの実行結果を取得するメソッドです。

機 能	メソッド名
ストアドプロシジャを実行します。	Execute
SQL 文の非同期実行時のエラー情報を得るために，DBSQLCA オブジェクトへのポインタを取得します。	GetErrorStatus
SetMaxFieldSize メソッドで設定したデータ長を取得します。	GetMaxFieldSize
ResultSet に検索できるレコード数の最大値を取得します。	GetMaxRows
DBCallableStatement オブジェクトの名前を取得します。	GetName
IN/OUT,OUT パラメタの数を取得します。	GetOutputParams
ストアドプロシジャの実行結果を取得します。	GetParam
IN,IN/OUT パラメタの数を取得します。	GetParamCount
ストアドプロシジャを実行し，検索したレコードから，ResultSet オブジェクトを生成します。	GetResultSet
DBCallableStatement オブジェクトに非同期実行中（又は実行待ち）のステートメントがあるかどうかを確認します。	InExecute
ストアドプロシジャの実行が完了したかどうかを確認します。	IsCompleted
IN/OUT,OUT パラメタの値が NULL（欠損値）かどうか確認します。	IsNull

機 能	メソッド名
対象となるオブジェクトを生成した DBConnection オブジェクトのポインタを取得します。	Parent
DBCallableStatement オブジェクトを削除します。	Remove
指定されたパラメタの値に NULL（欠損値)を設定します。	SetNull
DBResultSet オブジェクトを削除します。	RemoveResultSet
ストアドプロシジャの実行を再開します。	Resume
アプリケーションで受け取るフィールドの長さを指定します。	SetMaxFieldSize
ResultSet に検索するレコード数の最大値を指定します。	SetMaxRows
IN/OUT,IN パラメタの値を設定します。	SetParam
引数で指定したストアドプロシジャ名をデータベースに通知します。	SetProcedure
DBResultSet オブジェクトの生成オプションを指定します。	SetResultSetType
DBCallableStatement オブジェクトで要求した実行待ち，及び実行中の非同期処理が終了するまで待ちます。	WaitForDataSource

Execute メソッド

機能

ストアドプロシジャを実行します。

形式

```
void Execute (void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

SetProcedure で設定されているストアドプロシジャを実行します。

このメソッドは非同期実行可能なメソッドです。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_ERROR_PROCEDURE_NOT_SET
　実行するプロシジャが設定されていません。

DB_ERROR_DAB_ACCESS_ERROR
　DABroker でエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR
　システムコールでエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_DRV_ERROR_PHOLDER_AND_QP
　プレースホルダ，及び？パラメタが混在する SQL 文は指定できません。

DB_ERROR_TIMEOUT
　タイムアウトが発生したため制御を戻します。

GetErrorStatus メソッド

機能

SQL 文の非同期実行時のエラー情報を得るために，DBSQLCA オブジェクトへのポインタを取得します。

形式

```
DBSQLCA *GetErrorStatus(void)
```

引数

なし

戻り値

データ型：DBSQLCA*

DBSQLCA オブジェクトへのポインタ。

機能詳細

DBSQLCA オブジェクトへのポインタを取得します。

非同期処理のときは，エラーが発生しても直にエラーを取得できません。このため，ユーザは任意の時点でエラーを確認する必要があります。

非同期実行時にエラーが発生した場合，DBSQLCA オブジェクトにエラー情報が設定されます。

DBSQLCA オブジェクトでは，非同期実行中に発生したエラー情報を最大 100 回分保存できます。エラー情報は，新しいエラー情報から 100 回分保存されるため，エラー情報を確認したあとは DBSQLCA クラスの Delete メソッドを呼び出して，不要なエラー情報をクリアしておいてください。

発生する例外

なし

GetMaxFieldSize メソッド

機能

SetMaxFieldSize メソッドで設定したデータ長を取得します。

形式

フィールド名で指定する場合

```
UINT32   GetMaxFieldSize(LPCTSTR lpctFieldName = NULL) throw DBSQLCA
```

インデクス番号で指定する場合

```
UINT32   GetMaxFieldSize(UINT32 dwIndex) throw DBSQLCA
```

引数

lpctFieldName

フィールドの名称，又は NULL を指定します。

NULL が指定された場合，すべてのフィールド長の合計値をバイト数で取得します。

dwIndex

1 から始まるフィールドの番号を指定します。

戻り値

データ型：UINT32

SetMaxFieldSize メソッドで設定したデータ長。

機能詳細

SetMaxFieldSize メソッドで設定したデータの長さを指定します。データベース中のフィールドの定義長ではないことに注意してください。

取得するデータがフィールドの値を格納する領域の最大長よりも大きい場合，データは切り捨てられます。

このメソッドを呼び出す前に，Execute メソッドを呼び出しておく必要があります。

不定長のフィールドの場合は，0 を返します。

このメソッドは，アクセスする DBMS が SQL Anywhere，Adaptive Server Anywhere の場合だけ利用できます。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_PROCEDURE_NOT_SET
　実行するプロシジャが指定されていません。

DB_DRV_ERROR_INVALID_ARGUMENT
　引数に指定した値は不正です。

GetMaxRows メソッド

機能

ResultSet に検索できるレコード数の最大値を取得します。

形式

```
UINT32 GetMaxRows(void)
```

引数

なし

戻り値

テータ型：UINT32

SetMaxRows メソッドで設定した，レコード数の最大値を取得します。

機能詳細

SetMaxRows メソッドで設定した，DBResultSet オブジェクトで取得できるレコード数の最大値を取得します。SetMaxRows メソッドを一度も呼び出していない場合はデフォルト値の 100 を返します。

このメソッドは，アクセスする DBMS が SQL Anywhere，Adaptive Server Anywhere の場合だけ利用できます。

発生する例外

なし

GetName メソッド

機能

DBCallableStatement オブジェクトの名前を取得します。

形式

```
LPCTSTR GetName(void)
```

引数

なし

戻り値

データ型：LPCTSTR

DBCallableStatement オブジェクトの名前が返ります。

機能詳細

DBCallableStatement オブジェクトの名前を取得します。DBCallableStatement オブジェクトの名称は，DBConnection クラスの CreateCallableStatement メソッドの引数から生成されます。

発生する例外

なし

GetOutputParams メソッド

機能

IN/OUT，OUT パラメタの数を取得します。

形式

```
UINT32 GetOutputParams(void) throw DBSQLCA
```

引数

なし

戻り値

データ型：UINT32

指定したプロシジャの IN/OUT，OUT パラメタの個数です。

機能詳細

SetProcedure メソッドで指定したプロシジャの IN/OUT，OUT パラメタの個数を取得します。

リターン値の戻るプロシジャの場合，戻り値にはリターン値の分も含みます。

使用する DBMS が SQL Anywhere，Adaptive Server Anywhere，SQL Server の場合は，このメソッドで返す値は，リターン値の有無にかかわらず常に IN/OUT，OUT パラメタの個数＋ 1 を返します。

これ以外の DBMS の場合は，常に IN/OUT，OUT パラメタの個数だけが取得できます。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_PROCEDURE_NOT_SET
　実行するプロシジャが指定されていません。

GetParam メソッド

機能

ストアドプロシジャの実行結果を引数で取得します。

形式１：ユーザ確保領域へコピーする場合

インデクス番号指定

```
void GetParam(UINT32 dwIndex, INT16 * sParam) throw DBSQLCA
void GetParam(UINT32 dwIndex, INT32 * lParam) throw DBSQLCA
void GetParam(UINT32 dwIndex, UINT16* swParam) throw DBSQLCA
void GetParam(UINT32 dwIndex, UINT32* dwParam) throw DBSQLCA
void GetParam(UINT32 dwIndex, SINGLE* sfParam) throw DBSQLCA
void GetParam(UINT32 dwIndex, DOUBLE* dfParam) throw DBSQLCA
void GetParam(UINT32 dwIndex, DBR_DATETIME* dtParam)
                                                throw DBSQLCA
```

フィールド名指定

```
void GetParam(LPCTSTR lpctParamName, INT16 * sParam)
                                                throw DBSQLCA
void GetParam(LPCTSTR lpctParamName, INT32 * lParam)
                                                throw DBSQLCA
void GetParam(LPCTSTR lpctParamName, UINT16* swParam)
                                                throw DBSQLCA
void GetParam(LPCTSTR lpctParamName, UINT32* dwParam)
                                                throw DBSQLCA
void GetParam(LPCTSTR lpctParamName, SINGLE* sfParam)
                                                throw DBSQLCA
void GetParam(LPCTSTR lpctParamName, DOUBLE* dfParam)
                                                throw DBSQLCA
void GetParam(LPCTSTR lpctParamName, DBR_DATETIME* dtParam)
                                                throw DBSQLCA
```

形式２：DABroker が用意した領域を参照する場合

インデクス番号指定

```
void GetParam(UINT32 dwIndex, LPTSTR* lptParam) throw DBSQLCA
void GetParam(UINT32 dwIndex, DBR_BINARY* blobParam)
                                                throw DBSQLCA
```

フィールド名指定

```
void GetParam(LPCTSTR lpctParamName, LPTSTR* lptParam)
                                                throw DBSQLCA
void GetParam(LPCTSTR lpctParamName, DBR_BINARY* blobParam)
                                                throw DBSQLCA
```

形式３：ユーザ確保領域へコピーする場合

インデクス番号指定

```
INT32 GetParam(UINT32 dwIndex, LPTSTR lptParam, UINT32 dwSize)
                                                throw DBSQLCA
INT32 GetParam(UINT32 dwIndex, DBR_BINARY* blobParam,
                                UINT32 dwSize) throw DBSQLCA
```

フィールド名指定

```
UINT32 GetParam(LPCTSTR lpctParamName, LPTSTR lptParam,
                                UINT32 dwSize) throw DBSQLCA
UINT32 GetParam(LPCTSTR lpctParamName, DBR_BINARY* blobParam,
                                UINT32 dwSize) throw DBSQLCA
```

引数

dwIndex

1 から始まるパラメタの番号を指定します。指定するパラメタの番号は，プロシジャのパラメタの定義順に割り当てられます。

*xxParam（形式 1，3 の第 2 引数）

データをコピーする領域のポインタを指定します。

*xxParam（形式 2 の第 2 引数）

ポインタ変数を指定します。

dwSize

コピー先の文字列領域の大きさをバイト数で指定します。

lpctParamName

プロシジャを作成したときのパラメタの名前を指定します。パラメタの名前は，プロシジャで定義した名前を使用してください。

戻り値

データ型：UINT32

コピーした文字列の長さが返ります。パラメタの値を指定された型にキャストして取得します（形式 3 の場合）。

機能詳細

IN/OUT，OUT 引数に出力された結果（パラメタ値）を取得します。このメソッドはプロシジャの実行が終了してから実行してください。プロシジャの実行が終了したかどうかを確認するには，IsCompleted メソッドを利用できます。

形式 1 の処理

パラメタ値を，引数に指定されたポインタが指す領域に格納します。

形式 2 の処理

DABroker が確保した領域にパラメタ値をコピーし，その先頭アドレスをポインタ変数に設定します。このため，LPTSTR または DBR_BINARY の変数の確保だけで，パラメタ値をコピーする領域を確保する必要はありません。

- パラメタ値の有効期間
  パラメタ値は，Execute メソッドや SetProcedure メソッドを実行するまで，又は取得したパラメタに対して SetParam を実行するまで有効です。

形式 3 の処理

- LPTSTR 型
  引数 lptParam の指す領域は，引数 dwSize バイト分確保されている必要があります。
  - 引数 dwSize がデータベースから取得したデータの長さ以下の場合

プロシジャのデータ型が可変長文字列型の場合，引数 dwSize-1 分のデータ（最後には NULL 終端文字が入る）がコピーされ，残りのデータは切り捨てられます。

プロシジャのデータ型が固定長文字列の場合は，引数 dwSize 分のデータ（最後に NULL 終端文字が入らない）がコピーされ，残りのデータは切り捨てられます。

- 引数 dwSize がデータベースから取得したデータの長さよりも大きい場合

  文字列はすべてコピーします。固定長文字列の場合，残りの領域には NULL 終端文字が埋められます。

- DBR_BINARY 型

  パラメタ値をコピーする領域（LPTSTR 型）を確保し，DBR_BINARY 型変数（構造体）の Data メンバに設定し，この変数を GetField メソッドの引数に指定します。引数 dwSize はこの領域のサイズを指定します。

  引数 dwSize は DBR_BINARY 型の Data メンバの領域のサイズを指定します。引数 dwSize がデータベースから取得したデータの長さ以下の場合，引数 dwSize 分のデータがコピーされ，残りのデータは切り捨てられます。この場合，NULL 終端文字は入りません。

  引数 dwSize がデータベースから取得したデータの長さより大きい場合，データはすべてコピーされ，残りの領域には NULL 終端文字が埋められます。

リターン値を返せるプロシジャを実行した場合

1 番目のパラメタにリターン値が返ります。プロシジャがリターン値を返さない場合はリターン値としてパラメタには 0 が返ります。

データ変換について

データベースのデータ型と GetParam の引数に指定したデータ型とが異なる場合は，値の変換できるものについては引数で指定したデータ型に変換して返します。

文字列から数値データ型への変換に失敗した場合は 0 を返します。データ型の変換規則については，「7.1 クラスライブラリで扱うデータ型と変換規則」の GetField，GetParam メソッドでのデータ型変換規則を参照してください。

NULL 値の扱い

NULL であるパラメタに対して GetParam メソッドを呼び出すと，戻される値は意味のない値(0，空文字列，要素がすべて 0，空文字列からなる構造体，又は不定値)となります。

このため，パラメタの値が NULL である可能性がある場合は，IsNull メソッドを呼び出して，NULL 値かどうかを確認してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しました。

DB_ERROR_CANNOT_BE_NULL
　引数 lpctParamName，又は引数 xxParam に NULL を指定しています。

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数 dwRow，又は引数 dwIndex の範囲が不正です（Execute 実行前に実行した場合を含みます）。

DB_ERROR_DATA_TRUNCATED
　パラメタの値を指定した型に変換できません。
DB_ERROR_PROCEDURE_NOT_COMPLETE
　プロシジャが完了していません。
DB_DRV_ERROR_INVALID_ARGUMENT
　引数に指定したパラメタ名は不正です。

GetParamCount メソッド

機能

IN，IN/OUT パラメタの数を取得します。

形式

```
UINT32 GetParamCount(void)
```

引数

なし

戻り値

データ型：UINT32

パラメタの個数。

機能詳細

SQL 文中に含まれる IN，IN/OUT パラメタの個数を取得します。

このメソッドを呼び出す前に，Execute メソッドを呼び出しておく必要があります。

発生する例外

DB_ERROR_PROCEDURE_NOT_SET
　実行するプロシジャが指定されていません。

GetResultSet メソッド

機能

ストアドプロシジャを実行し，検索したレコードから ResultSet オブジェクトを生成します。

形式

```
DBResultSet *GetResultSet (void) throw DBSQLCA
```

引数

なし

戻り値

データ型：DBResultSet*

DBResultSet オブジェクトへのポインタ。

機能詳細

ストアドプロシジャ内の SELECT 文(副問合わせ,サブクエリーを除く)の検索結果を取得します。

データベースから SetMaxRows メソッドで指定されたレコード数分のレコードを検索後，DBResultSet オブジェクトを生成し，生成したオブジェクトへのポインタを返します。

GetResultSet メソッドを呼び出す前に，Execute メソッドを呼び出します。

BLOB 型データを扱う場合の制限事項については，SetResultSetType メソッド，及び「3.3.4 BLOB 型データの取得方法についての制限」を参照してください。

このメソッドは，非同期実行可能なメソッドです。

このメソッドは，アクセスする DBMS が SQL Anywhere，Adaptive Server Anywhere の場合だけ利用できます。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しました。

DB_ERROR_BEFORE_EXECUTE
　Execute メソッドが実行されていません。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_ERROR_DAB_ACCESS_ERROR
　DABroker でエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR
　システムコールでエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_DRV_ERROR_RDA_PARAM_TYPE
　パラメタのデータ型が設定されていないか，又は設定されたデータ型が不正です。

DB_ERROR_TIMEOUT
　タイムアウトが発生したため制御を戻します。

DB_DRV_ERROR_REQUEST_RESULTSET_ROWS
　ResultSet を生成するために指定されたレコード数が不正です。SetMaxRows メソッドで正しいレコード数を指定してください。

InExecute メソッド

機能

DBCallableStatement オブジェクトに非同期実行中（又は実行待ち）のステートメントがあるかどうかを確認します。

形式

```
BOOLEAN  InExecute(void)
```

引数

なし

戻り値

データ型：BOOLEAN

TRUE：ステートメントが非同期実行中(又は実行待ち)です。

FALSE：非同期実行中(又は実行待ち)ではありません。

機能詳細

InExecute メソッドを実行したオブジェクト内のステートメントが非同期実行中(又は実行待ち）かどうかを確認します。

非同期実行中(又は実行待ち）の場合は TRUE を，非同期実行中(又は実行待ち）でない場合は FALSE を返します。

同期実行接続時は FALSE を返します。

コネクション内のすべてのステートメントを対象に非同期実行中かどうかを確認するには，InWaitForDataSource メソッドを呼び出します(DBConnection クラス)。

発生する例外

なし

IsCompleted メソッド

機能

ストアドプロシジャの実行が完了したかどうかを確認します。

形式

```
BOOLEAN  IsCompleted(void)
```

引数

なし

戻り値

データ型：BOOLEAN

TRUE：ストアドプロシジャの実行は完了しています。

FALSE：ストアドプロシジャは非同期実行中(又は実行待ち)です。

機能詳細

IsCompleted メソッドを実行したオブジェクト内のプロシジャの実行が完了したかどうかを確認します。

完了している場合は TRUE を，非同期実行中(又は実行待ち）の場合は FALSE を返します。同期実行接続時は TRUE を返します。SetProcedure メソッドを呼び出した後，まだ，Execute メソッドを実行していない場合も TRUE を返します。

検索結果を ResultSet で受け取るプロシジャの場合の注意事項

複数の DBResultSet オブジェクトを生成する場合は，すべてのオブジェクトを取得し終えたときに TRUE が返ります。取得できる DBResultSet オブジェクトがある間は FALSE を返します。

発生する例外

なし

IsNull メソッド

機能

IN/OUT，OUT パラメタの値が NULL（欠損値）かどうかを確認します。

形式

インデクス番号で指定する場合

```
BOOLEAN  IsNull(UINT32 dwIndex) throw DBSQLCA
```

パラメタ名で指定する場合

```
BOOLEAN  IsNull(LPCTSTR lpctName) throw DBSQLCA
```

引数

dwIndex

1 から始まる IN/OUT，OUT パラメタの番号を指定します。

lpctName

プロシジャ作成時に指定したパラメタ名を指定します。

戻り値

データ型：BOOLEAN

TRUE：IN/OUT，OUT パラメタの値が NULL です。

FALSE：IN/OUT，OUT パラメタの値は NULL ではありません。

機能詳細

指定された IN/OUT，OUT パラメタが NULL かどうかを確認します。

NULL は C 言語で使用する NULL ポインタの意味ではなく，データベースシステムでは「値がない」ことを意味します。

NULL である IN/OUT，OUT パラメタに対して GetParam メソッドを呼び出すと，戻される値は意味のない値(0，NULL 文字列，要素がすべて 0，NULL 文字列からなる構造体，又は不定値)となります。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_PROCEDURE_NOT_COMPLETE
　プロシジャが完了していません。

DB_ERROR_RESULTSET_NOT_EXIST
　非同期実行時に，検索結果がまだ取得できていません。

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数 dwIndex の範囲が不正です。

DB_ERROR_CANNOT_BE_NULL
　引数 lpctName に NULL を指定しています。

DB_DRV_ERROR_INVALID_ARGUMENT
　引数に指定したパラメタ名，パラメタ番号は不正です。

Parent メソッド

機能

対象となるオブジェクトを生成した DBConnection オブジェクトのポインタを取得します。

形式

```
LPOBJECT Parent(void)
```

引数

なし

戻り値

データ型：LPOBJECT

対象となるオブジェクトを生成した DBConnection オブジェクトのポインタ。

機能詳細

なし

発生する例外

なし

Remove メソッド

機能

DBCallableStatement オブジェクトを削除します。

形式

```
void Remove(void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

DBCallableStatement オブジェクト（自分自身）を削除します。

非同期処理中に実行待ち，又は実行中のステートメントがある場合は削除できません。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中ステートメントがあります。

RemoveResultSet メソッド

機能

DBResultSet オブジェクトを削除します。

形式

```
void RemoveResultSet(void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

GetResultSet メソッドで生成した DBResultSet オブジェクトを削除します。

非同期処理中に実行待ち，又は実行中のステートメントがある場合は削除できません。

このメソッドは，アクセスする DBMS が SQL Anywhere，Adaptive Server Anywhere の場合だけ利用できます。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行中です。

Resume メソッド

機能

ResultSet を返すストアドプロシジャの実行を再開します。

形式

```
void Resume(void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

ストアドプロシジャの検索結果を ResultSet で取得する場合だけ，GetResultSet メソッドを呼び出した後に Resume メソッドを呼び出す必要があります。Resume メソッドを呼び出すことで次の処理を継続できます。

このメソッドは，アクセスする DBMS が SQL Anywhere，Adaptive Server Anywhere の場合だけ利用できます。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_PROCEDURE_NOT_EXECUTE
　プロシジャが実行されていません。

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています(非同期実行時だけ)。

DB_ERROR_IN_ASYNC_EXECUTE
　Execute 又は GetResultSet が非同期実行中です(非同期実行時だけ)。

DB_ERROR_DRIVER_ERROR

DBMS でエラーが発生しました。

DB_ERROR_DAB_ACCESS_ERROR

DABroker でエラーが発生しました。「8.2　C++クラスライブラリのエラー情報」を参照してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR

システムコールでエラーが発生しました。「8.2　C++クラスライブラリのエラー情報」を参照してください。

DB_ERROR_TIMEOUT

タイムアウトが発生したため制御を戻します。

SetMaxFieldSize メソッド

機能

アプリケーションで受け取るフィールドの長さを指定します。

形式

フィールド名で指定する場合

```
void SetMaxFieldSize(LPCTSTR lpctFieldName,
                     UINT32 dwMaxSize=0)    throw DBSQLCA
```

インデクス番号で指定する場合

```
void SetMaxFieldSize(UINT32 dwIndex,
                     UINT32 dwMaxSize=0)    throw DBSQLCA
```

引数

lpctFieldName

フィールド名を指定します。

dwIndex

1 から始まるフィールドの番号を指定します。

dwMaxSize

アプリケーションで受け取るフィールドの長さを，0 以上のバイト数で指定します。この引数で指定された値が，データベースでの定義長よりも大きい場合，引数で指定した値は無視されます。

0 を指定すると，データベース中の定義長が仮定されます。

戻り値

なし

機能詳細

アプリケーションで受け取るフィールドの長さを, バイト数で指定します。アプリケーションで必要とするデータがフィールド値全体でなく, フィールド値の一部である場合に利用します。データベース中のフィールドの定義長ではないことに注意してください。

このメソッドを呼び出す前に，Execute メソッドを呼び出しておく必要があります。

このメソッドは，アクセスする DBMS が SQL Anywhere，Adaptive Server Anywhere の場合だけ利用できます。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数 dwIndex が 1 より小さい値です。

DB_ERROR_CANNOT_BE_NULL
　引数 lpctFieldName に NULL が指定されています。

DB_ERROR_PROCEDURE_NOT_SET
　実行するプロシジャが指定されていません。

DB_DRV_ERROR_INVALID_ARGUMENT
　指定した引数は不正です。

DB_DRV_ERROR_INVALID_MAX_SIZE
　バッファの最大長の指定値が不正です。

SetMaxRows メソッド

機能

ResultSet に検索するレコード数の最大値を指定します。

形式

```
void SetMaxRows(UINT32 dwMaxSize=MAX_ROWS_DEFAULT) throw DBSQLCA
```

引数

dwMaxSize

ResultSet に検索するレコード数の最大値を，1 以上 4096 以下の値で指定します。

システムデフォルト値は，MAX_ROWS_DEFAULT です（=100 が仮定されます）。

戻り値

なし

機能詳細

DBResultSet オブジェクトで一度に取得できる最大レコード数を指定します。

一度の読み込み（GetResultSet メソッド，又は PageNext メソッドの呼び出し）で，データベースから取得するレコード数の最大値を設定します。

1 レコードのフィールド数は，DBResultSetMetaData オブジェクトの GetColumnCount メソッドを呼び出して取得できます。

このメソッドは，アクセスする DBMS が SQL Anywhere，Adaptive Server Anywhere の場合だけ利用できます。

更新可能な DBResultSet オブジェクトでの扱い

DBResultSet オブジェクトが更新可能なオブジェクトとして生成された場合(SetResultSetType メソッドの引数 swType で TYPE_EXCLUSIVE を指定した場合)，SetMaxRows メソッドによる指定は無視され，常に 1 が仮定されます。更新可能な DBResultSet は常に 1 レコードだけを読み込みます。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数 dwMaxSize に範囲外（1 より小さい）が指定されています。

SetNull メソッド

機能

指定したパラメタの値に NULL（欠損値)を設定します。

形式

```
void SetNull(UINT32 dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まる IN,IN/OUT パラメタの番号を指定します。

戻り値

なし

機能詳細

指定されたパラメタに NULL を設定します。

NULL は C 言語で使用する NULL ポインタの意味ではなく，データベースシステムでは「値がない」ことを意味します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_FOUND
　引数 dwIndex の範囲が不正です。

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

SetParam メソッド

機能

IN/OUT，IN パラメタの値を設定します。

形式

```
void SetParam(UINT32 dwIndex, LPCTSTR lpctParam) throw DBSQLCA
void SetParam(UINT32 dwIndex, INT16 sParam) throw DBSQLCA
void SetParam(UINT32 dwIndex, INT32 lParam) throw DBSQLCA
void SetParam(UINT32 dwIndex, UINT16 swParam) throw DBSQLCA
void SetParam(UINT32 dwIndex, UINT32 dwParam) throw DBSQLCA
void SetParam(UINT32 dwIndex, SINGLE sfParam) throw DBSQLCA
void SetParam(UINT32 dwIndex, DOUBLE dfParam) throw DBSQLCA
void SetParam(UINT32 dwIndex, const DBR_DATETIME& dtParam)
                                              throw DBSQLCA
void SetParam(UINT32 dwIndex, const DBR_BINARY& blobParam)
                                              throw DBSQLCA
```

引数

指定されたパラメタを，データベースの型にキャストして設定します。

dwIndex

1 から始まるパラメタの番号を指定します。指定するパラメタの番号は，SQL 文中での出現順に割り当てられます。

xxParam（データ型：形式の第 2 引数を参照してください）

パラメタに設定する値を指定します。

戻り値

なし

機能詳細

指定されたパラメタの値を設定します。プロシジャの定義で IN/OUT，IN で宣言したパラメタのうち，SetProcedure メソッドの引数 lpctProcedure で指定した?パラメタに値を設定します。

値は指定するデータ型からデータベース固有の型に変換されます。

SetParam メソッドは SetProcedure メソッドを実行した後に実行してください。

引数 lpctParam が NULL（LPCTSTR 型の変数に NULL を代入して引数に渡した場合，又は NULL を LPCTSTR 型にキャストした場合）の時も欠損値が設定されます。キャストしないで引数に NULL だけを指定した場合は欠損値とはみなしません。

パラメタに欠損値を設定したい場合は，SetNull メソッドを使用してください。

DBR_BINARY 型を指定する場合

DBR_BINARY 型の各メンバには次の値を設定します。

Length：設定するデータの長さ（単位はバイト）

RealLength：設定不要

Data：実際のデータの領域を指すポインタ

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数 dwIndex の範囲が不正です。

DB_ERROR_IN_ASYNC_EXECUTE
　ステートメントが非同期実行処理中です。

DB_ERROR_PROCEDURE_NOT_SET
　実行するプロシジャが指定されていません。

SetProcedure メソッド

機能

引数で指定したストアドプロシジャ名をデータベースに通知します。

形式

```
void SetProcedure(LPCTSTR lpctProcedure,
                  UINT16 swWait = LOCK_OPT_DEFAULT)    throw DBSQLCA
```

引数

lpctProcedure

実行するプロシジャ名を文字列で指定します。

- ORACLE の場合
  OCI プログラムでプロシジャを実行する際に指定する PL/SQL ブロック内の記述を指定します。ただし，プレースホルダは?に置き換えて指定してください。
  PL/SQL については DBMS のマニュアルを参照してください。
  OCI プログラムの指定と DABroker for C++の指定方法の比較を次に示します。
  実行するプロシジャ名は GET_SALARY(in,out)とします。
  OCI プログラム

```
BEGIN
  GET_SALARY(:id, :salary);
END;
```

DABroker for C++プログラム

```
SetProcedure("GET_SALARY(?,?)");
```

- HiRDB の場合

  操作系 SQL の CALL 文に記述する内容を指定します。プロシジャの引数には?パラメタだけが指定できます。

  CALL 文の仕様については DBMS のマニュアルを参照してください。

  HiRDB の埋め込み SQL の指定と DABroker for C++の指定方法の比較を次に示します。

  実行するプロシジャ名は GET_SALARY(in,out)とします。

  埋め込み SQL

  ```
  CALL GET_SALARY(?,?)
  ```

  DABroker for C++プログラム

  ```
  SetProcedure("GET_SALARY(?,?)");
  ```

- SQL Anywhere の場合

  CALL 文に記述する内容を指定します。動的に設定するプロシジャの引数には?パラメタだけが指定できます。

  RETURN 文を含むプロシジャや，ResultSet を返すプロシジャの場合も，SetProcedure メソッドの引数に指定する内容には特に変更ありません。

  RETURN 文で返す値は GetParam メソッドで取得します。

  CALL 文の仕様については DBMS のマニュアルを参照してください。

  CALL 文の指定と DABroker for C++の指定方法の比較を次に示します。

  実行するプロシジャ名は GET_SALARY(in,out)とします。

  CALL 文の指定

  ```
  CALL GET_SALARY(id,salary)
  ```

  DABroker for C++プログラム

  ```
  SetProcedure("GET_SALARY(?,?)");
  ```

  CALL 文の指定(RETURN 文を含むプロシジャの場合)

  ```
  RETVALUE = CALL GET_SALARY_RET(id,salary)
  ```

  DABroker for C++プログラム

  ```
  SetProcedure("GET_SALARY(?,?)");
  ```

- Adaptive Server Anywhere，及び SQL Server の場合

  ODBC でプロシジャを実行する際に記述する内容を指定します。

  動的に設定するプロシジャの引数には?パラメタだけが指定できます。

  RETURN 値は GetParam メソッドで取得します。

  ODBC でのプロシジャの実行方法については DBMS のマニュアルを参照してください。

  ODBC の指定と DABroker for C++の指定方法の比較を次に示します。

  実行するプロシジャ名は GET_SALARY(in,out)とします。

  ODBC の指定

  ```
  {?=CALL GET_SALARY(id,salary)}
  ```

  DABroker for C++プログラム

  ```
  SetProcedure("GET_SALARY(?,?)");
  ```

swWait

実行する SQL 単位に，処理対象のレコードがほかのトランザクションによってロックされている場合の処理を指定します。次の値のうち，どれか一つを指定します。この引数は，使用する DBMS が SQL Anywhere，Adaptive Server Anywhere の場合だけ指定できます。

- LOCK_OPT_DEFAULT
  DBConnection オブジェクトの設定を引き継ぎます。（DBDriver クラスの Connect メソッドの引数 swWait に指定した値）
- LOCK_OPT LOCK_OPT_NOWAIT，又は LOCK_OPT_WITH_ROLLBACK
  ロックの解除を待たないで，すぐにエラーを返します。HiRDB 又は XDM/RD の場合は同時にロールバックを実行します。それ以外の DBMS の場合は，ロールバックは実行しません。
- LOCK_OPT_WITHOUT_ROLLBACK
  ロックの解除を待たないで，すぐにエラーを返します。ロールバックは実行しません。
  ただし，現在のバージョンでは，XDM/RD ではロールバックを実行します。また，HiRDB の DELETE,INSERT,UPDATE 文では，このオプションは無効で，LOCK_OPY_WAIT と同じ動作になります。
- LOCK_OPT_WAIT
  ロックが解除されるまで待ちます。

戻り値

なし

機能詳細

引数で指定したストアドプロシジャ名をデータベースに通知します。

使用する DBMS ごとのパラメタの扱いを次に示します。

ORACLE の場合

- ストアドプロシジャが実行できます。
- IN，IN/OUT パラメタで動的に設定するパラメタ，又は OUT パラメタには「?」を指定します。
- 指定できるパラメタ数の最大値は 64 個です。

HiRDB の場合

- ストアドプロシジャが実行できます。
- IN，IN/OUT，OUT パラメタで動的に設定するパラメタかどうかにかかわらず，「?」を指定します。
- 指定できるパラメタ数は，データベースの仕様に従います。

SQLAnywhere，Adaptive Server Anywhere の場合

- ストアドプロシジャ及びファンクションが実行できます。
- IN，IN/OUT パラメタで動的に設定するパラメタには，「?」を指定します。OUT パラメタにも，?を指定します。
- 指定できるパラメタ数は，データベースの仕様に従います。

SQL Server の場合

- ストアドプロシジャが実行できます。
- IN，IN/OUT パラメタで動的に設定するパラメタには，「?」を指定します。OUT パラメタにも，?を指定します。
- 指定できるパラメタ数は，データベースの仕様に従います。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行中です。

DB_ERROR_PROCEDURE_NOT_COMPLETE
　プロシジャが終了していません。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_ERROR_DAB_ACCESS_ERROR
　DABroker でエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR
　システムコールでエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_TIMEOUT
　タイムアウトが発生したため制御を戻します。

SetResultSetType メソッド

機能

DBResultSet オブジェクトの生成オプションを指定します。

DBResultSet オブジェクトを生成するときに，参照専用，又は更新可能のどちらで生成するか選択します。検索レコードに対するロック方法を指定します。

検索時にバッファを幾つ使うかも指定できます。

また，BLOB 型データを扱う場合，その取得方法についても指定します。

形式

```
void SetResultSetType (UINT16 swType = TYPE_EXCLUSIVE|BUFFER_TYPE_SINGLE,
                       DBR_BLOB_TYPE nBLOBType = TYPE_BLOB_MEMORY,
                       LPCTSTR lpctBLOBFileName = NULL)
                                              throw DBSQLCA
```

引数

swType

次の三つの値を指定します。

- 検索したレコードに対するロック方法
- 検索時に使用するバッファ数
- VARCHAR データを DBR_BINARY 型で取得した場合の Length メンバ，RealLength メンバの意味

指定する値は，上記三つの値の論理和になります。

このメソッドを実行しなかった場合の動作は，TYPE_EXCLUSIVE | BUFFER_TYPE_SINGLE | VARCHAR_LENGTH_DEF を指定した場合と同じになります。

検索したレコードに対するロック方法

次に示す TYPE_EXCLUSIVE，TYPE_EXCLUSIVE2，TYPE_NONE，TYPE_WAIT，TYPE_NOWAIT，TYPE_SHARED のどれかの値で，検索したレコードに対するロック方法を指定します。

- TYPE_EXCLUSIVE 又は TYPE_EXCLUSIVE2

  DBResultSet オブジェクトからレコードの追加，更新，削除をする場合は，このオプションを指定します。検索したレコードに排他ロックが掛かり，他のトランザクションからの参照・更新を制限できます。実際に排他ロックが掛かるのは，GetResultSet メソッドを実行した時点からです。

  このオプションで生成した DBResultSet オブジェクトでは，一度に複数のレコードを読み込めないため，SetMaxRows で指定したレコード数は無視されます。また，検索条件に一致する次のレコードをアクセスするには，Next メソッドを使用します。TYPE_EXCLUSIVE，又は TYPE_EXCLUSIVE2 で生成したオブジェクトの場合，カーソル制御のメソッドとしては Next メソッドだけが指定できます。Next メソッドについては，「5.7 DBResultSet クラスの詳細」を参照してください。

  なお，この引数の値は，実行時に SQL 文のオプション文字列に変換されて実行されます。DBMS 別に，どのオプション文字列へ変換されて実行されるのかを次に示します。なお，HiRDB 以外の DBMS では TYPE_EXCLUSIVE と TYPE_EXCLUSIVE2 で同じオプション文字列へ変換されて実行されます。

  - ORACLE：FOR UPDATE
  - SQL Anywhere,Adaptive Server Anywhere：オプション文字列は付加しません。
  - HiRDB

    TYPE_EXCLUSIVE：FOR UPDATE

    TYPE_EXCLUSIVE2：WITH EXCLUSIVE LOCK FOR UPDATE
  - XDM/RD：WITH EXCLUSIVE LOCK FOR UPDATE
  - SQL Server：UPDLOCK
  - SQL/K：FOR UPDATE

    詳細については，マニュアル「SQL/K」を参照してください。
  - XDM/SD：LOCK SU
- TYPE_NONE

  参照専用の DBResultSet オブジェクトを生成します。検索するレコードに対して，参照ロックを掛けます。

  このオプションで生成した DBResultSet オブジェクトでは，複数レコードを読み込めます。

  なお，この引数の値は，実行時に SQL 文のオプション文字列に変換されて実行されます。DBMS 別に，どのオプション文字列へ変換されて実行されるのかを次に示します。

  - ORACLE：付加しません。
  - SQL Anywhere，Adaptive Server Anywhere：付加しません。
  - HiRDB：付加しません。

- XDM/RD：付加しません。
- SQL Server：付加しません。
- SQL/K：付加しません。
- XDM/SD：付加しません(SDEXCLUSIVE 値が仮定されます)。
  SDEXCLUSIVE 値の詳細については，マニュアル「Database Connection Server」のコントロール空間起動制御文を参照してください。

- TYPE_WAIT(HiRDB，XDM/RD，SQL Server，SQL/K，XDM/SD の場合に有効です)
  参照専用の DBResultSet オブジェクトを生成します。
  検索したレコードに対して参照ロックを掛けますが，見終わった行から排他制御を解除します。このため，DABroker for C++では，ResultSet にレコードをすべて読み込んだ時点，つまり，GetResultSet メソッド，又は PageNext メソッドが完了した時点で，ロックが解除されています。
  このオプションで生成した DBResultSet オブジェクトでは，複数レコードを読み込めます。
  なお，この引数の値は，実行時に SQL 文のオプション文字列に変換されて実行されます。DBMS 別に，どのオプション文字列へ変換されて実行されるのかを次に示します。
  - HiRDB：WITHOUT LOCK WAIT
  - XDM/RD：WITHOUT LOCK WAIT
  - SQL Server：HOLDLOCK
  - SQL/K：付加しません。
  - XDM/SD：LOCK SR
- TYPE_NOWAIT(HiRDB，XDM/RD，SQL Server，SQL/K，XDM/SD の場合に有効です)
  参照専用の DBResultSet オブジェクトを生成します。該当するレコードをほかのトランザクションが更新中だったり，TYPE_EXCLUSIVE で検索している場合でも読み込めます。排他制御を全くしません。
  このオプションで生成した DBResultSet オブジェクトでは，複数レコードを読み込めます。
  なお，この引数の値は，実行時に SQL 文のオプション文字列に変換されて実行されます。DBMS 別に，どのオプション文字列へ変換されて実行されるのかを次に示します。
  - HiRDB：WITHOUT LOCK NOWAIT
  - XDM/RD：WITHOUT LOCK NOWAIT
  - SQL Server：NOLOCK
  - SQL/K：付加しません。
  - XDM/SD：LOCK NR
- TYPE_SHARED(HiRDB，XDM/RD，SQL/K，XDM/SD の場合だけ有効です)
  検索するレコードに共有ロックを掛けます。共有ロックの掛かったレコードは，ほかのトランザクションから参照できますが，更新はできません。
  - HiRDB：WITH SHARE LOCK
  - XDM/RD：WITH SHARE LOCK
  - SQL/K：付加しません。
  - XDM/SD：LOCK SR

検索時に使用するバッファ数

次に示す BUFFER_TYPE_SINGLE，又は BUFFER_TYPE_DOUBLE のどちらかの値で，検索時に使用するバッファ数を指定します。指定を省略した場合は，BUFFER_TYPE_SINGLE が仮定されます。なお，バッファの利用方法については，「3.3.3 検索性能の向上策」を参照してください。

- BUFFER_TYPE_SINGLE
  検索結果を保持するバッファを一つだけ使います。
- BUFFER_TYPE_DOUBLE
  検索結果を保持するバッファを二つ使います。

VARCHAR データを DBR_BINARY 型で取得したときの Length メンバ，RealLength メンバの意味

次に示す値のどれか一つを指定します。

- VARCHAR_LENGTH_DEF
  - GetField メソッドの形式 2 を使用した場合，又は形式 3 を使用してもデータの切り捨てが発生しなかった場合
    Length：カラムの定義長※
    RealLength：0
  - GetField メソッドの形式 3 を使用してデータの切り捨てが発生した場合
    Length：切り捨てたデータ長
    RealLength：カラムの定義長※
    注※　SetMaxFieldSize メソッドで取得データ長を指定している場合は，「カラムの定義長」ではなく「SetMaxFieldSize メソッドで指定したサイズ」となります。
- VARCHAR_LENGTH_REAL
  - GetField メソッドの形式 2 を使用した場合，又は形式 3 を使用してもデータの切り捨てが発生しなかった場合
    Length：実際のデータ長※
    RealLength：0
  - GetField メソッドの形式 3 を使用してデータの切り捨てが発生した場合
    Length：切り捨てたデータ長
    RealLength：実際のデータ長※
    注※　SetMaxFieldSize メソッドでカラムの定義長よりも小さいサイズを指定している場合は，データベースから取得したデータは実際のデータよりも小さい場合があるため，「実際のデータ長」ではなく「取得データ長」となります。

nBLOBType

BLOB 型データを取得するときは次の方法を使います。

- TYPE_BLOB_MEMORY
  メモリ上に一括して取得します。
  ResultSet のレコード数が 1 の場合，BLOB 型データをすべて取得します。
  ResultSet のレコード数が複数の場合，先頭から 32K バイトまでを取得します。

戻り値

なし

機能詳細

検索レコードに対するロック方法，検索時にバッファを幾つ使うか，及び BLOB 型データの取得方法について指定します。

指定した各引数の値は，次に GetResultSet メソッドを実行したときから有効になります。

このメソッドは，アクセスする DBMS が SQL Anywhere，Adaptive Server Anywhere の場合だけ利用できます。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

WaitForDataSource メソッド

機能

DBCallableStatement オブジェクトで要求した実行待ち，及び実行中の非同期処理が終了するまで待ちます。非同期処理中に同期を取りたい場合に利用できます。

形式

```
BOOLEAN WaitForDataSource(UINT16 swWaitTime = DBR_INFINITE)
```

引数

swWaitTime

非同期処理の終了を待つための最大時間（単位：ミリ秒），又は DBR_INFINITE を指定します。

指定した時間内に非同期処理が終了した場合，TRUE が返ります。指定した時間を経過しても非同期処理が終了しない場合，FALSE が返ります。非同期実行中の SQL がない場合は，TRUE が返ります。

DBR_INFINITE を指定した場合はタイムアウト時間を設定しません。非同期処理が終了するまで待ち続けます。

同期実行時はすぐに TRUE を返します。

戻り値

データ型：BOOLEAN

TRUE：すべての非同期処理が終了しました。

FALSE：タイムアウト時間が経過しました。

機能詳細

DBCallableStatement オブジェクトで要求した実行待ち，及び実行中の非同期処理(SQL)が，終了するのを待ちます。

DBConnection クラスの WaitForDataSource メソッドとの違い

DBCallableStatement オブジェクトの WaitForDataSource メソッドでは，WaitForDataSource メソッドを実行した DBCallableStatement オブジェクトの非同期実行処理が終了するまで待ちます。ほかのオブジェクトの非同期実行処理については待ちません。そのため, ほかのオブジェクトの非同期実行処理が実行中でも，WaitForDataSource メソッドを呼び出したオブジェクトの非同期実行処理が終了すれば, WaitForDataSource メソッドは TRUE を返します。

同じ DBConnection オブジェクトで実行しているすべての非同期実行処理が終了するのを待ちたい場合は，DBConnection クラスの WaitForDataSource メソッドを実行してください。

発生する例外

なし

5.11　DBDatabaseMetaData クラスの詳細

データベース情報を取得します。このクラスは，次に示すようなデータベース情報を取得するメソッドを提供するクラスです。

- フィールド一覧
- プライマリキー一覧
- ストアドプロシジャの一覧
- ストアドプロシジャのパラメタ一覧
- テーブル一覧

なお，使用する DBMS によっては，取得できない一覧情報があります。このような情報を取得しようとした場合，0 件のオブジェクトが返ります。次に，DBMS ごとの取得できない一覧を示します。

HiRDB の場合

プライマリキー一覧（GetPrimaryKeys メソッド）

XDM/RD の場合

ストアドプロシジャ一覧（GetProcedures メソッド）

ストアドプロシジャのパラメタ一覧（GetProcedureColumns メソッド）

プライマリキー一覧（GetPrimaryKeys メソッド）

機 能	メソッド名
フィールド名一覧を取得します。	GetColumns
SQL 文の非同期実行時のエラー情報を得るために，DBSQLCA オブジェクトのポインタ取得します。	GetErrorStatus
プライマリキー一覧を取得します。	GetPrimaryKeys
ストアドプロシジャの一覧を取得します。	GetProcedures
ストアドプロシジャのパラメタ一覧を取得します。	GetProcedureColumns
テーブル一覧を取得します。	GetTables
DBDatabaseMetaData オブジェクトに非同期実行中（又は実行待ち）のステートメントがあるかどうかを確認します。	InExecute
対象となる DBDatabaseMetaData オブジェクトを生成した DBConnection オブジェクトのポインタを取得します。	Parent

GetColumns メソッド

機能

フィールド名一覧を取得するために，classListColumns オブジェクトへのポインタを取得します。

形式

```
classListColumns *GetColumns(LPCTSTR lpctOwner,
                             LPCTSTR lpctTableName,
                             UINT32 dwMaxSize=0,
                             LPCTSTR lpctCondition = NULL,
```

```
                                LPCTSTR lpctEscChar = NULL)
                                                throw DBSQLCA
```

引数

lpctOwner

テーブルの所有者名を指定します。

lpctTableName

テーブル名を指定します。

dwMaxSize

取得するフィールド数の最大値を指定します。

0 を指定した場合は，対象のすべてのフィールドを検索します。

lpctCondition

一覧に含むフィールドのフィールド名をワイルドカードを使用して指定します。この指定は, SQL の LIKE 演算子に指定する文字列の形式を使用してください。

XDM/SD の場合，ワイルドカードの指定は無効です。ワイルドカードを指定しても，テーブルに含まれるすべてのフィールドが取得されます。

lpctEscChar

ワイルドカード文字列中に含むエスケープ文字を 1 文字指定します。2 文字以上指定した場合は先頭の文字を使用します。

次のデータベースの場合，この引数は指定できません。

- SQL Anywhere
- Adaptive Server Anywhere
- SQL Server
- SQL/K
- XDM/SD

戻り値

データ型：classListColumns*

classListColumns オブジェクトへのポインタ。

機能詳細

フィールド名の一覧を取得するために，classListColumns オブジェクトへのポインタを取得します。

フィールドを定義した順番をキーにして，昇順にソートして取得します。ただし，次のデータベースの場合，順序は DBMS に依存するため，ソートされません。

- SQL Anywhere
- Adaptive Server Anywhere

- SQL/K
- XDM/SD

このメソッドは，非同期実行可能なメソッドです。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_CANNOT_BE_NULL
　所有者名又はテーブル名に NULL を指定しています。

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　最大件数の指定が範囲内ではありません。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しています。

DB_ERROR_DAB_ACCESS_ERROR
　DABroker でエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR
　システムコールでエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_ILLEGAL_VALUE
　指定した引数が不正です。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_TIMEOUT
　タイムアウトが発生したため制御を戻します。

GetErrorStatus メソッド

機能

SQL 文の非同期実行時のエラー情報を得るために，DBSQLCA オブジェクトのポインタ取得します。

形式

```
DBSQLCA *GetErrorStatus(void)
```

引数

なし

戻り値

データ型：DBSQLCA *

DBSQLCA オブジェクトへのポインタ。

機能詳細

DBSQLCA オブジェクトへのポインタを取得します。

非同期処理のときは，エラーが発生しても直にエラーを取得できません。このため，ユーザは任意の時点でエラーを確認する必要があります。

非同期実行時にエラーが発生した場合，DBSQLCA オブジェクトにエラー情報が設定されます。

DBSQLCA オブジェクトでは，非同期実行中に発生したエラー情報を最大 100 回分保存できます。エラー情報は，新しいエラー情報から 100 回分保存されるため，エラー情報を確認したあとは DBSQLCA クラスの Delete メソッドを呼び出して，不要なエラー情報をクリアしておいてください。

発生する例外

なし

GetPrimaryKeys メソッド

機能

プライマリキー一覧を取得するために，classListPrimaryKeys オブジェクトへのポインタを取得します。

形式

```
classListPrimaryKeys *GetPrimaryKeys(LPCTSTR lpctOwner,
                                     LPCTSTR lpctTableName,
                                     LPCTSTR lpctCondition = NULL,
                                     LPCTSTR lpctEscChar = NULL)
                                                    throw DBSQLCA
```

引数

lpctOwner

テーブルの所有者名を指定します。

lpctTableName

テーブル名を指定します。

lpctCondition

一覧に含むプライマリキーのフィールド名をワイルドカードで指定します。この指定には，SQL の LIKE 演算子に指定する文字列の形式を使用してください。SQL Anywhere,Adaptive Server Anywhere,SQL Server の場合，この引数は指定できません。

lpctEscChar

ワイルドカード文字列中に含むエスケープ文字を 1 文字指定します。2 文字以上を指定した場合は先頭の文字を使用します。SQL Anywhere,Adaptive Server Anywhere,SQL Server の場合，この引数は指定できません。

戻り値

データ型：classListPrimaryKeys*

classListPrimaryKeys オブジェクトへのポインタ。

機能詳細

指定したテーブルのプライマリキーを構成するフィールドの一覧情報を持つ classListPrimaryKeys オブジェクトへのポインタを取得します。

プライマリキーの優先順位をキーにして，昇順にソートして取得します。ただし，SQL Anywhere，Adaptive Server Anywhere の場合は，順序は DBMS に依存するため，ソートされません。

使用している DBMS が HiRDB，又は XDM/RD の場合，プライマリキーの一覧は取得できません。

このメソッドは，非同期実行可能なメソッドです。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_CANNOT_BE_NULL
　所有者名，又はテーブル名に NULL を指定しています。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_ERROR_DAB_ACCESS_ERROR
　DABroker でエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR
　システムコールでエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_ILLEGAL_VALUE
　指定した引数が不正です。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_TIMEOUT
　タイムアウトが発生したため制御を戻します。

GetProcedures メソッド

機能

ストアドプロシジャの一覧を取得するために，classListProcedures オブジェクトへのポインタを取得します。

形式

```
classListProcedures *GetProcedures(LPCTSTR lpctOwner = NULL,
                                   UINT32 dwMaxSize=0,
                                   LPCTSTR lpctCondition = NULL,
                                   LPCTSTR lpctEscChar = NULL)
                                                  throw DBSQLCA
```

引数

lpctOwner

プロシジャの所有者名，又は NULL を指定します。

NULL を指定した場合，参照できるすべてのストアドプロシジャを検索します。

dwMaxSize

取得するプロシジャの数の最大値を指定します。

0 を指定した場合は，対象になるすべてのプロシジャを検索します。

lpctCondition

一覧に含むプロシジャのプロシジャ名を，ワイルドカードを使用して指定します。この指定には，SQL の LIKE 演算子に指定する文字列の形式を使用してください。

lpctEscChar

ワイルドカード文字列中に含むエスケープ文字を 1 文字指定します。2 文字以上を指定した場合は先頭の文字を使用します。SQL Anywhere,Adaptive Server Anywhere,SQL Server の場合，この引数は指定できません。

戻り値

データ型：classListProcedures*

classListProcedures オブジェクトへのポインタ。

機能詳細

ストアドプロシジャの一覧を取得するために，classListProcedures オブジェクトへのポインタを取得します。

SQL Anywhere では，ストアドプロシジャとファンクションの一覧を取得できます。

ストアドプロシジャ名をキーにして，昇順にソートして取得します。ただし，SQL Anywhere,Adaptive Server Anywhere の場合は，順序は DBMS に依存するため，ソートされません。

このメソッドは，非同期実行可能なメソッドです。

XDM/RD では取得できません。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　最大件数の指定が範囲内ではありません。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_ERROR_DAB_ACCESS_ERROR
　DABroker でエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR
　システムコールでエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_ILLEGAL_VALUE
　指定した引数が不正です。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_TIMEOUT
　タイムアウトが発生したため制御を戻します。

GetProcedureColumns メソッド

機能

ストアドプロシジャのパラメタ一覧を取得するために，classListProcedureColumns オブジェクトへのポインタを取得します。

形式

```
classListProcedureColumns *GetProcedureColumns
                                (LPCTSTR lpctOwner,
                                 LPCTSTR lpctProcedureName,
                                 UINT32 dwMaxSize=0,
                                 LPCTSTR lpctCondition = NULL,
                                 LPCTSTR lpctEscChar = NULL)
                                                throw DBSQLCA
```

引数

lpctOwner

プロシジャの所有者名を指定します。

lpctProcedureName

プロシジャ名を指定します。

dwMaxSize

取得するパラメタの数の最大値を指定します。

0 を指定した場合は，対象になるすべてのパラメタを検索します。

lpctCondition

一覧に含むプロシジャのパラメタ名を，ワイルドカードを使用して指定します。この指定には，SQL の LIKE 演算子に指定する文字列の形式を使用してください。

lpctEscChar

ワイルドカード文字列中に含むエスケープ文字を 1 文字指定します。2 文字以上を指定した場合は先頭の文字を使用します。SQL Anywhere,Adaptive Server Anywhere,SQL Server の場合，この引数は指定できません。

戻り値

データ型：classListProcedureColumns*

classListProcedureColumns オブジェクトへのポインタ。

機能詳細

ストアドプロシジャのパラメタ一覧を取得するために，classListProcedureColumns オブジェクトへのポインタを取得します。

ストアドプロシジャで定義した順番をキーにして，昇順にソートして取得します。ただし，SQL Anywhere，Adaptive Server Anywhere の場合は，順序は DBMS に依存するため，ソートされません。

このメソッドは，非同期実行可能なメソッドです。

XDM/RD の場合
　ストアドプロシジャのパラメタ一覧は取得できません。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_CANNOT_BE_NULL
　所有者名，又はプロシジャ名に NULL を指定しています。

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　最大件数の指定が範囲内ではありません。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_ERROR_DAB_ACCESS_ERROR
　DABroker でエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR

システムコールでエラーが発生しました。

DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_ILLEGAL_VALUE

指定した引数が不正です。

DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_TIMEOUT

タイムアウトが発生したため制御を戻します。

GetTables メソッド

機能

テーブル一覧を取得するために，classListTables オブジェクトへのポインタを取得します。

形式

```
classListTables *GetTables(LPCTSTR lpctOwner=NULL,
                           UINT16 swType = TBL_TYPE_TABLE
                                         | TBL_TYPE_VIEW
                                         | TBL_TYPE_SYNONYM
                                         | TBL_TYPE_SYSTEM ,
                           UINT32 dwMaxSize=0,
                           LPCTSTR lpctCondition = NULL,
                           LPCTSTR lpctEscChar = NULL)
                                              throw DBSQLCA
```

引数

lpctOwner

テーブルの所有者名，又は NULL を指定します。

NULL を指定した場合は，参照できるすべてのテーブルを検索します。

XDM/SD の場合，指定できる所有者名は常に 8 バイトです。8 バイト未満の所有者名を指定すると，後ろに空白が付加されます。

swType

検索するテーブルの種別を指定します。

- TBL_TYPE_TABLE：実テーブル（システム所有のテーブルを除く）

  XDM/SD を使用しているときは，TBL_TYPE_TABLE だけが指定できます。TBL_TYPE_TABLE を含まない指定をすると，エラーをスローします。ほかの種別と一緒に指定しても，TBL_TYPE_TABLE だけが有効になります。
- TBL_TYPE_VIEW：ビュー（システム所有のビューを除く）
- TBL_TYPE_SYNONYM：シノニム（システム所有のシノニムを除く）

  シノニムの一覧は Oracle7 を使用しているときだけ取得できます。SQL Anywhere，HiRDB，XDM/RD を使用しているときに TBL_TYPE_SYNONYM だけを指定するとエラーをスローします。ほかの種別と一緒に指定した場合は無視します。
- TBL_TYPE_SYSTEM：システム所有のオブジェクト

TBL_TYPE_SYSTEM を指定した場合は，引数 lpctOwner の指定がシステムのものでなくてもシステムディクショナリを取得します。

HiRDB を使用している場合，TBL_TYPE_SYSTEM を指定しても所有者が"HiRDB"の実表については取得できません。

SQL/K を使用している場合，TBL_TYPE_SYSTEM を指定しても一覧は取得できません。システム所有のオブジェクト件数は 0 件になります。

dwMaxSize

取得するテーブル数の最大値を指定します。

0 を指定した場合は，対象になるすべてのテーブルを検索します。

lpctCondition

一覧に含むテーブルのテーブル名をワイルドカードを使って指定します。この指定には，SQL の LIKE 演算子に指定する文字列の形式を使ってください。

lpctEscChar

ワイルドカード文字列中に含むエスケープ文字を 1 文字指定します。2 文字以上を指定した場合は先頭の文字を使用します。

次のデータベースの場合，この引数は指定できません。

- SQL Anywhere
- Adaptive Server Anywhere
- SQL Server
- SQL/K
- XDM/SD

戻り値

データ型：classListTables*

classListTables オブジェクトへのポインタ。

機能詳細

テーブル一覧を取得するために，classListTables オブジェクトへのポインタを取得します。

所有者名，テーブル名の順で，それぞれを昇順にソートして取得します。ただし，次のデータベースの場合，順序は DBMS に依存するため，ソートされません。

- SQL Anywhere
- Adaptive Server Anywhere
- SQL/K
- XDM/SD

Adaptive Server Anywhere の場合は，順序は DBMS に依存するため，ソートされません。

このメソッドは，非同期実行可能なメソッドです。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　最大件数の指定が範囲内ではありません。

DB_DRV_ERROR_INVALID_TABLELIST_TYPE
　実テーブル(システム所有のテーブルを除く)以外の一覧取得はサポートしていません（XDM/SD の場合)。

DB_DRV_ERROR_NOT_SUPPORT_TABLE_LIST
　シノニムの一覧表示はサポートしていません（HiRDB，XDM/RD，SQL/K，XDM/SD の場合)。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_ERROR_DAB_ACCESS_ERROR
　DABroker でエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR
　システムコールでエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_ILLEGAL_VALUE
　指定した引数が不正です。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_TIMEOUT
　タイムアウトが発生したため制御を戻します。

InExecute メソッド

機能

DBDatabaseMetaData オブジェクトに非同期実行中（又は実行待ち）のステートメントがあるかどうかを確認します。

形式

```
BOOLEAN  InExecute(void)
```

引数

なし

戻り値

データ型：BOOLEAN

TRUE：一覧取得処理が非同期実行中(又は実行待ち)です。

FALSE：非同期実行中(又は実行待ち)ではありません。

機能詳細

InExecute メソッドを実行したオブジェクト内のステートメントが非同期実行中(又は実行待ち）かどうかを確認します。

非同期実行中(又は実行待ち）の場合は TRUE を，非同期実行中(又は実行待ち）でない場合は FALSE を返します。

同期実行接続時は FALSE を返します。

コネクション内のすべてのステートメントを対象に非同期実行中かどうかを確認するには，InWaitForDataSource メソッドを呼び出します(DBConnection クラス)。

発生する例外

なし

Parent メソッド

機能

対象となる DBDatabaseMetaData オブジェクトを生成した DBConnection オブジェクトのポインタを取得します。

形式

```
LPOBJECT Parent(void)
```

引数

なし

戻り値

データ型：LPOBJECT

対象となる DBDatabaseMetaData オブジェクトを生成した DBConnection オブジェクトのポインタ。

機能詳細

なし

発生する例外

なし

5.12　DBTransaction クラスの詳細

トランザクションを管理するクラスです。

トランザクションを制御する，BeginTrans,Commit,Rollback などのメソッドや，トランザクションの処理中かどうかを確認するメソッドなどを提供します。

更新処理をするアプリケーションの作成では, できるだけデータベースアクセスにトランザクションを使用してください。もし，トランザクション管理をしなかった場合，コミットは次のように実行されます。

データベースを更新する SQL 文を一文実行するごとにコミットされます。これを，自動コミットといいます。自動コミットをするかしないかは，SetAutoCommit メソッドで指定します。

自動コミットは，SELECT 文以外の SQL を実行した直後にコミットします。

具体的には，DBConnection クラスの ExecuteDirect メソッド，DBStatement クラスの Execute メソッド，DBPreparedStatement クラスの ExecuteUpdate メソッドで SQL 文を実行した後，又は DBResultSet クラスの Update，Delete メソッドを実行した後にコミットします。

ただし，自動コミットを設定すると，更新処理では，そのたびに更新されてしまい，ロールバックなどが有効になりません。アプリケーションを作成する場合は，自動コミットではなく，トランザクションを使用することをお勧めします。

なお，SQL Anywhere の場合，自動コミットを設定していると，SQL 文の実行が成功するとコミット，失敗するとロールバックします。

TPBroker の OTS インタフェースでトランザクションを制御している場合

登録されている DBConnection オブジェクトが OTS インタフェースを使用している場合でもこのクラスの各メソッドは呼び出せますが，このような場合，DBTransaction オブジェクトではトランザクション制御はしないため，各メソッドの機能は無効になります。

機 能	メソッド名
トランザクションを開始します。	BeginTrans
トランザクションをコミットします。	Commit
DBTransaction オブジェクトの名前を取得します。	GetName
トランザクションの実行中かどうかを確認します。	InTransact
対象となるオブジェクトを生成した DBDriverManager オブジェクトのポインタを取得します。	Parent
DBTransaction オブジェクト(自分自身)を破棄します。	Remove
トランザクションをロールバックします。	Rollback
自動コミットを制御します。	SetAutoCommit

BeginTrans メソッド

機能

トランザクションを開始します。

形式

```
void BeginTrans (void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

トランザクションを開始します。SetAutoCommit メソッドで TRUE が設定されていた場合（自動コミット設定）は，FALSE に変更されます。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NO_CONNECTION_OBJECT
　DBConnection オブジェクトが登録されていません。

DB_ERROR_IN_TRANSACTION
　トランザクション処理中です。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_ERROR_DAB_ACCESS_ERROR
　DABroker でエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR
　システムコールでエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_TIMEOUT
　タイムアウトが発生したため制御を戻します。

Commit メソッド

機能

トランザクションをコミットします。

形式

```
void Commit(void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

BeginTrans メソッドによって開始されたトランザクションをコミットします。

対象とする DBConnection オブジェクトが非同期実行時，実行中，又は実行待ちのステートメントがある場合はコミットできません。コミットする前に，InWaitForDataSource メソッドを呼び出して，非同期実行中の処理がないことを確認してください。

コミットが成功すると，その時点から次のトランザクションが開始されます。あらためて BeginTrans メソッドを呼び出す必要はありません。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NO_CONNECTION_OBJECT
　DBConnection オブジェクトが登録されていません。

DB_ERROR_NOT_IN_TRANSACTION
　トランザクションが開始されていません。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中のステートメントがあります。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_ERROR_DAB_ACCESS_ERROR
　DABroker でエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR
　システムコールでエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_TIMEOUT
　タイムアウトが発生したため制御を戻します。

GetName メソッド

機能

DBTransaction オブジェクトの名前を取得します。

形式

```
LPCTSTR GetName(void)
```

引数

なし

戻り値

データ型：LPCTSTR

DBTransaction オブジェクトの名前。

機能詳細

DBTransaction オブジェクトの名前を取得します。DBTransaction オブジェクトの名前は，DBDriverManager クラスの Transaction メソッドの引数から生成されます。

発生する例外

なし

InTransact メソッド

機能

トランザクションの実行中かどうかを確認します。

形式

```
BOOLEAN InTransact(void)
```

引数

なし

戻り値

データ型：BOOLEAN

TRUE：トランザクション実行中です。

FALSE：自動コミットが設定されています。

機能詳細

現在トランザクションが実行されているかどうかを示します。

BeginTrans メソッド発行済み，又は SetAutoCommit メソッドで FALSE に設定している場合で，トランザクションが実行されている場合は TRUE が返されます。

SetAutoCommit メソッドで TRUE に設定し，ステートメントごとの自動コミットが実行されている場合は FALSE が返されます。

発生する例外

なし

Parent メソッド

機能

対象となるオブジェクトを生成した DBDriverManager オブジェクトのポインタを取得します。

形式

```
LPOBJECT Parent(void)
```

引数

なし

戻り値

データ型：LPOBJECT

対象となるオブジェクトを生成した DBDriverManager オブジェクトへのポインタ。

機能詳細

なし

発生する例外

なし

Remove メソッド

機能

DBTransaction オブジェクト（自分自身）を削除します。

形式

```
void Remove(void)
```

引数

なし

戻り値

なし

機能詳細

DBConnection オブジェクトと関連付けられている場合，ロールバックして DBConnection オブジェクトとの関連付けを解消した後，自分自身を削除します。

非同期処理を実行している場合は，実行待ちの処理をすべてキャンセルして，実行中の処理が終了するまで待ってから上記の処理を実行します。

発生する例外

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中ステートメントがあります。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

Rollback メソッド

機能

トランザクションをロールバックします。

形式

```
void Rollback(void) throw DBSQLCA
```

引数

なし

戻り値

なし

機能詳細

BeginTrans メソッドによって開始されたトランザクションをロールバックします。

対象とする DBConnection オブジェクトで非同期処理を実行している場合，実行中又は実行待ちの処理があるとロールバックできません。ロールバックする前に，InWaitForDataSource メソッドを呼び出して，非同期実行中の処理がないことを確認してください。

ロールバックが成功すると，その時点から次のトランザクションが開始されます。あらためて BeginTrans メソッドを呼び出す必要はありません。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NO_CONNECTION_OBJECT
　DBConnection オブジェクトが登録されていません。

DB_ERROR_NOT_IN_TRANSACTION
　トランザクションが開始されていません。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中ステートメントがあります。

DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_ERROR_DAB_ACCESS_ERROR
　DABroker でエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR
　システムコールでエラーが発生しました。
　DBSQLCA クラスで詳細コードを確認してください。
DB_ERROR_TIMEOUT
　タイムアウトが発生したため制御を戻します。

SetAutoCommit メソッド

機能

自動コミットを制御します。

形式

```
void SetAutoCommit(BOOLEAN bAutoCommit = TRUE) throw DBSQLCA
```

引数

bAutoCommit

自動コミットの制御を指定します。

TRUE：ステートメントごとに自動コミットをします。

FALSE：ステートメントごとの自動コミットをやめて，トランザクションを開始します。

戻り値

なし

機能詳細

トランザクションの開始を明示的に宣言しない場合，SQL 文が 1 文実行されるごとに自動的にコミットされます。この自動コミットを制御します。デフォルトでは自動コミットをする（TRUE）になっています。

SetAutoCommit メソッドで FALSE を指定するか，又は BeginTrans メソッドを呼び出すと，トランザクションを開始します。設定を変更しただけの場合は，それまでのトランザクションが継続されます。次の SQL 実行時から有効になります。

SetAutoCommit メソッドを引数 bAutoCommit に TRUE を指定して呼び出すと，自動コミットされるようになります。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NO_CONNECTION_OBJECT
　DBConnection オブジェクトが登録されていません。
DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中ステートメントがあります。
DB_ERROR_DRIVER_ERROR
　DBMS でエラーが発生しました。

DB_ERROR_DAB_ACCESS_ERROR

DABroker でエラーが発生しました。

DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_DAB_SYSTEMCALL_ERROR

システムコールでエラーが発生しました。

DBSQLCA クラスで詳細コードを確認してください。

DB_ERROR_TIMEOUT

タイムアウトが発生したため制御を戻します。

5.13　classListTables クラスの詳細

テーブル一覧情報を管理します。

このクラスは，テーブル一覧情報として，所有者名及びテーブル名などを取得するメソッドを提供するクラスです。

機 能	プロパティ名
テーブル一覧のデータ数を取得します。	Count

機 能	メソッド名
テーブルの所有者名を取得します。	OwnerName
テーブルのデータベース名などの識別子を取得します。	Qualifier
テーブルのコメントを取得します。	Remarks
テーブル名を取得します。	TableName
テーブルの種別（テーブル，ビュー，シノニム，システムディクショナリ）を取得します。	Type

Count プロパティ

引数

なし

機能

テーブル一覧のデータ数を取得します。

非同期実行時に一覧の取得が完了していない場合は 0 を返します。

データ型

UINT32

データ値

0～4294967295 の整数値

OwnerName メソッド

機能

インデクスで指定したテーブルの所有者名を返します。

形式

```
LPCTSTR OwnerName(UINT32 dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まる一覧中のインデクスを指定します。

戻り値

データ型：LPCTSTR

所有者名を文字列で返します。

機能詳細

インデクスで指定したテーブルの所有者名を返します。

ORACLE の場合，システム所有のシノニムの場合には所有者名は取得できません。空文字列（長さ 0 の文字列）を返します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中です。

Qualifier メソッド

機能

インデクスで指定したテーブルのデータベース名などの識別子を返します。

形式

```
LPCTSTR Qualifier(UINT32 dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まる一覧中のインデクスを指定します。

戻り値

データ型：LPCTSTR

識別子を文字列で返します。

機能詳細

インデクスで指定したテーブルのデータベース名などの識別子を返します。

なお，現在のバージョンでは，常に空文字列（長さ 0 の文字列）を返します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中です。

Remarks メソッド

機能

インデクスで指定したテーブルのコメントを返します。

形式

```
LPCTSTR Remarks(UINT32 dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まる一覧中のインデクスを指定します。

戻り値

データ型：LPCTSTR

コメントを文字列で返します。

機能詳細

コメントが設定されていればインデクスで指定したテーブルのコメントを返します。

コメントが設定されていなければ空文字列を返します。

ORACLE の場合，SYNONYM（システム所有のものを含む）はコメントを取得できません。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中です。

TableName メソッド

機能

インデクスで指定したテーブルのテーブル名を取得します。

形式

```
LPCTSTR TableName(UINT32 dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まる一覧中のインデクスを指定します。

戻り値

データ型：LPCTSTR

フィールド名を文字列で返します。

機能詳細

インデクスで指定したテーブルのテーブル名を取得します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中です。

Type メソッド

機能

テーブルの種別（テーブル，ビュー，シノニム，システムディクショナリ）を取得します。

形式

```
UINT16 Type(UINT32 dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まる一覧中のインデクスを指定します。

戻り値

データ型：UINT32

識別子を文字列で返します。

機能詳細

インデクスで指定したテーブルの種別を返します。

次の四つのマクロのうちのどれかが返ります。

- TBL_TYPE_TABLE（システム所有のテーブルを除く）
- TBL_TYPE_VIEW（システム所有のビューを除く）
- TBL_TYPE_SYNONYM（システム所有のシノニムを除く）
- TBL_TYPE_SYSTEM （システム所有のオブジェクトすべて）

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中です。

5.14　classListColumns クラスの詳細

フィールド一覧情報を管理します。

このクラスは，フィールド一覧情報として，フィールド名や属性値などを取得するメソッドを提供するクラスです。

機 能	プロパティ名
フィールド一覧のデータ数を取得します。	Count

機 能	メソッド名
繰り返し列の要素の数を取得します。	ArraySize
フィールド名を取得します。	ColumnName
フィールドのデータ型を C++の型で取得します。	CType
フィールドのデータ型を DBMS の型で取得します。	DBType
NULL を許すかどうかを確認します。	Nullable
フィールドの最大長を取得します。Type が COL_TYPE_NUMERIC の場合，小数点以下を含めた数値の有効桁数です。	Precision
フィールドのコメントを取得します。	Remarks
Type メソッドで返すフィールドの型が COL_TYPE_NUMERIC の場合，小数点位置を取得します。	Scale
フィールドのデータ型を取得します。	Type
値の重複を許すかどうかを確認します。	Uniqueness

Count プロパティ

引数

なし

機能

フィールド一覧のデータ数を取得します。

非同期実行時に一覧の取得が完了していない場合は 0 を返します。

データ型

UINT32

データ値

0～4294967295 の整数値

ArraySize メソッド

機能

繰り返し列の要素の数を取得します。

形式

```
UINT32 ArraySize(UINT32 ui32Index) throw DBSQLCA
```

引数

ui32Index

フィールド番号を，1 ～フィールド数の範囲の値で指定します。

戻り値

データ型：UINT32

繰り返し列の要素の数を返します。

機能詳細

フィールド番号によって特定した繰り返し列の，要素の数を返します。

指定したフィールドが繰り返し列でない場合は，0 が返ります。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数 ui32Index で 0 が指定されています。

ColumnName メソッド

機能

インデクスで指定したフィールドのフィールド名を取得します。

形式

```
LPCTSTR ColumnName(UINT32 dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まる一覧中のインデクスを指定します。

戻り値

データ型：LPCTSTR

フィールド名を文字列で返します。

機能詳細

インデクスで指定したフィールドのフィールド名を取得します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中です。

CType メソッド

機能

フィールドのデータ型を C++の型で取得します。

形式

```
DBR_CTYPE CType(UINT32 dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まる一覧中のインデクスを指定します。

戻り値

データ型：DBR_CTYPE

インデクスで指定したフィールドの C++データ型を返します。

機能詳細

インデクスで指定したフィールドの C++データ型を取得します。

戻り値として返る値については,「7.1 クラスライブラリで扱うデータ型と変換規則」を参照してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中です。

DBType メソッド

機能

フィールドのデータ型を DBMS の型で取得します。

形式

```
UINT16 DBType(UINT32 dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まる一覧中のインデクスを指定します。

戻り値

データ型：UINT16

インデクスで指定したフィールドの DBMS のデータ型を返します。

機能詳細

インデクスで指定したフィールドの DBMS のデータ型を返します。

データ型の詳細については，使用する DBMS のマニュアルを参照してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中です。

Nullable メソッド

機能

インデクスで指定したフィールドで NULL 値を許すかどうかを確認します。

形式

```
BOOLEAN Nullable(UINT32 dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まる一覧中のインデクスを指定します。

戻り値

データ型：BOOLEAN

TRUE：NULL を許します。

FALSE：NULL を許しません。

機能詳細

インデクスで指定したフィールドが NULL を許すかどうかを BOOLEAN 値で返します。

XDM/SD の場合の戻り値は常に FALSE です。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中です。

Precision メソッド

機能

インデクスで指定したフィールドの最大長を取得します。

形式

```
UINT32 Precision(UINT32 dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まる一覧中のインデクスを指定します。

戻り値

データ型：UINT32

インデクスで指定したフィールドの最大長を返します。

機能詳細

インデクスで指定したフィールドの最大長を取得します。

Type メソッドで返すフィールドの型が COL_TYPE_NUMERIC の場合，小数点以下を含めた数値の有効桁数を返します。

Type メソッドの戻り値が以下に示す値の場合はフィールドの定義長を返します。

COL_TYPE_CHAR

COL_TYPE_VARCHAR

COL_TYPE_LONGVARCHAR

COL_TYPE_BINARY

COL_TYPE_VARBINARY

COL_TYPE_LONGVARBINARY

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中です。

Remarks メソッド

機能

インデクスで指定したフィールドのコメントを取得します。

形式

```
LPCTSTR Remarks(UINT32 dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まる一覧中のインデクスを指定します。

戻り値

データ型：LPCTSTR

コメントを文字列で返します。

機能詳細

コメントが設定されている場合，インデクスで指定したフィールドのコメントを文字列で取得します。

コメントが設定されていない場合，空文字列を返します。

フィールドのコメントを取得できるのは SQL Anywhere，Adaptive Server Anywhere，SQL Server 及び HiRDB です。他の DBMS の場合は常に空文字列（長さ 0 の文字列）を返します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中です。

Scale メソッド

機能

Type メソッドで返すフィールドの型が COL_TYPE_NUMERIC の場合，小数点位置を取得します。

形式

```
INT32 Scale(UINT32 dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まる一覧中のインデクスを指定します。

戻り値

データ型：INT32

インデクスで指定したフィールドの小数点位置を返します。

機能詳細

インデクスで指定したフィールドの，Type メソッドで取得したフィールドの型が
COL_TYPE_NUMERIC の場合，小数点位置を取得します。

この値が 0 のとき整数となります。また，マイナスの場合桁上がりとなります。

ただし，Type メソッドで取得したフィールドの型が次に示す値の場合，Scale メソッドは常に 0 を返します。

COL_TYPE_CHAR

COL_TYPE_VARCHAR

COL_TYPE_LONGVARCHAR

COL_TYPE_BINARY

COL_TYPE_VARBINARY

COL_TYPE_LONGVARBINARY

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中です。

Type メソッド

機能

インデクスで指定したフィールドのデータ型を取得します。

形式

```
UINT16 Type(UINT32 dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まる一覧中のインデクスを指定します。

戻り値

データ型：UINT16

「7.1　クラスライブラリで扱うデータ型と変換規則」のデータ型対応表を参照してください。

現バージョンでサポートしていないデータ型のフィールドの場合，Type メソッドの戻り値は COL_TYPE_NULL となります。

機能詳細

インデクスで指定したフィールドのデータ型を返します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中です。

Uniqueness メソッド

機能

インデクスで指定したフィールドが値の重複を許すかどうかを確認します。

形式

```
BOOLEAN Uniqueness(UINT32 dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まる一覧中のインデクスを指定します。

戻り値

データ型：BOOLEAN

インデクスで指定したフィールドが重複した値を許すかどうかを返します。

TRUE：重複した値を許しません。

FALSE：重複した値を許します。

機能詳細

使用する DBMS が ORACLE の場合，インデクスで指定したフィールドが重複した値を許すかどうかを BOOLEAN 値で返します。

使用する DBMS が ORACLE 以外の場合は，常に FALSE を返します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中です。

5.15　classListProcedures クラスの詳細

ストアドプロシジャの一覧情報を管理します。

このクラスは，プロシジャ一覧情報として，プロシジャ名，その定義などを取得するメソッドを提供するクラスです。

使用している DBMS が XDM/RD，XDM/SD，又は SQL/K の場合はどの一覧も取得できません。取得した場合,常に 0 件が返ります。

機 能	プロパティ名
プロシジャ一覧のデータ数を取得します。	Count

機 能	メソッド名
プロシジャの定義を取得します。	Define
所有者名を取得します。	OwnerName
プロシジャ名を取得します。	ProcedureName
データベース名などの識別子を取得します。	Qualifier
コメントを取得します。	Remarks

Count プロパティ

引数

なし

機能

プロシジャ一覧のデータ数を取得します。

非同期実行時に一覧の取得が完了していない場合は 0 を返します。

データ型

UINT32

データ値

0～4294967295 の整数値

Define メソッド

機能

インデクスで指定したプロシジャの定義を返します。

形式

```
LPCTSTR Define(UINT32 dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まる一覧中のインデクスを指定します。

戻り値

データ型：LPCTSTR

プロシジャの定義を文字列で返します。

機能詳細

インデクスで指定したプロシジャの定義を返します。

SQL Anywhere，Adaptive Server Anywhere，SQL Server の場合は，このメソッドは利用できません。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中です。

OwnerName メソッド

機能

インデクスで指定したプロシジャの所有者名を返します。

形式

```
LPCTSTR OwnerName(UINT32 dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まる一覧中のインデクスを指定します。

戻り値

データ型：LPCTSTR

所有者名を文字列で返します。

機能詳細

インデクスで指定したプロシジャの所有者名を返します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中です。

ProcedureName メソッド

機能

インデクスで指定したプロシジャのプロシジャ名を返します。

形式

```
LPCTSTR ProcedureName(UINT32 dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まる一覧中のインデクスを指定します。

戻り値

データ型：LPCTSTR

プロシジャ名を文字列で返します。

機能詳細

インデクスで指定したプロシジャのプロシジャ名を返します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中です。

Qualifier メソッド

機能

インデクスで指定したプロシジャのデータベース名などの識別子を返します。

形式

```
LPCTSTR Qualifier(UINT32 dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まる一覧中のインデクスを指定します。

戻り値

データ型：LPCTSTR

識別子を返します。

機能詳細

インデクスで指定したプロシジャのデータベース名などの識別子を返します。

なお，この指定は常に空文字列（長さ 0 の文字列）を返します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中です。

Remarks メソッド

機能

インデクスで指定したプロシジャのコメントを返します。

形式

```
LPCTSTR Remarks(UINT32 dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まる一覧中のインデクスを指定します。

戻り値

データ型：LPCTSTR

コメントを文字列で返します。

機能詳細

インデクスで指定したプロシジャのコメントを文字列で返します。

なお，コメントを取得できるのは SQL Anywhere,Adaptive Server Anywhere,SQL Server の場合だけで，これ以外の DBMS では常に空文字列（長さ 0 の文字列）を返します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中です。

5.16　classListProcedureColumns クラスの詳細

ストアドプロシジャのパラメタ一覧情報を管理します。

このクラスは，パラメタ名や属性，及びパラメタの用途などを取得するメソッドを提供するクラスです。

使用している DBMS が XDM/RD，XDM/SD，又は SQL/K の場合は，このクラスは利用できません。

機 能	プロパティ名
プロシジャのパラメタ一覧のデータ数を取得します。	Count

機 能	メソッド名
パラメタ名を取得します。	ColumnName
パラメタの用途を取得します。	ColumnType
パラメタのデータ型を C++の型で取得します。	CType
パラメタのデータ型を DBMS の型で取得します。	DBType
パラメタで NULL 値を許すかどうかを確認します。	Nullable
パラメタの長さの最大値を取得します。	Precision
コメントを取得します。	Remarks
Type メソッドで返すパラメタの型が COL_TYPE_NUMERIC の場合，小数点位置を取得します。	Scale
パラメタのデータ型を取得します。	Type

Count プロパティ

引数

なし

機能

ストアドプロシジャのパラメタ一覧のデータ数を取得します。

非同期実行時に一覧の取得が完了していない場合は 0 を返します。

データ型

UINT32

データ値

0～4294967295 の整数値

ColumnName メソッド

機能

ストアドプロシジャのパラメタ名を取得します。

形式

```
LPCTSTR ColumnName(UINT32 dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まる一覧中のインデクスを指定します。

戻り値

データ型：LPCTSTR

パラメタ名を文字列で返します。

機能詳細

インデクスで指定したプロシジャのパラメタ名を取得します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中です。

ColumnType メソッド

機能

ストアドプロシジャのパラメタの用途を取得します。

形式

```
UINT16 ColumnType(UINT32 dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まる一覧中のインデクスを指定します。範囲外のインデクスを指定しないようにしてください。

戻り値

データ型：UINT16

次の三つのマクロの値のうちどれか一つを返します。

- PARAM_TYPE_INPUT
- PARAM_TYPE_OUTPUT
- PARAM_TYPE_INOUT

機能詳細

インデクスで指定したプロシジャのパラメタの用途を取得します。

- PARAM_TYPE_INPUT：入力用のパラメタ
- PARAM_TYPE_OUTPUT：出力用のパラメタ
- PARAM_TYPE_INOUT：入出力用のパラメタ

非同期実行時で一覧の取得が完了する前に，このメソッドは呼び出せません。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中です。

CType メソッド

機能

パラメタのデータ型を C++の型で取得します。

形式

```
DBR_CTYPE CType(UINT32 dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まる一覧中のインデクスを指定します。

戻り値

データ型：DBR_CTYPE

インデクスで指定したプロシジャのパラメタのデータ型を C++データ型で返します。

機能詳細

インデクスで指定したプロシジャのパラメタの C++データ型を返します。

返す値については，「7.1　クラスライブラリで扱うデータ型と変換規則」を参照してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中です。

DBType メソッド

機能

パラメタのデータ型を DBMS の型で取得します。

形式

```
UINT16 DBType(UINT32 dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まる一覧中のインデクスを指定します。

戻り値

データ型：UINT16

インデクスで指定したプロシジャのパラメタの DBMS のデータ型を返します。詳細については，「7.1　クラスライブラリで扱うデータ型と変換規則」のデータ型対応表を参照してください。

機能詳細

インデクスで指定したプロシジャのパラメタの DBMS のデータ型を返します。

データ型の詳細については，使用する DBMS のマニュアルを参照してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中です。

Nullable メソッド

機能

パラメタで NULL 値を許すかどうかを確認します。

形式

```
BOOLEAN Nullable(UINT32 dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まる一覧中のインデクスを指定します。

戻り値

データ型：BOOLEAN

TRUE：NULL を許します。

FALSE：NULL を許しません。

機能詳細

インデクスで指定したプロシジャのパラメタで NULL を許すかどうかを，BOOLEAN 型の値で取得します。

NULL を許す場合 TRUE を，NULL を許さない場合 FALSE を返します。

HiRDB を使用している場合は常に TRUE を返します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中です。

Precision メソッド

機能

プロシジャで使用するパラメタの長さの最大値を取得します。

形式

```
UINT32 Precision(UINT32 dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まる一覧中のインデクスを指定します。

戻り値

データ型：UINT32

インデクスで指定したプロシジャのパラメタ長の最大値を返します。

機能詳細

インデクスで指定したプロシジャのパラメタ長の最大値を取得します。

Type メソッドで取得したパラメタの型が COL_TYPE_NUMERIC の場合，小数点以下を含めた数値の有効桁数を返します。

ただし，Type メソッドで取得したパラメタの型が以下に示す値の場合はパラメタの定義長を返します。

COL_TYPE_CHAR

COL_TYPE_VARCHAR

COL_TYPE_LONGVARCHAR

COL_TYPE_BINARY

COL_TYPE_VARBINARY

COL_TYPE_LONGVARBINARY

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中です。

Remarks メソッド

機能

パラメタにコメントがある場合，そのコメントを取得します。

形式

```
LPCTSTR Remarks(UINT32 dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まる一覧中のインデクスを指定します。

戻り値

データ型：LPCTSTR

コメントを文字列で返します。

機能詳細

インデクスで指定したプロシジャのパラメタのコメントを文字列で取得します。

なお，SQL Anywhere，Adaptive Server Anywhere,SQL Server の場合だけ指定できます。これ以外の DBMS の場合は，常に空文字列（長さ 0 の文字列）を返します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中です。

Scale メソッド

機能

Type メソッドで返すパラメタの型が COL_TYPE_NUMERIC の場合，小数点位置を取得します。

形式

```
INT32 Scale(UINT32 dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まる一覧中のインデクスを指定します。

戻り値

データ型：INT32

インデクスで指定したプロシジャのパラメタの小数点位置を返します。

機能詳細

インデクスで指定したプロシジャの，パラメタの小数点位置を取得します。

Type メソッドで取得したパラメタの型が COL_TYPE_NUMERIC の場合，小数点位置を返します。この値が 0 のとき整数となります。また，マイナスの場合桁上がりとなります。

そのほかの型については，Precision メソッドを参照してください。

ただし，Type メソッドで取得したパラメタの型が以下に示す値の場合，Scale メソッドは常に 0 を返します。

COL_TYPE_CHAR

COL_TYPE_VARCHAR

COL_TYPE_LONGVARCHAR

COL_TYPE_BINARY

COL_TYPE_VARBINARY

COL_TYPE_LONGVARBINARY

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中です。

Type メソッド

機能

プロシジャのパラメタのデータ型を取得します。

形式

```
UINT16 Type(UINT32 dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まる一覧中のインデクスを指定します。

戻り値

データ型：UINT16

「7.1　クラスライブラリで扱うデータ型と変換規則」のデータ型対応表を参照してください。

現在のバージョンでサポートしていないデータ型のパラメタの場合，Type メソッドの戻り値は COL_TYPE_NULL となります。

機能詳細

インデクスで指定したプロシジャのパラメタのデータ型を返します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中です。

5.17　classListPrimaryKeys クラスの詳細

テーブル中のプライマリキーを構成するフィールドの一覧情報を管理します。

このクラスは，フィールド名，プライマリーキー名，テーブル名，などを取得するメソッドを提供するクラスです。

使用している DBMS が XDM/RD，XDM/SD，又は SQL/K の場合は，このクラスは利用できません。

機 能	プロパティ名
プライマリキーを構成するフィールドの一覧のデータ数を取得します。	Count

機 能	メソッド名
フィールド名を取得します。	ColumnName
プライマリキー名を取得します。	KeyName
所有者名を取得します。	OwnerName
フィールドのプライマリキー中の優先順位を取得します。	Sequence
テーブル名を取得します。	TableName

Count プロパティ

引数

なし

機能

プライマリキーを構成するフィールドの一覧のデータ数を取得します。

非同期実行時に一覧の取得が完了していない場合は 0 を返します。

HiRDB，又は XDM/RD を使用している場合は常に 0 を返します。

データ型

UINT32

データ値

0～4294967295 の整数値

ColumnName メソッド

機能

プライマリキー一覧中のインデクスで指定したフィールドのフィールド名を返します。

形式

```
LPCTSTR ColumnName(UINT32 dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まる一覧中のインデクスを指定します。

戻り値

データ型：LPCTSTR

NULL 文字で終了する文字列を返します。

機能詳細

プライマリキー一覧中のインデクスで指定したフィールドのフィールド名を返します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中です。

KeyName メソッド

機能

プライマリキー一覧中のインデクスで指定したフィールドのプライマリキーの名称を返します。

形式

```
LPCTSTR KeyName(UINT32 dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まる一覧中のインデクスを指定します。

戻り値

データ型：LPCTSTR

プライマリキーの名称を返します。

機能詳細

プライマリキーのインデクスで指定したフィールドのキーの名称を文字列で返します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中です。

OwnerName メソッド

機能

プライマリキー一覧中のインデクスで指定したフィールドの所有者名を返します。

形式

```
LPCTSTR OwnerName(UINT32 dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まる一覧中のインデクスを指定します。

戻り値

データ型：LPCTSTR

所有者名を文字列で返します。

機能詳細

プライマリキー一覧中のインデクスで指定したフィールドの所有者名を返します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中です。

Sequence メソッド

機能

プライマリキー一覧中のインデクスで指定したフィールドのプライマリキー中の優先順位を取得します。

形式

```
UINT32 Sequence(UINT32 dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まる一覧中のインデクスを指定します。

戻り値

データ型：UINT32

プライマリキー一覧中のインデクスで指定したフィールドのプライマリキー中の優先順位を返します。

機能詳細

プライマリキー一覧中のインデクスで指定したフィールドのプライマリキー中の優先順位を取得します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中です。

TableName メソッド

機能

プライマリキー一覧中のインデクスで指定したフィールドがあるテーブル名を返します。

形式

```
LPCTSTR TableName(UINT32 dwIndex) throw DBSQLCA
```

引数

dwIndex

1 から始まる一覧中のインデクスを指定します。

戻り値

データ型：LPCTSTR

テーブル名を文字列で返します。

機能詳細

プライマリキー一覧中のインデクスで指定したフィールドがあるテーブル名を返します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

DB_ERROR_IN_ASYNC_EXECUTE
　非同期実行処理中です。

6 共通関数詳細

DABroker では，C++による開発向けのプログラミングインタフェースとして，C++クラスライブラリを提供します。ここでは，エラー処理に使うクラスライブラリ及び繰り返し列に関するプロパティやメソッドの詳細について説明します。

6.1　DBSQLCA クラスの詳細

エラー情報を管理します。

このクラスは，エラー情報やその数などを取得するメソッドを提供するクラスです。

同期処理を実行している場合，スローされた例外をキャッチします。非同期処理を実行している場合は，最大 100 個までのエラーを保存できます。各種のエラー情報を保存するプロパティを持ち，各プロパティ値を参照するメソッドや，保存されているエラーを削除するメソッドを提供します。

DBMS 非依存エラーコードについて

DBMS 非依存エラーコードとは，DABroker が，DBMS ごとに異なるエラーコードの違いを吸収し，アプリケーションで DBMS を意識しないでエラー処理を記述するためのものです。

e_USERCODE プロパティと e_USERERROR プロパティには,DBMS 非依存エラーコードが設定されます。内容については「2.4 エラー処理」を参照してください。

エラー情報の参照

同期処理実行の場合

アプリケーションで同期処理を実行している場合は, 一度に一つのエラーが DBSQLCA オブジェクトにスローされます。

エラー情報を取得するためには，オブジェクトのプロパティ値を参照します。

（例 1）

```
catch (DBSQLCA ca)
 {
   code=ca.e_USERCODE   // codeにエラーコードを取得
 }
```

非同期処理実行の場合

非同期処理を実行している場合に発生したエラーは，DBSQLCA オブジェクトに複数保存されています。この DBSQLCA オブジェクトへのポインタは，各クラスの GetErrorStatus メソッドで取得できます。

（例 2）

```
DBSQLCA     *pError;
pError = pStatement->GetErrorStatus();
         // pErrorにDBSQLCAオブジェクトのポインタを取得
```

DBSQLCA オブジェクトに保存される複数のエラーには，発生した順番に 1 から始まるインデクス番号が割り当てられます。また，ある時点までに取得したエラーの個数が，Count プロパティに設定されます。エラー情報は最大 100 個まで保存しますが，それ以降は古いエラー情報からエラーが発生するたびに削除されます。

現在エラー情報が幾つ保存されているかを知るには, Count プロパティの値を参照します。例えば, Count プロパティの値が「3」の場合は，現在 3 個のエラーが保存されています。

エラー情報を取得するには，DBSQLCA オブジェクトの Get～メソッドを使います。これらのメソッドでは，取得したいエラーのインデクス番号を指定できます。例えば， Count プロパティの値が「3」の場合に，最新のエラー情報を取得するためには， Get～メソッドのインデクスに「3」を設定します。過去のエラー情報を参照したい場合は，「3」より小さい数を指定します。

過去のエラー情報をすべて削除したい場合は Delete メソッドを呼び出します。Delete メソッドでエラー情報を削除すると，その後はまた発生した順番に，1 から始まるインデクス番号が割り当てられて，複数のエラーが保存されます。

機 能	プロパティ名
DBSQLCA オブジェクトが保存しているエラーの総数が設定されます。	Count
最新のエラーの，DABroker メッセージテキストが設定されます。	ErrorMessage
最新のエラーの，DABroker エラーコードが設定されます。	RetCode
最新のエラーの，データベース固有のエラーコードが設定されます。	e_SQLCODE
最新の UPDATE/INSERT/DELETE 操作で実際に処理されたレコード数が設定されます。	e_SQLCOUNT
最新のエラーの，データベース固有のエラーメッセージが設定されます。	e_SQLERROR
最新のエラーの，データベース固有の SQLSTATE 値が設定されます。	e_SQLSTATE
最新のエラーの， DBMS 非依存エラーコードが設定されます。	e_USERCODE
最新のエラーの， DBMS 非依存エラーコードのエラーメッセージが設定されます。	e_USERERROR

機 能	メソッド名
DBSQLCA オブジェクトを削除します。	Delete
指定されたエラーの，DABroker メッセージテキストを取得します。	GetErrorMessage
指定されたエラーの，DABroker エラーコードを取得します。	GetRetCode
指定されたエラーの，データベース固有のエラーコードを取得します。	GetSQLCODE
指定された UPDATE/INSERT/DELETE 操作で実際に処理されたレコード数を取得します。	GetSQLCOUNT
指定されたエラーの，データベース固有のエラーメッセージを取得します。	GetSQLERROR
指定されたエラーの，データベース固有の SQLSTATE 値を取得します。	GetSQLSTATE
指定されたエラーの，DBMS 非依存エラーコードを取得します。	GetUSERCODE
指定されたエラーの，DBMS 非依存エラーコードのエラーメッセージを取得します。	GetUSERERROR

Count プロパティ

引数

なし

機能

DBSQLCA オブジェクトに保存しているエラーの個数が設定されます。

同期処理中の DBSQLCA オブジェクトでは，常に 1 となります。

非同期処理の DBSQLCA オブジェクトの場合，エラーが発生していなければ 0 となります。複数のエラーを保存している場合，保存しているエラーの個数が設定されます。なお， DBSQLCA オブジェクトでは，新しいエラーから最大 100 個までのエラーを保存できます。

データ型

INT16

データ値

0～100

ErrorMessage プロパティ

引数

なし

機能

C++クラスライブラリで発生したエラーのメッセージテキストが設定されます。DBDriverManager クラスの InitializeMessage メソッドを実行していない場合は，このプロパティには NULL が設定されます。

非同期処理の DBSQLCA オブジェクトの場合，エラーが発生していなければ NULL が設定されます。複数のエラーを保存している場合は，最新のメッセージテキストが設定されます。

Red Hat Linux 版では，エラーメッセージは英語で出力されます（ロケールや環境ファイルにも依存しません）。

データ型

LPCTSTR

データ値

NULL 文字で終了する文字列

e_SQLCODE プロパティ

引数

なし

機能

DBMS のエラー，DABroker 本体のエラー，システムコール（OS のエラー），又は C++クラスライブラリのエラーが発生した場合の詳細コードが設定されます。

エラーの内容が上記以外の場合は 0 が返されます。

非同期処理の DBSQLCA オブジェクトの場合，エラーが発生していなければ 0 が設定されます。複数のエラーを保存している場合は，最新の詳細コードが設定されます。

データ型

INT32

データ値

機能に示したエラーの発生元のエラーコードに依存します。内容は，それぞれについて説明したドキュメントを参照してください。

e_SQLCOUNT プロパティ

引数

なし

機能

UPDATE，INSERT，DELETE 文の実行中にエラーが発生した場合，その発生までに処理されたレコード数が設定されます。

非同期処理の DBSQLCA オブジェクトの場合，エラーが発生していなければ 0 が設定されます。複数のエラーを保存している場合は，最新のエラーでの処理済みレコード数が設定されます。

データ型

UINT32

データ値

0～4294967295 の整数値

e_SQLERROR プロパティ

引数

なし

機能

DBMS のエラー，DABroker 本体のエラー，システムコール（OS のエラー），又は C++クラスライブラリのエラーが発生した場合のエラーメッセージが設定されます。

エラーの内容が上記以外の場合は 0 が返されます。

このプロパティは DBDriverManager クラスの InitializeMessage メソッドを実行していても，していなくても，エラーメッセージを取得します。

非同期処理の DBSQLCA オブジェクトの場合，エラーが発生していなければ 0 が設定されます。複数のエラーを保存している場合は，最新のエラーメッセージが設定されます。

データ型

LPCTSTR

データ値

機能に示したエラーの発生元のエラーコードに依存します。内容は，それぞれについて説明したドキュメントを参照してください。

e_SQLSTATE プロパティ

引数

なし

機能

データベース固有の SQLSTATE 値が設定されます。ただし，接続先 DBMS として ORACLE7 を使用している場合は空文字列が設定されます。

エラーの発生場所がデータベースでなく C++クラスライブラリの場合は NULL が設定されます。

非同期処理の DBSQLCA オブジェクトの場合，エラーが発生していなければ 0 が設定されます。複数のエラーを保存している場合は，最新のエラーメッセージが設定されます。

データ型

DBR_SQLSTATE

データ値

DBMS のマニュアルを参照してください。

e_USERCODE プロパティ

引数

なし

機能

DBMS 非依存エラーコードが設定されます。

DBMS 非依存エラーコードについては，「2.4 エラー処理」を参照してください。

DBMS のエラーコードに対応した DBMS 非依存エラーコードがない場合は 0 が設定されます。

データ型

INT32

データ値

-2147483648～2147483647 の整数値

e_USERERROR プロパティ

引数

なし

機能

DBMS 非依存エラーコードに対応したメッセージが設定されます。このプロパティは，e_USERCODE と対になっています。

DBMS 非依存エラーコードについては，「2.4 エラー処理」を参照してください。

DBMS のエラーコードに対応した DBMS 非依存エラーコードがない場合は 0 が設定されます。

Red Hat Linux 版では，エラーメッセージは英語で出力されます（ロケールや環境ファイルにも依存しません）。

データ型

LPCTSTR

データ値

NULL 文字で終了する文字列

RetCode プロパティ

引数

なし

機能

C++クラスライブラリで発生した最新のエラーコードが設定されます。

RetCode プロパティに次の値が返された場合，その詳細については，e_SQLERROR，又は e_SQLCODE プロパティの値を参照してください。

- DB_ERROR_DRIVER_ERROR
- DB_ERROR_DAB_ACCESS_ERROR
- DB_ERROR_DAB_SYSTEMCALL_ERROR
- DB_ERROR_DAB_ILLEGAL_VALUE

DB_ERROR_DRIVER_ERROR（データベースで発生したエラー)の場合，e_SQLSTATE プロパティにも値が返されることがあります。

非同期処理の DBSQLCA オブジェクトの場合，エラーが発生していなければ 0 が設定されます。複数のエラーを保存している場合は，最新のエラーコードが設定されます。

データ型

DBR_RETCODE

データ値

「8.2　C++クラスライブラリのエラー情報」を参照してください。

Delete メソッド

機能

DBSQLCA オブジェクトに保存されているエラー情報を削除します。

形式

```
void Delete(void)
```

引数

なし

戻り値

データ型：void

複数のエラーが保存されている場合，保存されているエラーを削除します。

機能詳細

非同期実行の場合，DBSQLCA オブジェクトには最大 100 個のエラーが保存されます。

Delete メソッドを呼び出すと，保存されているエラー情報を一度に削除できます。

GetErrorMessage メソッド

機能

指定されたエラーに設定されている，DABroker C++クラスライブラリのエラーメッセージを取得します。

形式

```
LPCTSTR GetErrorMessage(INT16 nIndex) throw DBSQLCA
```

引数

nIndex

複数のエラーが保存されている場合，エラーのインデクス番号を指定します。

戻り値

データ型：LPCTSTR

インデクスで指定されたエラーに設定されている，C++クラスライブラリのエラーメッセージを取得します。

機能詳細

非同期実行などで複数のエラーが返った場合，インデクスで指定したエラーに設定されている，DABroker のエラーメッセージを取得します。最新のエラーメッセージを知りたい場合は，ErrorMessage プロパティの値を参照してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

GetRetCode メソッド

機能

指定されたエラーに設定されている，DABroker C++クラスライブラリのエラーコードを取得します。

形式

```
LPCTSTR GetRetCode(INT16 nIndex) throw DBSQLCA
```

引数

nIndex

複数のエラーが保存されている場合，エラーのインデクス番号を指定します。

戻り値

データ型：LPCTSTR

インデクスで指定されたエラーに設定されている，DABroker エラーコードを取得します。

機能詳細

非同期実行などで複数のエラーが返った場合，インデクスで指定したエラーに設定されている，DABroker のエラーコードを取得します。最新のエラーコードを知りたい場合は，RetCode プロパティの値を参照してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

GetSQLCODE メソッド

機能

指定されたエラーに設定されている，データベース固有のエラーコードを取得します。

形式

```
LPCTSTR GetSQLCODE(INT16 nIndex) throw DBSQLCA
```

引数

nIndex

複数のエラーが保存されている場合，エラーのインデクス番号を指定します。

戻り値

データ型：LPCTSTR

インデクスで指定されたエラーに設定されている，データベース固有のエラーコードを取得します。

機能詳細

非同期実行などで複数のエラーが戻された場合，インデクスで指定したエラーに設定されている，データベース固有のエラーコードを取得します。最新のエラーコードを知りたい場合は，e_SQLCODE プロパティの値を参照してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

GetSQLCOUNT メソッド

機能

UPDATE，INSERT，DELETE 文の実行中にエラーが発生した場合，その発生までに処理されたレコード数を取得します。

形式

```
LPCTSTR GetSQLCOUNT(INT16 nIndex) throw DBSQLCA
```

引数

nIndex

複数のエラーが保存されている場合，エラーのインデクス番号を指定します。

戻り値

データ型：LPCTSTR

インデクスで指定されたエラーが発生する前に，UPDATE，INSERT，DELETE 操作で実際に処理されたレコード数。

機能詳細

非同期実行などで複数のエラーが戻された場合，インデクスで指定されたエラーが発生する前に，UPDATE，INSERT，DELETE 操作で処理されたレコード数を取得します。最新のレコード数を知りたい場合は，e_SQLCOUNT プロパティの値を参照してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

GetSQLERROR メソッド

機能

指定されたエラーに設定されている，データベース固有エラーメッセージを取得します。

形式

```
LPCTSTR GetSQLERROR(INT16 nIndex) throw DBSQLCA
```

引数

nIndex

複数のエラーが保存されている場合，エラーのインデクス番号を指定します。

戻り値

データ型：LPCTSTR

インデクスで指定されたエラーに設定されている，データベース固有エラーメッセージを取得します。

機能詳細

非同期実行などで複数のエラーが戻された場合，インデクスで指定されたエラーに設定されている，データベース固有のエラーメッセージを取得します。最新のエラーメッセージを知りたい場合は，e_SQLERROR プロパティの値を参照してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

GetSQLSTATE メソッド

機能

指定されたエラーに設定されている，データベース固有の SQLSTATE 値を取得します。

形式

```
LPCTSTR GetSQLSTATE(INT16 nIndex) throw DBSQLCA
```

引数

nIndex

複数のエラーが保存されている場合，エラーのインデクス番号を指定します。

戻り値

データ型：LPCTSTR

インデクスで指定されたエラーに設定されている，データベース固有の SQLSTATE 値を取得します。

機能詳細

非同期実行処理で複数のエラーが戻された場合，インデクスで指定されたエラーに設定されている，データベース固有の SQLSTATE 値を取得します。最新のエラーの SQLSTATE 値を知りたい場合は，e_SQLSTATE プロパティの値を参照してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

GetUSERCODE メソッド

機能

指定されたエラーに設定されている，DBMS 非依存エラーコードを取得します。

形式

```
LPCTSTR GetSQLUSERCODE(INT16 nIndex) throw DBSQLCA
```

引数

nIndex

複数のエラーが保存されている場合，エラーのインデクス番号を指定します。

戻り値

データ型：LPCTSTR

インデクスで指定されたエラーに設定されている，DBMS 非依存エラーコードを取得します。

機能詳細

非同期実行などで複数のエラーが戻された場合，インデクスで指定されたエラーに設定されている，DBMS 非依存エラーコードを取得します。最新のエラーコードを知りたい場合は，e_USERCODE プロパティの値を参照してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

GetUSERERROR メソッド

機能

指定されたエラーに設定されている，DBMS 非依存エラーコードのエラーメッセージを取得します。

形式

```
LPCTSTR GetSQLUSERERROR(INT16 nIndex) throw DBSQLCA
```

引数

nIndex

複数のエラーが保存されている場合，エラーのインデクス番号を指定します。

戻り値

データ型：LPCTSTR

インデクスで指定されたエラーに設定されている，DBMS 非依存エラーコードのエラーメッセージを取得します。

機能詳細

非同期実行などで複数のエラーが戻された場合，インデクスで指定されたエラーに設定されている，DBMS 非依存エラーコードのエラーメッセージを取得します。最新のエラーを知りたい場合は，e_USERERROR プロパティの値を参照してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数が指定できる範囲を超えています。

6.2　DBRArrayDataFactory クラス

繰り返し列を扱う DBArrayData クラスを管理します。DBRArrayDataFactory クラスは，DBRArrayData オブジェクトを生成するメソッドを提供するクラスです。

なお，このクラスの派生クラスは生成できません。

機 能	メソッド名
DBRArrayData オブジェクトを生成します。	CreateArrayData

DBRArrayData オブジェクトは，繰り返し列のデータを格納するオブジェクトです。

CreateArrayData メソッド

機能

更新のための DBRArrayData オブジェクトを生成し，DBRArrayDataPtr オブジェクトのポインタを取得します。

形式 1 繰り返し列の要素を一括して更新する場合

```
virtual DBRArrayDataPtr CreateArrayData
                     (INT32 i32Type,
                      INT32 i32MaxSize = 1,
                      INT32 i32Scale = DBR_ARRAYSCALE_DEFAULT)
                                              throw DBSQLCA
```

形式 2 既存のオブジェクトを基に更新用オブジェクトを作成する場合

```
virtual DBRArrayDataPtr CreateArrayData
           (const DBRArrayDataConstPtr& cpArray) throw DBSQLCA
```

引数

i32Type

繰り返し列のデータ型を，次に示すどれかで指定します。

COL_TYPE_INT16：2 バイト長符号付き 2 進整数。

COL_TYPE_INT32：4 バイト長符号付き 2 進整数。

COL_TYPE_SINGLE：4 バイト長浮動小数点数。

COL_TYPE_DOUBLE：8 バイト長浮動小数点数。

COL_TYPE_NUMERIC：符号付き 10 進数。(DECIMAL 型)

COL_TYPE_CHAR：固定長文字列。

COL_TYPE_VARCHAR：可変長文字列。最大長指定あり。

COL_TYPE_DATE：日付型。

COL_TYPE_TIME：時間型。

COL_TYPE_INTERVAL_YEAR：年間隔型。

COL_TYPE_INTERVAL_HOUR：時間間隔型。

i32MaxSize

引数 i32Type で指定したデータ型によって，指定する値が異なります。

- COL_TYPE_CHAR，又は COL_TYPE_VARCHAR の場合
  繰り返し列の要素長の最大値を 1～2147483647 の範囲のバイト数で指定します。指定する長さに終端文字分は含みません。
- COL_TYPE_NUMERIC の場合
  DBRArrayData クラスの SetData メソッドで設定する要素の値に合わせて，要素の精度（最大桁数）を 1～28 の範囲の値で指定します。
- その他のデータ型の場合
  この引数の値は無視されます。

i32Scale

- COL_TYPE_CHAR，又は COL_TYPE_VARCHAR の場合
  文字列データの 1 文字のバイト数を指定します。
  - 1
    1 文字の長さを 1 バイトとして扱います。
    HiRDB でのデータ型が CHAR 型，VARCHAR 型の場合に指定します。
  - 2
    1 文字の長さを 2 バイトとして扱います。
    HiRDB でのデータ型が NCHAR 型，及び NVARCHAR 型の場合に指定します。引数 i32MaxSize の値が奇数の時は，この引数に 2 は指定できません。
- COL_TYPE_NUMERIC の場合
  DBRArrayData クラスの SetData メソッドで設定する要素の値に合わせて，要素の小数点以下桁数を 1～28 の範囲の値で指定します。
- その他のデータ型の場合
  この引数の値は無視されます。

cpArray

繰り返し列の要素を更新するときに指定します。インスタンスを持たないオブジェクトは指定できません。また，引数は省略できません。

この引数に指定したオブジェクトが保持する DBRArrayData クラスのインスタンスと同じ状態の DBRArrayData オブジェクトを生成し，そのポインタを保持する DBRArrayDataConstPtr オブジェクトを返します。

- 参照用 DBArrayData オブジェクトから更新用 DBArrayData オブジェクトを生成する場合
  コピー元の DBRArrayDataConstPtr オブジェクトのポインタを指定します。
- 更新用 DBRArrayData オブジェクトが既にあり，その複製を作成する場合
  コピー元の DBRArrayDataPtr オブジェクトのポインタを指定します。

戻り値

データ型：DBRArrayDataPtr

DBRArrayDataPtr オブジェクトを返します。

機能詳細

DBRArrayData オブジェクトを生成し，DBRArrayDataPtr オブジェクトのポインタを取得します。

形式 1

ResultSet の内容を参照する必要のない場合，例えば繰り返し列の要素を新規作成する場合や要素全体を一括して更新する場合に，ResultSet に格納する繰り返し列を定義する必要があります。このような場合に，形式 1 を使ってデータ属性を定義し，DBRArrayData オブジェクトの Create メソッドを使って，設定する要素分の領域を確保します。

形式 2

既にある DBRArrayData オブジェクトを基に，その複製を更新用として生成する場合に指定します。ResultSet の繰り返し列を取得するときに GetField メソッドを呼び出しますが，このとき生成される DBRArrayData オブジェクトは参照だけができます。このような場合に，形式 2 の引数 pArray に参照元のオブジェクトを指定し，更新用の DBRArrayData オブジェクト生成します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_NOT_SUPPORTED
　引数 i32Type に，指定できないデータ型で値が指定されました。

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　指定できない範囲の値が指定されています。

DB_ERROR_MAXSIZE_NOT_EVEN
　引数 i32MaxSize の値が奇数の時に引数 i32Scale に 2 を指定しています。

DB_ERROR_NO_INSTANCE
　指定された cpArray オブジェクトがインスタンスを持っていません。

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

6.3　DBRArrayData クラス

繰り返し列のデータを操作するために，データ領域の確保，要素へのデータの設定，及び定義情報の参照などのメソッドを提供します。

このクラスの ArrayData オブジェクトはスマートポインタで管理されるため，ArrayDataPtr オブジェクトから ArrayData オブジェクトのポインタを取得し，メソッドを利用します。

このクラスは，new や delete 演算子を使って直接オブジェクトを生成したり，削除したりできません（スマートポインタクラスで管理するための参照カウンタを持ちます)。また，派生クラスは定義できません。

機 能	メソッド名
繰り返し列の要素を扱う ArrayData オブジェクト(データ領域)を確保します。	Create
フィールド数，又は要素の数を取得します。	GetArrayCount
指定した番号の要素の値を取得します。	GetData
要素のデータの型を取得します。	GetDataType
繰り返し列の要素の最大長又は精度を取得します。	GetPrecision
位取り又は 1 文字の長さ（バイト数）を取得します。	GetScale
指定された番号の要素に値を設定します。	SetData
要素に欠損値を設定します。	SetNull

Create メソッド

機能

DBRArrayDataFactory の CreateArrayData メソッドで繰り返し列のデータ属性が定義されている場合に，実際に設定する要素分の領域（ArrayData オブジェクト）を確保します。

形式

```
virtual void Create(INT32 i32ArrayCount) throw DBSQLCA
```

引数

i32ArrayCount

確保する要素数を，1 から 30,000 の範囲の値で指定します。ただし，データベースのテーブル定義で設定されている最大要素数以下の値で設定してください。

戻り値

なし

機能詳細

引数 i32ArrayCount で指定した要素数分のデータ領域を持つ，ArrayData オブジェクトを生成します。要素数は次のメソッドで取得できます。

簡易版：DBRResultSet クラスの GetArraySize メソッド。

詳細版：DBResultSetMetaData クラスの GetArraySize メソッド

ArrayData オブジェクトに要素の値を設定するには，SetData メソッドを呼び出します。

既に Create メソッドが呼び出されている場合，要素数は再設定されるため，それまで設定されていたデータ領域の値は初期化されます。

なお，DBPreparedStatement オブジェクトの実行中に，Create メソッドは呼び出せません。

HiRDB では，?パラメタに対して 1 要素だけの値を使えません。このため，1 を指定すると，実行時に HiRDB のエラーになります。HiRDB 使用時は，Create メソッドでは必ず 2 以上の値を指定してください。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_CANNOT_ACCESS_WHILE_EXECUTED
　DBPreparedStatement オブジェクトの実行中に，Create メソッドを呼び出しました。

DB_ERROR_NOT_ENOUGH_MEMORY
　メモリ容量が不足しています。

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数 i32ArrayCount で範囲外の値を指定しました。

GetArrayCount メソッド

機能

Create メソッドで指定された要素の数を取得します。

形式

```
virtual INT32 GetArrayCount(void) const
```

引数

なし

戻り値

データ型：INT32

要素の数を取得します。

機能詳細

Create メソッドで確保したデータ領域の要素数を返します。

発生する例外

なし

GetData メソッド

機能

指定した番号の要素の値を取得します。

形式

```
virtual BOOLEAN GetData(INT32 i32Index, INT32 * pi32Data)
                                              const throw DBSQLCA
virtual BOOLEAN GetData(INT32 i32Index, INT16 * pi16Data)
                                              const throw DBSQLCA
virtual BOOLEAN GetData(INT32 i32Index, SINGLE * psfData)
                                              const throw DBSQLCA
virtual BOOLEAN GetData(INT32 i32Index, DOUBLE * pdfData)
                                              const throw DBSQLCA
virtual BOOLEAN GetData(INT32 i32Index, LPTSTR * lptData)
                                              const throw DBSQLCA
virtual BOOLEAN GetData(INT32 i32Index, DBR_DATETIME * pdtData)
                                              const throw DBSQLCA
virtual BOOLEAN GetData(INT32 i32Index,
                    DBR_BINARY * pbinData) const throw DBSQLCA
```

引数

i32Index

取得する値の要素番号を，1 からフィールドの最大要素数以下の値で指定します。

* xxData

取得するデータ型の領域を指すポインタを指定します。

戻り値

データ型：BOOLEAN

TRUE：取得しようとした値は欠損値ではありません。

FALSE：取得しようとした値は欠損値です。

機能詳細

指定した要素番号の値を取得します。取得できるデータ型については，SetData メソッドを参照してください。

GetData メソッドは，DBRArrayData オブジェクトに要素の値が設定されている場合に呼び出せます。

日付･時間関連のデータ型の場合にデータを取得するメンバ変数を以下に示します。その他のメンバ変数には 0 が設定されます。

COL_TYPE_DATE：Year, Month, Day, (Sign は 1)

COL_TYPE_TIME：Hour, Minute, Second, (Sign は 1)

COL_TYPE_INTERVAL_YEAR：Year, Month, Day, Sign

COL_TYPE_INTERVAL_HOUR：Hour, Minute, Second, Sign

発生する例外

DBSQLCA(RetCode)

DB_ERROR_CANNOT_ACCESS_WHILE_EXECUTED
　DBPreparedStatement オブジェクトの ExecuteUpdate メソッドを実行中に呼び出しました。

DB_ERROR_ARGUMENT_OUT_OF_RANGE
　引数 i32Index に範囲外の値を指定しています。

DB_ERROR_NO_DATA_AREA
　Create メソッドでデータ領域を生成する前に，このメソッドが呼び出されました。

DB_ERROR_INVALID_ARGUMENT
　メソッドの引数に誤りがあります。

DB_ERROR_CANNOT_BE_NULL
　引数*xxData で NULL を指定しました。

GetDataType メソッド

機能

要素のデータの型を取得します。

形式

```
virtual INT32 GetDataType(void) const
```

引数

なし

戻り値

データ型：INT32

DBRArrayData オブジェクトの保持するデータの型(COL_TYPE_～)を返します。

機能詳細

DBRArrayData オブジェクトの保持するデータの型を返します。DBRArrayDataFactory オブジェクトの CreateArrayData メソッドで指定されたデータのデータ型を返します。

発生する例外

なし

GetPrecision メソッド

機能

ArrayData オブジェクト中の要素の最大長，又は精度を取得します。

形式

```
virtual INT32 GetPrecision(void) const
```

引数

なし

戻り値

データ型：INT32

ArrayData オブジェクト中の要素の最大長（バイト数），又は精度を返します。

機能詳細

要素のデータ型によって，繰り返し列中の要素の最大長（バイト数），又は精度(COL_TYPE_NUMERICの場合だけ)を取得します。表 6-1 にデータ型ごとの戻り値を示します。

表 6-1　データ型ごとの戻り値

データ型	戻り値（バイト数）
COL_TYPE_INT16	2
COL_TYPE_INT32	4
COL_TYPE_SINGLE	4
COL_TYPE_DOUBLE	8
COL_TYPE_NUMERIC	精度として指定した値
COL_TYPE_CHAR	文字列長（バイト数です。文字数ではありません）
COL_TYPE_VARCHAR	文字列長（バイト数です。文字数ではありません）
COL_TYPE_DATE	11
COL_TYPE_TIME	9
COL_TYPE_INTERVAL_YEAR	5
COL_TYPE_INTERVAL_HOUR	4

GetScale メソッド

機能

小数点以下の桁数，又は 1 文字の長さ(バイト数)を取得します。

形式

```
virtual INT32 GetScale(void) const
```

引数

なし

戻り値

データ型：INT32

小数点以下の桁数，又は 1 文字の長さ(バイト数)を返します。

機能詳細

データ型が COL_TYPE_NUMERIC の場合は小数点以下の桁数を，COL_TYPE_CHAR，COL_TYPE_VARCHAR の場合は 1 文字の長さ（バイト数（1 又は 2））を取得します。

発生する例外

なし

SetData メソッド

機能

指定された番号の要素に，値を設定します。

形式

```
virtual void SetData(INT32 i32Index, INT32 i32Data) throw DBSQLCA
virtual void SetData(INT32 i32Index, INT16 i16Data) throw DBSQLCA
virtual void SetData(INT32 i32Index, SINGLE sfData) throw DBSQLCA
virtual void SetData(INT32 i32Index, DOUBLE dfData) throw DBSQLCA
virtual void SetData(INT32 i32Index, LPCTSTR lpctData)
                                                    throw DBSQLCA
virtual void SetData(INT32 i32Index, const DBR_DATETIME&
                                             dtData) throw DBSQLCA
virtual void SetData(INT32 i32Index, const DBR_BINARY & binData)
                                                    throw DBSQLCA
```

引数

i32Index

データを設定する要素番号を，1 からフィールドの要素数以下の値で指定します。

xxData

設定する値を指定します。この引数に指定できるデータ型については，機能詳細の「第 2 引数に指定できるデータの型」を参照してください。

戻り値

なし

機能詳細

ArrayData オブジェクトに，要素の値を設定します。値を設定しなかった要素は，欠損値の扱いになります。

第 2 引数に指定できるデータの型

第 2 引数に指定できるデータの型は，DBRArrayData オブジェクトの生成時に DBRArrayDataFactory オブジェクトの CreateArrayData メソッドで指定した型によって異なります。CreateArrayData メソッドの引数ごとに指定できるデータ型を表 6-2，6-3 に示します。

なお，コピーして作成した更新用の DBRArrayData オブジェクトへ指定できるデータ型は，コピー元のデータ型に依存します。

表 6-2　CreateArrayData メソッドの引数ごとに指定できるデータ型(1)

CreateArrayData の引数	引数の型			
	INT16	INT32	SINGLE	DOUBLE
COL_TYPE_INT16	◎	△	△	△
COL_TYPE_INT32	△	◎	△	△
COL_TYPE_SINGLE	△	△	◎	△
COL_TYPE_DOUBLE	△	△	△	◎
COL_TYPE_NUMERIC	×	×	×	×
COL_TYPE_CHAR	×	×	×	×
COL_TYPE_VARCHAR	×	×	×	×
COL_TYPE_DATE	×	×	×	×
COL_TYPE_TIME	×	×	×	×
COL_TYPE_INTERVAL_YEAR	×	×	×	×
COL_TYPE_INTERVAL_HOUR	×	×	×	×

表 6-3　CreateArrayData メソッドの引数ごとに指定できるデータ型(2)

CreateArrayData の引数	引数の型		
	LPCTSTR	DBR_DATETIME	DBR_BINARY
COL_TYPE_INT16	×	×	×
COL_TYPE_INT32	×	×	×
COL_TYPE_SINGLE	×	×	×
COL_TYPE_DOUBLE	×	×	×
COL_TYPE_NUMERIC	○	×	×
COL_TYPE_CHAR	◎※	×	◎※
COL_TYPE_VARCHAR	◎	×	◎
COL_TYPE_DATE	×	○	×
COL_TYPE_TIME	×	○	×
COL_TYPE_INTERVAL_YEAR	×	○	×

CreateArrayData の引数	引数の型		
	LPCTSTR	DBR_DATETIME	DBR_BINARY
COL_TYPE_INTERVAL_HOUR	×	○	×

(凡例)

◎：そのまま値を取得・設定します。

○：DABroker for C++で DBMS の要求するデータ形式に変換して取得・設定します。

△：値をキャスト(cast)して取得・設定します。

×：例外 DB_ERROR_INVALID_ARGUMENT をスローします。

注※

引数で渡された値が，DBRArrayDataFactory オブジェクトの CreateArrayData メソッドの第 2 引数で指定した最大長に満たない場合，残りの領域はスペース（バイト数は，DBRArrayDataFactory オブジェクトの CreateArrayData メソッドの第 3 引数に依存）で埋められます。

- 引数 lpctData の文字列の長さや binData の長さは，定義した要素長より小さな値で指定してください。
- LPCTSTR 型の引数及び DBR_BINARY 型の Data メンバ変数へ，NULL ポインタは設定できません。
- COL_TYPE_VARCHAR のフィールドに対して長さ 0 の空文字列 ("") (欠損値とは異なります)を設定したい場合は，LPCTSTR 型の引数に対して長さ 0 の空文字列（NULL ではない）を指定するか，DBR_BINARY 型の Length メンバ変数に 0 を設定し，Data メンバ変数に"¥0"を設定してください。

DBR_DATETIME 型を指定する場合の注意

DBR_DATETIME 型の各メンバ変数に対して，以下の制限に反した値を指定した場合，DB_ERROR_TOO_LARGE_DATA のエラーをスローします。値の正当性はチェックしないため，例えば，Hour に対して 50 を設定してもエラーにはなりません。

DBR_DATETIME 型の指定可能な値

Year < 10000, Month < 100, Day < 100, Hour < 100, Minute < 100, Second < 100

設定するデータ型ごとに参照するメンバ変数を以下に示します。そのほかのメンバ変数の値は無視します。

- COL_TYPE_DATE：Year, Month, Day
- COL_TYPE_TIME：Hour, Minute, Second
- COL_TYPE_INTERVAL_YEAR：Year, Month, Day, Sign
- COL_TYPE_INTERVAL_HOUR：Hour, Minute, Second, Sign

DECIMAL 型を指定する場合の注意

DECIMAL 型の値を設定する場合，ユーザが指定した文字列が以下の条件に当てはまる場合は，DB_DRV_ERROR_DATA_CONVERT のエラーをスローします。

- Precision - Scale よりも整数部の桁数が大きい場合。（ただし，左端の 0 は無視します）
- 数字ではない文字が含まれる場合。

（例）DECIMAL(10,3)の場合

"1234567.891" 正常

"11234567.891" エラー（整数部の桁数が大きい）

"001234567.890" 正常（整数部，小数部ともに端の 0 は無視する）

"0001234567.89123" 正常（小数部下 2 桁切り捨て，トレースに警告メッセージ出力）

発生する例外

DBSQLCA(RetCode)

DB_DRV_ERROR_DATA_CONVERT

DECIMAL 型の値を設定するときに，指定した文字列が正しくありません。

DB_ERROR_CANNOT_ACCESS_WHILE_EXECUTED

パラメタを設定した DBPreparedStatement オブジェクトで ExecuteUpdate メソッドを呼び出す前に，SetData メソッドが呼び出されています。

DB_ERROR_INVALID_ARGUMENT

メソッドの引数に誤りがあります。

DB_ERROR_ARGUMENT_OUT_OF_RANGE

指定できない範囲の値が指定されています。

DB_ERROR_TOO_LARGE_DATA

指定したデータの値は大きすぎます。

引数 lpctData，又は binData の Length メンバで指定した長さが，DBRArrayDataFactory オブジェクトの CreateArrayData メソッドの第 2 引数で指定した値より大きな値です。

DB_ERROR_NO_DATA_AREA

Create メソッドでデータ領域を生成する前に，このメソッドを呼び出しています。

DB_ERROR_CANNOT_BE_NULL

LPCTSTR 型の引数，又は DBR_BINARY 型の Data メンバ変数で NULL を設定しました。

SetNull メソッド

機能

要素に欠損値を設定します。

形式

```
virtual void SetNull(INT32 i32Index) throw DBSQLCA
```

引数

i32Index

欠損値として設定する要素の番号を，1 からフィールドの最大要素数以下の値で指定します。

戻り値

なし

機能詳細

引数 i32Index で指定した番号の要素を，欠損値として設定します。

発生する例外

DBSQLCA(RetCode)

DB_ERROR_CANNOT_ACCESS_WHILE_EXECUTED

パラメタを設定した DBPreparedStatement オブジェクトで ExecuteUpdate メソッドを呼び出す前に，SetData メソッドが呼び出されています。

DB_ERROR_ARGUMENT_OUT_OF_RANGE

引数 i32Index に範囲外の値を設定しました。

DB_ERROR_NO_DATA_AREA

Create メソッドでデータ領域を生成する前にこのメソッドが呼び出されました。

6.4　DBRArrayDataPtr クラス

更新を目的とした検索で ResultSet に取得した検索結果のうち，繰り返し列を更新する場合に利用します。

DBRArrayDataPtr クラスは，DBRArrayData クラスのインスタンスを管理するためのスマートポインタクラスです。

このクラスは，参照カウンタの増減や，ヒープ上にある DBRArrayData クラスのインスタンスの解放処理を自動的に行ないます。

機 能	コンストラクタ・デストラクタ名
DBRArrayDataPtr オブジェクトを生成します。	DBRArrayDataPtr コンストラクタ
削除時に呼び出されます。インスタンスを保持していればそのインスタンスの参照カウンタを下げます。	~DBRArrayDataPtr デストラクタ

機 能	ユーザ定義演算子
インスタンスを保持させます。	operator=
DBRArrayData オブジェクトのポインタを返します。	operator->
DBRArrayData オブジェクトのポインタを返します。	operator*

機 能	メソッド名
インスタンスを保持しているかどうかを調べます。	IsNull

DBRArrayDataPtr コンストラクタ

機能

オブジェクトの生成時に呼び出されるコンストラクタです。

形式 1 インスタンスを保持しないオブジェクトを生成する場合

```
DBRArrayDataPtr()
```

形式 2 引数に指定したオブジェクトを生成する場合

```
DBRArrayDataPtr(DBRArrayData* pArray)
```

形式 3 コピーコンストラクタの場合

```
DBRArrayDataPtr(const DBRArrayDataPtr& pArrayPtr)
```

引数

pArray

DBRArrayData オブジェクトのポインタを指定します。

pArrayPtr

DBRArrayDataPtr の参照を指定します。

戻り値

なし

機能詳細

形式 1 の処理

インスタンスを保持しない DBRArrayDataPtr オブジェクトを生成します。

形式 2 の処理

引数 pArray で指定された DBRArrayData オブジェクトを保持します。このとき指定された DBRArrayData オブジェクトに対する参照カウンタをインクリメントします。

形式 3 の処理

コピーコンストラクタです。引数 pArrayPtr で指定された DBRArrayDataPtr オブジェクトが保持している DBRArrayData オブジェクトから，DBRArrayDataPtr オブジェクトを生成します。このとき，指定された DBRArrayDataPtr オブジェクトが保持する DBRArrayData オブジェクトに対する参照カウンタをインクリメントします。

発生する例外

なし

~DBRArrayDataPtr デストラクタ

機能

DBRArrayDataPtr のデストラクタです。

形式

```
~DBRArrayDataPtr
```

引数

なし

戻り値

なし

機能詳細

削除時に呼び出されます。インスタンスを保持していれば, そのインスタンスの参照カウンタをデクリメントします。

発生する例外

なし

operator=

機能

インスタンスを保持させます。

形式 1

```
DBRArrayDataPtr& operator=(DBRArrayData* pArray)
```

形式 2

```
DBRArrayDataPtr& operator=(const DBRArrayDataPtr& pArrayPtr)
```

引数

pArray

保持させたい DBRArrayData オブジェクトを指定します。

pArrayPtr

保持させたいインスタンスを保持している DBRArrayDataPtr オブジェクトを指定します。

戻り値

データ型：DBRArrayDataPtr&

DBRArrayDataPtr オブジェクトの参照を返します。

機能詳細

引数 pArray 又は pArrayPtr で指定された DBRArrayData オブジェクト，又は DBRArrayDataPtr オブジェクトが保持している DBRArrayData オブジェクトから，DBRArrayDataPtr オブジェクトのインスタンスを生成します。このとき，指定された DBRArrayData オブジェクト又は DBRArrayDataPtr オブジェクトが保持する DBRArrayData オブジェクトに対する参照カウンタをインクリメントします。

DBRArrayDataPtr オブジェクトが既にインスタンスを保持している場合は，保持していた DBRArrayData オブジェクトに対する参照カウンタをデクリメントします。

発生する例外

なし

operator->

機能

DBRArrayData オブジェクトのポインタを取得します。

形式

```
DBRArrayData* operator->() const
```

引数

なし

戻り値

データ型：DBRArrayData*

DBRArrayData オブジェクトへのポインタ。

機能詳細

DBRArrayData オブジェクトのポインタを返します。取得したポインタを使って，DBRArrayData オブジェクトのメソッドを呼び出します。

自分自身が DBRArrayData オブジェクトのインスタンスを保持していない場合は, NULL が返ります。しかし，戻り値が NULL の場合，このメソッドを使って DBRArrayData オブジェクトのメソッド呼び出すと異常終了します。インスタンスを保持しているかどうかを調べるには，IsNull メソッドを呼び出してください。

発生する例外

なし

operator*

解説

DBRArrayData オブジェクトのポインタを取得します。

形式

```
DBRArrayData* operator*() const
```

引数

なし

戻り値

データ型：DBRArrayData*

DBRArrayData オブジェクトのポインタ。

機能詳細

DBRArrayData オブジェクトのポインタが返ります。取得したポインタを使って，DBRArrayData オブジェクトのメソッドを呼び出します。

自分自身が DBRArrayData オブジェクトのインスタンスを保持していない場合は, NULL が返ります。しかし，戻り値が NULL の場合，このメソッドを使って DBRArrayData クラスのメソッド呼び出しを行うと異常終了します。インスタンスを保持しているかどうかを調べるには，IsNull メソッドを呼び出してください。

発生する例外

なし

IsNull メソッド

機能

インスタンスを保持しているかどうかを確認します。

形式

```
BOOLEAN IsNull() const
```

引数

なし

戻り値

データ型：BOOLEAN

TRUE：インスタンスを保持していません。

FALSE：インスタンスを保持しています。

機能詳細

DBRArrayDataPtr オブジェクトで DBRArrayData クラスのメソッド呼び出すときに，インスタンスを保持しているかどうかを確認します。

発生する例外

なし

6.5　DBRArrayDataConstPtr クラス

更新を必要としない検索で ResultSet に取得した検索結果のうち，繰り返し列を参照する場合に利用します。

DBRArrayDataConstPtr クラスは，DBRArrayData クラスのインスタンスを管理するためのスマートポインタクラスです。

このクラスは，参照カウンタの増減や，ヒープ上にある DBRArrayData クラスのインスタンスの解放処理を自動的に行ないます。

機 能	コンストラクタ・デストラクタ名
DBRArrayDataConstPtr オブジェクトを生成します。	DBRArrayDataConstPtr コンストラクタ
削除時に呼び出されます。インスタンスを保持していればそのインスタンスの参照カウンタを下げます。	~DBRArrayDataConstPtr デストラクタ

機 能	ユーザ定義演算子
インスタンスを保持させます。	operator=
DBRArrayData オブジェクトのポインタを返します。	operator->
DBRArrayData オブジェクトのポインタを返します。	operator*

機 能	メソッド名
インスタンスを保持しているかどうかを調べます。	IsNull

DBRArrayDataConstPtr コンストラクタ

機能

オブジェクトの生成時に呼び出されるコンストラクタです。

形式 1

```
DBRArrayDataConstPtr()
```

形式 2

```
DBRArrayDataConstPtr(const DBRArrayData* cpArray)
```

形式 3

```
DBRArrayDataConstPtr(const DBRArrayDataConstPtr& cpArrayPtr)
```

形式 4

```
DBRArrayDataConstPtr(const DBRArrayDataPtr& pArrayPtr)
```

引数

cpArray

参照を目的とした DBRArrayData オブジェクトのポインタを指定します。

cpArrayPtr

参照を目的とした DBRArrayDataConstPtr の，参照を指定します。

pArrayPtr

参照を目的とした DBRArrayDataPtr の，参照を指定します。

戻り値

なし

機能詳細

形式 1 の処理

インスタンスを保持しない DBRArrayDataConstPtr オブジェクトを生成します。

形式 2 の処理

引数 cpArray で指定された DBRArrayData オブジェクトを保持します。このとき指定された DBRArrayData オブジェクトに対する参照カウンタをインクリメントします。

形式 3 の処理

コピーコンストラクタです。引数 cpArrayPtr で指定された DBRArrayDataConstPtr オブジェクトが保持している DBRArrayData オブジェクトから，DBRArrayDataConstPtr オブジェクトを生成します。このとき，指定された DBRArrayDataConstPtr オブジェクトが保持する DBRArrayData オブジェクトに対する参照カウンタをインクリメントします。

形式 4 の処理

コピーコンストラクタです。引数 pArrayPtr で指定された DBRArrayDataPtr オブジェクトが保持している DBRArrayData オブジェクトから，DBRArrayDataConstPtr オブジェクトを生成します。このとき，指定された DBRArrayDataPtr オブジェクトが保持する DBRArrayData オブジェクトに対する参照カウンタをインクリメントします。

発生する例外

なし

~DBRArrayDataConstPtr デストラクタ

機能

DBRArrayDataConstPtr のデストラクタです。

形式

```
~DBRArrayDataConstPtr
```

引数

なし

戻り値

なし

機能詳細

削除時に呼び出されます。インスタンスを保持していれば, そのインスタンスの参照カウンタをデクリメントします。

発生する例外

なし

operator=

機能

インスタンスを保持させます。

形式 1

```
DBRArrayDataConstPtr& operator=(const DBRArrayData* cpArray)
```

形式 2

```
DBRArrayDataConstPtr& operator=(const DBRArrayDataConstPtr&
                                                cpArrayPtr)
```

形式 3

```
DBRArrayDataConstPtr& operator=(const DBRArrayDataPtr&
                                                 pArrayPtr)
```

引数

pArray

保持させたい const な DBRArrayData オブジェクトのポインタを指定します。

cpArrayPtr

保持させたいインスタンスを保持している DBRArrayDataConstPtr オブジェクトを指定します。

pArrayPtr

保持させたいインスタンスを保持している DBRArrayDataPtr オブジェクトを指定します。

戻り値

データ型：DBRArrayDataConstPtr&

DBRArrayDataConstPtr オブジェクトの参照を返します。

機能詳細

引数 cpArray，cpArrayPtr，及び pArrayPtr で指定された DBRArrayData オブジェクト，又は DBRArrayDataConstPtr オブジェクト及び DBRArrayDataPtr オブジェクトが保持している DBRArrayData オブジェクトから, DBRArrayDataConstPtr オブジェクトのインスタンスを生成します。このとき，指定された DBRArrayData オブジェクト又は DBRArrayDataConstPtr オブジェクト及び DBRArrayDataPtr オブジェクトが保持する DBRArrayData オブジェクトに対する参照カウンタをインクリメントします。

DBRArrayDataConstPtr オブジェクトが既にインスタンスを保持している場合は，保持していた DBRArrayData オブジェクトに対する参照カウンタをデクリメントします。

発生する例外

なし

operator->

機能

const な DBRArrayData オブジェクトのポインタを取得します。

形式

```
const DBRArrayData* operator->() const
```

引数

なし

戻り値

データ型：const DBRArrayData*

const な DBRArrayData オブジェクトのポインタ。

機能詳細

const な DBRArrayData オブジェクトのポインタを返します。取得したポインタを使って，DBRArrayData オブジェクトのメソッドを呼び出します。なお，戻り値は const なオブジェクトのため，const なメソッドしか呼び出せません。

自分自身が DBRArrayData オブジェクトのインスタンスを保持していない場合は, NULL が返ります。しかし，戻り値が NULL の場合，このメソッドを使って DBRArrayData オブジェクトのメソッドを呼び出すと異常終了します。インスタンスを保持しているかどうかを調べるには，IsNull メソッドを呼び出してください。

発生する例外

なし

operator*

機能

DBRArrayData オブジェクトのポインタを取得します。

形式

```
const DBRArrayData* operator*() const
```

引数

なし

戻り値

データ型：const DBRArrayData*

const な DBRArrayData オブジェクトのポインタ。

機能詳細

const な DBRArrayData オブジェクトのポインタを返します。取得したポインタを使って，DBRArrayData オブジェクトのメソッドを呼び出します。なお，戻り値は const なオブジェクトのため，const なメソッドしか呼び出せません。

自分自身が DBRArrayData オブジェクトのインスタンスを保持していない場合は，NULL が返ります。しかし，戻り値が NULL の場合，このメソッドを使って DBRArrayData オブジェクトのメソッドを呼び出すと異常終了します。インスタンスを保持しているかどうかを調べるには，IsNull メソッドを呼び出してください。

発生する例外

なし

IsNull メソッド

機能

インスタンスを保持しているかどうかを確認します。

形式

```
BOOLEAN IsNull() const
```

引数

なし

戻り値

データ型：BOOLEAN

TRUE：インスタンスを保持していません。

FALSE：インスタンスを保持しています。

機能詳細

DBRArrayDataConstPtr オブジェクトで DBRArrayData クラスのメソッドを呼び出すときに，インスタンスを保持しているかどうかを確認します。

発生する例外

なし

7 データ型

クラスライブラリで扱うデータ型と変換規則について説明します。

7.1　クラスライブラリで扱うデータ型と変換規則

7.1.1　クラスライブラリで使用するデータ型と C++のデータ型との関係

クラスライブラリで使用するデータ型と C++のデータ型との関係を表 7-1 に示します。

表 7-1　クラスライブラリで使用するデータ型と C++のデータ型との関係

クラスライブラリで使用するデータ型	C++のデータ型
short int	short int
long int	long int
DBR_INT8	char
DBR_INT16	short int
DBR_INT32	long int
INT64	long long int
DBR_UINT8	unsigned char
DBR_UINT16	unsigned short int
DBR_UINT32	unsigned long int
UINT64	unsigned long long int
DBR_SINGLE	single float
DBR_DOUBLE	double float
BOOLEAN	boolean
TCHAR	char
LPTSTR	char *
LPCTSTR	const char *

DBR_DATETIME※1	struct DATE_DATA {
	UINT32 Year;
	UINT32 Month;
	UINT32 Day;
	UINT32 Hour;
	UINT32 Minute;
	UINT32 Second;
	UINT32 Microsecond;
	INT16 Sign;
	}

DBR_BINARY	struct BINARY_DATA { UINT32 Length; UINT32 RealLength; LPTSTR Data; }DBR_BINARY;
DBR_RETCODE	UINT32
DBR_SQLCODE	INT32
DBR_SQLSTATE	LPCTSTR

クラスライブラリで使用するデータ型	C++のデータ型
DBR_SQLWARN	LPCTSTR

注※1

DBR_DATETIME 型の Sign メンバは符号を示します。ここには，1 又は-1 が入り，それぞれ，正の値，負の値を示します。データを更新するときに指定する値は，1，-1 のどちらかを指定します。負の値を取らない日付・時間型の場合は Sign メンバの値は無視します。

7.1.2　データ型のサイズと範囲

クラスライブラリでのデータ型に対する OS 別のサイズと範囲を表 7-2 に示します。

表 7-2　データ型のサイズと範囲

クラスライブラリで使用するデータ型	Windows		UNIX	
	サイズ（バイト）	値の範囲	サイズ（バイト）	値の範囲
DBR_INT8	1	-256 ～ 255	1	-256 ～ 255
DBR_INT16	2	-32,768 ～ 32,767	2	-32,768 ～ 32,767
DBR_INT32	4	-2,147,483,648 ～ 2,147,483,647	4	-2,147,483,648 ～ 2,147,483,647
DBR_UINT8	1	0 ～ 512	1	0 ～ 512
DBR_UINT16	2	0 ～ 65,535	2	0 ～ 65,535
DBR_UINT32	4	0 ～ 4,294,967,295	4	0 ～ 4,294,967,295
DBR_SINGLE	4	3.4E-38 ～ 3.4E+38	4	3.4E-38 ～ 3.4E+38
DBR_DOUBLE	8	1.7E-308 ～ 1.7E+308	8	1.7E-308 ～ 1.7E+308
BOOLEAN	システムに依存	TRUE（0 以外）又は FALSE（0）	システムに依存	TRUE（0 以外）又は FALSE（0）
TCHAR	1	-128 ～ 127	1	-128 ～ 127
LPTSTR	4	－	4	－
LPCTSTR	4	－	4	－
DBR_DATETIME	30	－	30	－
DBR_BINARY	－	－	－	－

（凡例）

－：該当なし。

7.1.3　戻り値と DBMS でのデータ型の対応

戻り値と DBMS でのデータ型の対応を次に示します。戻り値とは，次のメソッドを呼び出したときに取得する値を指します。

- classListColumns オブジェクトの Type メソッド
- classListProcedureColumns オブジェクトの Type メソッド，
- DBResultSetMetaData オブジェクトの GetColumnType メソッド

(1)　SQL Anywhere，ORACLE，HiRDB，SQL Server

SQL Anywhere，ORACLE，HiRDB，SQL Server の場合の，戻り値と DBMS でのデータ型の対応を表 7-3 に示します。

表 7-3　データ型対応表

クラスライブラリから返る値	DBMS の種類			
	SQL Anywhere	ORACLE	HiRDB	SQL Server
COL_TYPE_INT16	smallint bit 又は oldbit tinyint	－	smallint	smallint tinyint
COL_TYPE_INT32	int integer	－	int	int
COL_TYPE_NUMERIC	decimal numeric money smallmoney	number integer binary_integer float	decimal	decimal numeric money smallmoney
COL_TYPE_SINGLE	real(float)※1	－	smallflt	real(float)
COL_TYPE_DOUBLE	double(float)※1	－	float	float
COL_TYPE_CHAR	－	char	char mchar nchar	char
COL_TYPE_VARCHAR	char character varchar character varying sysname	varchar varchar2	varchar mvarchar nvarchar	sysname(SQL Server6.5 のとき) varchar
COL_TYPE_LONGVARCHAR	long varchar text	long	－	text
COL_TYPE_DATE	date	－	date	－
COL_TYPE_TIME	time	－	time	－
COL_TYPE_TIMESTAMP	datetime smalldatetime timestamp	date	－	datetime smalldatetime
COL_TYPE_INTERVAL_YEAR	－	－	interval year to day	－
COL_TYPE_INTERVAL_HOUR	－	－	interval hour to	－

クラスライブラリから返る値	DBMS の種類			
	SQL Anywhere	ORACLE	HiRDB	SQL Server
COL_TYPE_INTERVAL_HOUR	－	－	second	－
COL_TYPE_BINARY	binary varbinary	－	－	binary timestamp
COL_TYPE_VARBINARY	－	raw	－	varbinary
COL_TYPE_LONGVARBINARY	long binary image java serialization java.lang.Object	long raw	binary※2 blob	image
COL_TYPE_ROWID	－	rowid	－	－
COL_TYPE_BIT	bit	－	－	bit
COL_TYPE_BIGINT	bigint	－	－	－

（凡例）

－：対応するデータ型はありません。

注　SQL Anywhere の列には，SQL Anywhere 及び Adaptive Server Anywhere の両方のデータ型を含んでいます。

注※1　float 型が real 型とみなされるか double 型とみなされるかは定義内容によります。

注※2　HiRDB の binary 型はバージョン 02-12 以降で使用できます。

(2)　メインフレーム系データベース

メインフレーム系データベースの場合の，戻り値と DBMS でのデータ型の対応を表 7-4 に示します。

表 7-4　データ型対応表（メインフレーム系データベース）

クラスライブラリから返る値	XDM/RD	XDM/SD
COL_TYPE_INT16	smallint	smallint
COL_TYPE_INT32	integer	integer
COL_TYPE_NUMERIC	numeric trailing numeric unsigned decimal large decimal	numeric trailing decimal
COL_TYPE_SINGLE	smallflt	－
COL_TYPE_DOUBLE	float	－
COL_TYPE_CHAR	char mchar nchar xchar	char nchar

クラスライブラリから返る値	XDM/RD	XDM/SD
COL_TYPE_VARCHAR	varchar mvarchar nvarchar	varchar mvarchar nvarchar
COL_TYPE_LONGVARCHAR	long varchar long mvarchar long nvarchar	long varchar long mvarchar long nvarchar
COL_TYPE_DATE	date	date
COL_TYPE_BINARY	−	bit $dbk
COL_TYPE_ROWID	rowid	rowid

（凡例）

−：対応するデータ型はありません。

(3)　SQL/K

SQL/K の場合の，戻り値と DBMS でのデータ型の対応を表 7-5 に示します。

戻り値とは，次のメソッドを呼び出したときに取得する値を指します。

- classListColumns オブジェクトの Type メソッド
- DBResultSetMetaData オブジェクトの GetColumnType メソッド，
- DBRResultSet オブジェクトの GetFieldType メソッド

表 7-5　データ型対応表（SQL/K）

クラスライブラリから返る値	SQL/K
COL_TYPE_INT16	SMALLINT
COL_TYPE_INT32	INTEGER
COL_TYPE_NUMERIC	NUMERIC TRAILING NUMERIC UNSIGNED DECIMAL
COL_TYPE_CHAR	CHAR NCHAR MCHAR
COL_TYPE_BINARY	XCHAR

7.1.4　C++と DBMS のデータ型の対応

C++と DBMS のデータ型の対応を表 7-6 に示します。

表 7-6　C++のデータ型と DBMS のデータ型の関係

Ctype	Type
COL_CTYPE_INT16	COL_TYPE_INT16
	COL_TYPE_BIT
COL_CTYPE_INT32	COL_TYPE_INT32
	COL_TYPE_SERIAL
COL_CTYPE_UINT16	COL_TYPE_UINT16
COL_CTYPE_UINT32	COL_TYPE_UINT32
COL_CTYPE_SINGLE	COL_TYPE_SINGLE
COL_CTYPE_DOUBLE	COL_TYPE_DOUBLE
COL_CTYPE_CHAR	COL_TYPE_CHAR
	COL_TYPE_VARCHAR
	COL_TYPE_LONGVARCHAR
	COL_TYPE_NUMERIC
	COL_TYPE_MONEY
	COL_TYPE_ROWID
	COL_TYPE_BIGINT
COL_CTYPE_BINARY	COL_TYPE_BINARY
	COL_TYPE_VARBINARY
	COL_TYPE_LONGVARBINARY
COL_CTYPE_DATETIME	COL_TYPE_DATE
	COL_TYPE_TIME
	COL_TYPE_TIMESTAMP
	COL_TYPE_INTERVAL_YEAR
	COL_TYPE_INTERVAL_HOUR

7.1.5　データ型変換規則

(1)　GetField，GetParam メソッド

DBResultSet クラスの GetField メソッド,DBPreparedStatement，DBCallableStatement クラスの GetParam メソッドで取得するデータの，データ型変換規則を表 7-7，及び表 7-8 に示します。

表 7-7　GetField，GetParam メソッドでのデータ型変換規則(1)

DBMS データ型	取得するデータ型			
	INT16	INT32	UINT16	UINT32
COL_CTYPE_INT16	(B)	(A)	(C)	(A)
COL_CTYPE_INT32	(B)	○	(C)	(A)
COL_CTYPE_UINT16	(B)	(A)	(C)	(A)
COL_CTYPE_UINT32	(B)	(A)	(C)	○
COL_CTYPE_SINGLE	(B)	(A)	(C)	(A)
COL_CTYPE_DOUBLE	(B)	(A)	(C)	(A)
COL_CTYPE_BOOLEAN	(B)	(A)	(C)	(A)
COL_CTYPE_CHAR	f(atol)[B]	f(atol)[A]	f(strtoul) [C]	f(strtoul) [A]
COL_CTYPE_DATETIME	×	×	×	×
COL_CTYPE_BINARY	×	×	×	×

表 7-8　GetField，GetParam メソッドでのデータ型変換規則(2)

DBMS データ型	取得するデータ型					
	SINGLE	DOUBLE	BOOLEAN	LPTSTR	DATE	BINARY
COL_CTYPE_INT16	(D)	(A)	(E)	H：f(ltoa) AL：f(sprintf) W：f(_ltoa)	×	×
COL_CTYPE_INT32	(D)	(A)	(E)	H：f(ltoa) AL：f(sprintf) W：f(_ltoa)	×	×
COL_CTYPE_UINT16	(D)	(A)	(E)	H：f(ultoa) AL：f(sprintf) W：f(_ultoa)	×	×
COL_CTYPE_UINT32	(D)	(A)	(E)	H：f(ultoa) AL：f(sprintf) W：f(_ultoa)	×	×
COL_CTYPE_SINGLE	(D)	(A)	(E)	H：f(gcvt) AL：f(gcvt) W：f(_gcvt)	×	×
COL_CTYPE_DOUBLE	(D)	○	(E)	H：f(gcvt) AL：f(gcvt) W：f(_gcvt)	×	×
COL_CTYPE_BOOLEAN	(D)	(A)	(A)	※1	×	×

DBMS データ型	取得するデータ型					
	SINGLE	DOUBLE	BOOLEAN	LPTSTR	DATE	BINARY
COL_CTYPE_CHAR	f(atof)[D]	f(atof) [A]	f(atof) [E]	○	×	※2
COL_CTYPE_DATETIME	×	×	×	×	○	×
COL_CTYPE_BINARY	×	×	×	×	×	○

（凡例）

○：変換しないで取得します。

×：取得できません（DB_ERROR_DATA_TRUNCATED エラーをスローします）。

f(FUNC)[cast type]：FUNC（Cランタイム関数）で変換した後，cast type の方法でキャストします。

(cast type)：cast type の方法でキャストします。

cast type

A：取得データ型にキャスト

B：(INT16)(INT32)Data (INT32 にキャストし,その後 INT16 にキャストします)

C：(UINT16)(UINT32)Data

D：(SINGLE)(DOUBLE)Data

E：(BOOLEAN)(INT32)Data

H：HP-UX の場合

AL：AIX および Red Hat Linux の場合

W：Windows の場合

注※1　TRUE なら文字列"TRUE"，FALSE なら文字列"FALSE"

注※2　COL_TYPE_CHAR，COL_TYPE_VARCHAR の場合だけ取得できます。
その他の場合は,DB_ERROR_DATA_TRUNCATED エラーをスローします。

(2)　SetParam メソッド

DBCallableStatement,DBPreparedStatement クラスの SetParam メソッドで指定したデータがどのように変換されて，実際に設定されるのか，そのデータ型変換規則を表 7-9 に示します。

表 7-9　SetParam メソッドでのデータ型変換規則

設定データ型	変換規則
INT16	そのまま設定
UINT16	INT16 にキャストして設定
INT32	そのまま設定
UINT32	INT32 にキャストして設定
SINGLE	そのまま設定

設定データ型	変換規則
DOUBLE	そのまま設定
BOOLEAN	INT16 にキャストして設定
LPCTSTR	そのまま設定(ただし,NULL ポインタの場合は欠損値を設定)
DBR_DATETIME	DBMS の内部データ形式に変換して設定
DBR_BINARY	そのまま設定

! 注意事項

「そのまま設定」とあるものは,DBMS に対して指定されたままのデータ型で値を渡します。そのため指定された値が有効かどうかは DBMS に依存します。例えば,ORACLE の NUMBER 型に対しては LPCTSTR 型でも数字列を指定すれば ORACLE がデータ型を変換するため値を設定できますが,HiRDB の integer 型に対して LPCTSTR 型で数字列を指定しても HiRDB では文字列型は数値型に変換しないためエラーとなります。データを設定できるデータ型については,各 DBMS のマニュアルを参照してください。

(3)　SetField メソッド

DBResultSet クラスの SetField メソッドで設定するデータの，データ型変換規則を表 7-10，及び表 7-11 に示します。

表 7-10　SetField メソッドでのデータ型変換規則(1)

DBMS データ型	設定するデータ型			
	INT16	INT32	UINT16	UINT32
COL_CTYPE_INT16	○	(INT16)	(INT16)	(INT16)
COL_CTYPE_INT32	(INT32)	○	(INT32)	(INT32)
COL_CTYPE_UINT16	※1	※1	※1	※1
COL_CTYPE_UINT32	※1	※1	※1	※1
COL_CTYPE_SINGLE	(SINGLE)	(SINGLE)	(SINGLE)	(SINGLE)
COL_CTYPE_DOUBLE	(DOUBLE)	(DOUBLE)	(DOUBLE)	(DOUBLE)
COL_CTYPE_BOOLEAN	※1	※1	※1	※1
COL_CTYPE_CHAR	H：f(ltoa) AL：f(sprintf) W：f(_ltoa)	H：f(ltoa) AL：f(sprintf) W：f(_ltoa)	H：f(ultoa) AL：f(sprintf) W：f(_ultoa)	H：f(ultoa) AL：f(sprintf) W：f(_ultoa)
COL_CTYPE_DATETIME	×	×	×	×
COL_CTYPE_BINARY	×	×	×	×

表 7-11　SetField メソッドでのデータ型変換規則(2)

DBMS データ型	設定するデータ型					
	SINGLE	DOUBLE	BOOLEAN	LPTSTR	DATE	BINARY
COL_CTYPE _INT16	(INT16)	(INT16)	(INT16)	(INT16) f(atol)	×	×
COL_CTYPE _INT32	(INT32)	(INT32)	(INT32)	(INT32) f(atol)	×	×
COL_CTYPE _UINT16	※1	※1	※1	※1	※1	※1
COL_CTYPE _UINT32	※1	※1	※1	※1	※1	※1
COL_CTYPE _SINGLE	○	(SINGLE)	(SINGLE)	(SINGLE) f(atof)	×	×
COL_CTYPE _DOUBLE	(DOUBLE)	○	(DOUBLE)	(DOUBLE) f(atof)	×	×
COL_CTYPE _BOOLEAN	※1	※1	※1	※1	※1	※1
COL_CTYPE _CHAR	H：f(gcvt) AL：f(gcvt) W：f(_gcvt)	H:：f(gcvt) AL：f(gcvt) W：f(_gcvt)	※2	○	×	※3
COL_CTYPE _DATETIME	×	×	×	×	○※4	×
COL_CTYPE _BINARY	×	×	×	×	×	○

(凡例)

○：変換しないで設定します。

×：設定できません(DB_ERROR_DATA_TRUNCATED エラーをスローします)。

(data type)：data type にキャストして設定します。

f(func)：func 関数で変換して設定します。

(data type)f(func)：func 関数で変換した後,data type でキャストして設定します。

H：HP-UX の場合

AL：AIX および Red Hat Linux の場合

W：Windows の場合

注※1　現在のバージョンでは，形式 COL_CTYPE_UINT16,及び COL_CTYPE_UINT32, COL_CTYPE_BOOLEAN はありません。

注※2　TRUE なら文字列"TRUE"を，FALSE なら文字列"FALSE"を設定します。

注※3　COL_TYPE_CHAR，COL_TYPE_VARCHAR に対してだけ設定できます。

注※4　DBMS の内部データ形式に変換します。

7.1.6　DBMS のデータ型と識別子との対応

DBMS 固有の，データ型と識別子の対応を表 7-12 から 7-19 に示します。

表 7-12　DBMS のデータ型と識別子との対応（ORACLE）

ORACLE のデータ型	識別子
VARCHAR2	OR_DT_VARCHAR2
NUMBER	OR_DT_NUMBER
BINARY_INTEGER	OR_DT_BINARY_INTEGER
FLOAT	OR_DT_FLOAT
LONG	OR_DT_LONG
ROWID	OR_DT_ROWID
DATE	OR_DT_DATE
RAW	OR_DT_RAW
LONG RAW	OR_DT_LONG_RAW
CHAR	OR_DT_CHAR

表 7-13　DBMS のデータ型と識別子との対応（SQL Anywhere）

SQL Anywhere のデータ型	識別子
CHAR CHARACTER VARCHAR CHARACTER VARYING SYSNAME	ODBC_DT_VARCHAR
LONG VARCHAR TEXT	ODBC_DT_LONG_VARCHAR
DECIMAL MONEY SMALLMONEY NUMERIC	ODBC_DT_NUMERIC
TINYINT	ODBC_DT_TINYINT
SMALLINT	ODBC_DT_SMALLINT
INT INTEGER	ODBC_DT_INTEGER
REAL FLOAT	ODBC_DT_REAL
DOUBLE	ODBC_DT_DOUBLE
DATE	ODBC_DT_DATE

SQL Anywhere のデータ型	識別子
TIME	ODBC_DT_TIME
DATETIME SMALLDATETIME TIMESTAMP	ODBC_DT_TIMESTAMP
BINARY	ODBC_DT_BINARY
LONG BINARY IMAGE	ODBC_DT_LONG_VARBINARY

表 7-14　DBMS のデータ型と識別子との対応（Adaptive Server Anywhere）

DBMS のデータ型	識別子
CHAR SYSNAME VARCHAR	ODBC_DT_VARCHAR
LONG VARCHAR TEXT	ODBC_DT_LONG_VARCHAR
DECIMAL	ODBC_DT_DECIMAL
MONEY NUMERIC SMALLMONEY	ODBC_DT_NUMERIC
OLDBIT TINYINT	ODBC_DT_TINYINT
SMALLINT	ODBC_DT_SMALLINT
INTEGER	ODBC_DT_INTEGER
FLOAT	ODBC_DT_REAL
DATE	ODBC_DT_DATE
TIME	ODBC_TIME
DOUBLE FLOAT	ODBC_DT_DOUBLE
DATETIME SMALLDATETIME TIMESTAMP	ODBC_DT_TIMESTAMP
BINARY VARBINARY	ODBC_DT_BINARY
IMAGE Java serialization java.lang.Object LONG BINARY	ODBC_DT_LONG_VARBINARY

DBMS のデータ型	識別子
BIT	ODBC_DT_BIT
BIGINT	ODBC_DT_BIGINT

表 7-15　DBMS のデータ型と識別子との対応（SQL Server）

DBMS のデータ型	識別子
CHAR	ODBC_DT_CHAR
SYSNAME(SQL Server6.5 のとき) VARCHAR	ODBC_DT_VARCHAR
TEXT	ODBC_DT_LONG_VARCHAR
DECIMAL MONEY SMALLMONEY	ODBC_DT_DECIMAL
NUMERIC	ODBC_DT_NUMERIC
TINYINT	ODBC_DT_TINYINT
SMALLINT	ODBC_DT_SMALLINT
INT	ODBC_DT_INTEGER
FLOAT REAL	ODBC_DT_REAL
FLOAT	ODBC_DT_DOUBLE
DATETIME SMALLDATETIME	ODBC_DT_TIMESTAMP
BINARY TIMESTAMP	ODBC_DT_BINARY
VARBINARY	ODBC_DT_VARBINARY
IMAGE	ODBC_DT_LONG_VARBINARY
BIT	ODBC_DT_BIT

表 7-16　DBMS のデータ型と識別子との対応（HiRDB）

HiRDB のデータ型	識別子※1
INTERVAL YEAR TO DAY	HI_DT_INTER_YEAR
INTERVAL HOUR TO SECOND	HI_DT_INTER_HOUR
DATE	HI_DT_DATE
TIME	HI_DT_TIME
BLOB	HI_DT_BLOB

HiRDB のデータ型	識別子※1
BINARY	HI_DT_BINARY※2
MVARCHAR	HI_DT_MVARCHAR
MCHAR	HI_DT_MCHAR
NVARCHAR	HI_DT_NVARCHAR
NCHAR	HI_DT_NCHAR
VARCHAR	HI_DT_VARCHAR
CHAR	HI_DT_CHAR
FLOAT	HI_DT_DOUBLE
SMALLFLT	HI_DT_FLOAT
DEC	HI_DT_DECIMAL
INT	HI_DT_INT
SMALLINT	HI_DT_SMALLINT

注※1　欠損値を許さないフィールドの場合です。欠損値を許すフィールドの場合は，"HI_DT_DATE_N"のように識別子の末尾に"_N"を付けた値になります。

注※2　HiRDB の BINARY 型はバージョン 02-12 以降で使用できます。

表 7-17　DBMS のデータ型と識別子との対応（XDM/RD）

XDM/RD のデータ型	識別子※
XCHAR	RDA_DT_XCHAR
NUMERIC TRAILING	RDA_DT_NUM_TRAILING
NUMERIC UNSIGNED	RDA_DT_NUM_UNSIGNED
FLOAT	RDA_DT_FLOAT
SMALLFLT	RDA_DT_SMALLFLT
DECIMAL	RDA_DT_DECIMAL
LARGE DECIMAL	RDA_DT_LARGE_DECIMAL
INTEGER	RDA_DT_INTEGER
SMALLINT	RDA_DT_SMALLINT
MVARCHAR	RDA_DT_MVARCHAR
MCHAR	RDA_DT_MCHAR
LONG MVARCHAR	RDA_DT_LONG_MVARCHAR
NVARCHAR	RDA_DT_NVARCHAR
NCHAR	RDA_DT_NCHAR
LONG NVARCHAR	RDA_DT_LONG_NVARCHAR

XDM/RD のデータ型	識別子※
VARCHAR	RDA_DT_VARCHAR
CHAR	RDA_DT_CHAR
LONG VARCHAR	RDA_DT_LONG_VARCHAR
DATE	RDA_DT_DATE

注※　欠損値を許さないフィールドの場合です。欠損値を許すフィールドの場合は，"RDA_DT_FLOAT_N"のように識別子の末尾に"_N"を付けた値になります。

表 7-18　DBMS のデータ型と識別子との対応（SQL/K）

SQL/K のデータ型	識別子
SMALLINT	DBS_DT_SMALLINT
INTEGER	DBS_DT_INTEGER
DECIMAL	DBS_DT_DECIMAL
NUMERIC TRAILING	DBS_DT_NUM_TRAILING
NUMERIC UNSIGNED	DBS_DT_NUM_UNSIGNED
CHAR	DBS_DT_CHAR
NCHAR	DBS_DT_NCHAR
MCHAR	DBS_DT_MCHAR
XCHAR	DBS_DT_XCHAR

表 7-19　DBMS のデータ型と識別子との対応（XDM/SD）

XDM/SD のデータ型	識別子
BIT	DBS_DT_BIT
DBK(データベースキー)	DBS_DT_DBK
NUMERIC TRAILING	DBS_DT_NUM_TRAILING
DECIMAL	DBS_DT_DECIMAL
INTEGER	DBS_DT_INTEGER
SMALLINT	DBS_DT_SMALLINT
NCHAR	DBS_DT_NCHAR
CHAR	DBS_DT_CHAR

7.2　DBR_BINARY 型を使用した VARCHAR データの取得方法

DBResultSet クラスの GetField メソッドを使用して，VARCHAR 型のデータを DBR_BINARY 型で取得する方法について説明します。ここでいう VARCHAR 型とは，DABroker for C++の型識別子 COL_TYPE_VARCHAR に該当するデータ型のことを指します。COL_TYPE_VARCHAR と各 DBMS のデータ型の対応については，「戻り値と DBMS でのデータ型の対応」を参照してください。

7.2.1　DBR_BINARY 型の構造体とメンバに設定される値

DBR_BINARY 型は次に示すような構造体になっています。

```
typedef struct {
  DBR_UINT32  Length;
  DBR_UINT32  RealLength;
  LPTSTR      Data;
} DBR_BINARY;
```

それぞれのメンバは，SetResultSetType メソッドや GetField メソッド，SetMaxFieldSize メソッドの指定によって次に示す値が設定されます。簡易版クラスを使用する場合は，次の説明中の SetResultSetType メソッドを Execute メソッドと読み替えてください。また，簡易版には SetMaxFieldSize メソッドがありませんので，「7.2.1(1)　メンバに設定される値(SetMaxFieldSize メソッドの指定なし)」だけを参照してください。

(1)　メンバに設定される値(SetMaxFieldSize メソッドの指定なし)

GetField メソッドの形式 2 を使用した場合，又は形式 3 を使用してデータの切り捨てが発生しなかった場合にメンバに設定される値を表 7-20 に，GetField メソッドの形式 3 を使用してデータの切り捨てが発生した場合にメンバに設定される値を表 7-21 に示します。

VARCHAR_LENGTH_DEF を指定した場合は，ユーザが指定した領域長とカラムの定義長を比較して切り捨てたかどうか判定されます。そのため，データ長よりもユーザの指定した領域長の方が大きく，実際にはデータの切り捨てが発生していない場合でも，ユーザの指定した領域長が定義長より小さければ切り捨てが発生したと見なされ，Length メンバや RealLength メンバには表 7-21 の値が入ります。

VARCHAR_LENGTH_REAL を指定した場合は，ユーザが指定した領域長と実際のデータ長を比較して切り捨てたかどうか判定されます。

表 7-20　GetField メソッドの形式 2 を使用した場合，又は形式 3 を使用してデータの切り捨てが発生しなかった場合にメンバに設定される値(SetMaxFieldSize メソッドの指定なし)

メンバ	SetResultSetType の指定値	
	VARCHAR_LENGTH_DEF	VARCHAR_LENGTH_REAL
Length	カラムの定義長	実際のデータ長
RealLength	0	0
Data※	データを指すポインタのコピー(形式 2)，又はデータのコピー(形式 3)	データを指すポインタのコピー(形式 2)，又はデータのコピー(形式 3)

注※　形式 2 の場合，Data メンバにはデータを指すポインタがコピーされます。形式 3 の場合，Data メンバにはユーザが確保した領域のポインタが設定されているため，その領域に有効なデータがコピーされます。

表 7-21　GetField メソッドの形式 3 を使用してデータの切り捨てが発生した場合にメンバに設定される値(SetMaxFieldSize メソッドの指定なし)

メンバ	SetResultSetType の指定値	
	VARCHAR_LENGTH_DEF	VARCHAR_LENGTH_REAL
Length	切り捨て後の有効データ長	切り捨て後の有効データ長
RealLength	カラムの定義長	実際のデータ長
Data※	切り捨て後の有効データのコピー	切り捨て後の有効データのコピー

注※　形式 2 の場合，Data メンバにはデータを指すポインタがコピーされます。形式 3 の場合，Data メンバにはユーザが確保した領域のポインタが設定されているため，その領域に有効なデータがコピーされます。

(2)　メンバに設定される値(SetMaxFieldSize メソッドの指定あり)

GetField メソッドの形式 2 を使用した場合，又は形式 3 を使用してデータの切り捨てが発生しなかった場合にメンバに設定される値を表 7-22 に，GetField メソッドの形式 3 を使用してデータの切り捨てが発生した場合にメンバに設定される値を表 7-23 に示します。

SetMaxFieldSize で取得するデータの最大長を指定した場合，もし実際のデータが SetMaxFieldSize で指定した長さよりも大きいとしても，SetMaxFieldSize で指定した長さ分しかデータは取得されません。このような場合，実際のデータ長を取得することはできません。

VARCHAR_LENGTH_DEF を指定した場合は，ユーザが指定した領域長と SetMaxFieldSize メソッドで指定した長さを比較して切り捨てたかどうか判定されます。このため，データ長よりもユーザの指定した領域長の方が大きく，実際にはデータの切り捨てが発生していない場合でも，ユーザの指定した領域長が SetMaxFieldSize で指定した長さよりも小さければ切り捨てが発生したと見なされ，Length メンバや RealLength メンバには表 7-23 の値が入ります。

VARCHAR_LENGTH_REAL を指定した場合は，ユーザが指定した領域長と実際のデータ長を比較して切り捨てたかどうか判定されます。

表 7-22　GetField メソッドの形式 2 を使用した場合，又は形式 3 を使用してもデータの切り捨てが発生しなかった場合にメンバに設定される値(SetMaxFieldSize メソッドの指定あり)

メンバ	SetResultSetType の指定値	
	VARCHAR_LENGTH_DEF	VARCHAR_LENGTH_REAL
Length	SetMaxFieldSize の指定長	取得データ長※2
RealLength	0	0
Data※1	データを指すポインタのコピー(形式 2)，又はデータのコピー(形式 3)	データを指すポインタのコピー(形式 2)，又はデータのコピー(形式 3)

注※1　形式 2 の場合，Data メンバにはデータを指すポインタがコピーされます。形式 3 の場合，Data メンバにはユーザが確保した領域のポインタが設定されているため，その領域に有効なデータがコピーされます。

注※2　実際のデータ長が SetMaxFieldSize で指定した長さよりも大きい場合は，SetMaxFieldSize で指定した長さ分しかデータを取得されないため，ここは実際のデータ長ではなく，取得データ長となります。

表 7-23　GetField メソッドの形式 3 を使用してデータの切り捨てが発生した場合にメンバに設定される値(SetMaxFieldSize メソッドの指定あり)

メンバ	SetResultSetType の指定値	
	VARCHAR_LENGTH_DEF	VARCHAR_LENGTH_REAL
Length	切り捨て後の有効データ長	切り捨て後の有効データ長
RealLength	SetMaxFieldSize の指定長	取得データ長※2
Data※1	切り捨て後の有効データのコピー	切り捨て後の有効データのコピー

注※1　形式 2 の場合, Data メンバにはデータを指すポインタがコピーされます。形式 3 の場合, Data メンバにはユーザが確保した領域のポインタが設定されているため，その領域に有効なデータがコピーされます。

注※2　実際のデータ長が SetMaxFieldSize で指定した長さよりも大きい場合は，SetMaxFieldSize で指定した長さ分しかデータを取得されないため，ここは実際のデータ長ではなく，取得データ長となります。

8 トラブルシューティング

DABroker の使用中に何らかのエラーを生じることがあります。ここでは，エラー発生時にどのような手順で，どのように解決すればよいかを説明します。

8.1　手順

この節では，トラブルシューティングの手順と，トレースファイルの設定方法,及び見方について説明します。

8.1.1　トラブルシューティングについて

(1)　障害発生時の動作

障害発生時の動作は，次のように分けられます。

- C++クラスライブラリを利用しているアプリケーションが, アプリケーションエラーによって異常終了した場合。
  アプリケーションプログラムと共にアボートします。アプリケーションを開始したトランザクションについては，DBMS によってロールバックされます。
- DBMS 自身が異常終了した場合。
  DABroker for C++は，DBMS との接続が切れたことをアプリケーションにエラーとして通知します。アプリケーションのその後の動作は，アプリケーションの実装に依存します。アプリケーションが開始したトランザクションは無効になります。回復は DBMS に依存します。

(2)　トラブルシューティングの流れ

障害が発生するとエラーメッセージが出力されます。これをもとに障害の原因を取り除いてください。C++クラスライブラリで発生するエラー詳細については，「8.2 C++クラスライブラリのエラー情報」を参照してください。

また，DABroker for C++では必要に応じてトレースログも採取できます。このトレースログを基にエラーがどの時点で起こったかを調査できます。

次に，UNIX の場合のトレースログ採取方法を説明します。Windows の場合は環境設定ユティリティを使って採取方法を設定できます。Windows の場合の設定方法については，マニュアル「DABroker」を参照してください。

8.1.2　トレースログ採取のための設定

プログラムの障害調査及びプログラムテストのため，トレースログを採取できます。DABroker では，環境変数によって採取する情報のレベルを設定できるため, 実行時間と取得できる情報量を状況に応じて調整できます。

トレースログを採取するための設定は，conf ディレクトリの環境定義ファイル dasysconf で行います。

dasysconf ファイルの設定例を次に示します。

```
# DABroker Event Trace
# DABroker for C++
  DABCPP_EVTTRC_SIZE=256   # イベントトレースサイズ
  DABCPP_EVTTRC_LEVEL=1    # 取得レベル
  DABCPP_EVTTRC_00=ON   # 事象種別   20thビット 0x00000010
  DABCPP_EVTTRC_01=OFF
  DABCPP_EVTTRC_02=ON
  DABCPP_EVTTRC_03=OFF
```

- DABCPP_EVTTRC_SIZE
  採取するトレースログのファイルサイズを KB 単位で指定します。標準値は 32KB です。

-1：トレースログを採取しない
0：ファイルサイズに制限なし
10～32767：出力ファイルサイズを 10～32767[KB]の範囲で指定する。

- DABCPP_EVTTRC_LEVEL
  トレースログ出力レベルを指定します。
  0：エラー発生時にエラーの内容を出力します。それ以外の情報は出力しません。
  DABCPP_EVTTRC_00 が OFF のときは何も出力しません。
  1：C++インタフェース部
- DABCPP_EVTTRC_00
  エラー発生時に，エラーの内容を出力するかどうかを指定します。
  ON：出力する
  OFF：出力しない
- DABCPP_EVTTRC_01
  現在使用していません。ON でも OFF でも出力情報には変更ありません。
- DABCPP_EVTTRC_02
  関数の最初と最後で出力するかどうかを指定します。
  ON：出力する
  OFF：出力しない
- DABCPP_EVTTRC_03
  DABroker for C++の障害調査情報を出力します。
  ON：出力する
  OFF：出力しない

8.1.3　トレースファイルの見方

トレースファイルは，"dabevttrccpp1"の名称で作成されます。

出力されるファイルの内容は,次のとおりです。

```
10800041 00100000 711a 00e2 1998/04/04 13:31:52.443000
BEGIN:::Object(12fef8):::DBDriverManager::InitializeMessage()
```

１ブロック目の数値(10800041)：トレース出力に使用したレベル

２ブロック目の数値(00100000)：仮想プロセスＩＤ(関係ありません)

３ブロック目の数値(711a)：プロセスＩＤ

４ブロック目の数値(00e2)：スレッドＩＤ

５ブロック目の数値(1998/04/04)：採取した日付

６ブロック目の数値(13:31:52)：採取した時間

以降のブロック：実行時の動作結果のログ

なお，一つ目のログファイルが一杯になると（DABCPP_EVTTRC_SIZE で指定したサイズを超えると），バックアップファイルとして 1 世代前のファイルを dabevttrccpp2 という名称で作成します。
dabevttrccpp1 が一杯になると dabevttrccpp2 にコピーし，次に dabevttrccpp1 が一杯になると

dabevttrccpp1 を dabevttrccpp2 にもう一度コピーした上で dabevttrccpp1 をラップアラウンドします。

8.2　C++クラスライブラリのエラー情報

この節では，C++クラスライブラリで発生するエラー情報を示します。

8.2.1　C++クラスライブラリで発生するエラー情報

C++クラスライブラリで発生するエラーのエラーコードとそれに対応するメッセージテキストを表 8-1 に示します。

表 8-1　エラーコードとそれに対応するメッセージテキスト

エラーコード	メッセージテキスト
DB_DRV_ERROR_AT_NOT_PARAM	動的に SQL を実行するためのパラメタが指定されていません。
DB_DRV_ERROR_CANNOT_FILE_ACCESS	接続している DBMS ではファイルアクセスできません。
DB_DRV_ERROR_CONNECT_HANDLE	コネクトハンドルが取得できませんでした。
DB_DRV_ERROR_CONNECT_NAME_ALREADY	指定された接続名で既に接続されています。
DB_DRV_ERROR_CURSORID	カーソル ID の取得ができません。カーソル ID（1～64）がすべて使用中です。
DB_DRV_ERROR_CURSOR_COUNT	カーソルの個数が上限値を超えたためオープンできません。
DB_DRV_ERROR_CURSOR_NOT_DECLARE	カーソルが定義されていません。
DB_DRV_ERROR_CURSOR_NOT_OPEN	カーソルがオープンされていません。
DB_DRV_ERROR_DATA_CONVERT	データ変換でエラーが発生しました。
DB_DRV_ERROR_DBNAME_NULL	データベース名が指定されていません。
DB_DRV_ERROR_DECIMAL_TYPE	DECIMAL データは扱うことができません。
DB_DRV_ERROR_EXIST_OTHER_TYPE_DATA	同一フィールドに異なる型のデータがセットされています。
DB_DRV_ERROR_FIELD_NOT_NULL	指定されたフィールドは NULL を設定できません。
DB_DRV_ERROR_INIT_ERROR	初期化でエラーが発生しました。
DB_DRV_ERROR_INVALID_ACCESS_TYPE	SetData 関数又は SetFieldData 関数で指定した設定先はファイルアクセス専用です。
DB_DRV_ERROR_INVALID_ARGUMENT	メソッドの引数に誤りがあります。
DB_DRV_ERROR_INVALID_COLUMN_TYPE	選択リスト又はストアドプロシジャのパラメタ中にサポートされていないデータ型があります。
DB_DRV_ERROR_INVALID_CONNECT_NAME	接続名が不正です。
DB_DRV_ERROR_INVALID_MAX_SIZE	格納領域の最大長指定が不正です。

エラーコード	メッセージテキスト
DB_DRV_ERROR_INVALID_SQL	指定された SQL 文が不正です。
DB_DRV_ERROR_INVALID_SQL_EXCLUSIVE	WITH 句指定では EXCLUSIVE ロックを使用できません。
DB_DRV_ERROR_INVALID_SQL_TYPE	SetSQL 関数が実行されていないか，SQL が SELECT 文ではありません。
DB_DRV_ERROR_INVALID_TABLELIST_TYPE	テーブルの一覧表示以外はサポートされていません。
DB_DRV_ERROR_IN_ERROR	エラー処理中にエラーが発生しました。
DB_DRV_ERROR_NOT_ALLOC_PARAM_AREA	パラメタ設定領域が確保されていません。
DB_DRV_ERROR_NOT_BIND_PHOLDER	関連付けられていないプレースホルダがあります。
DB_DRV_ERROR_NOT_ENOUGH_MEMORY	メモリ不足のため処理を続行できません。
DB_DRV_ERROR_NOT_PLACE_HOLDER	SQL 文にプレースホルダがないのでパラメタ領域を確保できません。
DB_DRV_ERROR_NOT_SET_OBJECT	指定された接続名でオブジェクトは登録されていません。
DB_DRV_ERROR_NOT_SPECIFIED_MAX_SIZE	設定済み SQL 文に対して領域の最大長指定はできません。
DB_DRV_ERROR_NOT_SUPORT_TABLE_LIST	シノニムの一覧表示はサポートされていません。
DB_DRV_ERROR_OUTPUT_NOT_FOUND	OUTPUT データがありません。
DB_DRV_ERROR_PHOLDER_AND_QP	プレースホルダと？パラメタが混在する SQL 文の指定はできません。
DB_DRV_ERROR_PROC_COLUMN_LIST	プロシジャのフィールド情報は取得できません。(ORACLE)
DB_DRV_ERROR_RDA_FILE_ACCESS_PARAM_TYPE	LONG 型以外のパラメタでファイルアクセスはできません。
DB_DRV_ERROR_RDA_PARAM_TYPE	パラメタの型が未設定か又は設定された型が不正です。
DB_DRV_ERROR_READ_ONLY_RESULTSET	参照専用の ResultSet では，レコードを更新できません。
DB_DRV_ERROR_REQUEST_LIST_ROWS	テーブル又はフィールド名一覧の取得要求数が不正です。
DB_DRV_ERROR_REQUEST_RESULTSET_ROWS	ResultSet の生成に要求されたレコード数が不正です。
DB_DRV_ERROR_RESULTSET_NOT_FOUND	ResultSet が生成されていません。
DB_DRV_ERROR_SELECT_SET_OPERATION	集合演算子(UNION など)を含む SQL 文では更新可能な ResultSet を生成できません。

エラーコード	メッセージテキスト
DB_DRV_ERROR_SETDATA_TYPE	設定データと設定先の型が一致していません。
DB_DRV_ERROR_SETFIELDDATA	パラメタの設定先(フィールド又はレコード番号)が不正です。
DB_DRV_ERROR_SET_OVER_SIZE	指定された領域長が不正です。
DB_DRV_ERROR_SQL_NOT_FOUND	SQL 文が設定されていません。
DB_DRV_ERROR_STAT_COUNT	ステートメントの個数が上限値を超えました。
DB_DRV_ERROR_UPDATE_RESULTSET_NOT_FOUND	更新可能な ResultSet が生成されていません。
DB_DRV_ERROR_USERID_NULL	ユーザ ID が指定されていません。
DB_ERROR_CANNOT_ACCESS_WHILE_EXECUTED	SQL を実行中に当該 SQL で使用されている DBRArrayData クラスのインスタンスにアクセスすることはできません。
DB_ERROR_CANNOT_USE_NONXADRIVER	非 XA インターフェース用に作成された DBDriver オブジェクトは，XA インターフェース用として使用できません。
DB_ERROR_CANNOT_USE_XADRIVER	XA インターフェース用に作成された DBDriver オブジェクトは，非 XA インターフェース用として使用できません。
DB_ERROR_ALREADY_CONNECTED	既にデータベースに接続されています。
DB_ERROR_ALREADY_REGISTERED	DBTransaction オブジェクトは既に登録されています。
DB_ERROR_ARGUMENT_OUT_OF_RANGE	メソッドの引数は指定できる範囲を超えています。
DB_ERROR_BEFORE_EXECUTE	Execute メソッドが実行されていません。
DB_ERROR_CANCEL_FAILURE	実行中の処理をキャンセルできませんでした。
DB_ERROR_CANNOT_BE_NULL	文字列を指定する引数に NULL は指定できません。
DB_ERROR_CANNOT_EDIT_LAST_RECORD	カレントレコードは最後のレコードを超えています。更新はできません。
DB_ERROR_CANNOT_EXECUTE	SQL 文を実行できません。コミット又はロールバック実行のタイミングが不明です。
DB_ERROR_CANNOT_FIND_OBJECT	指定した名前を持つオブジェクトはありません。
DB_ERROR_CANNOT_GET_RESULTSET	プロシジャが検索結果を返さないため検索結果は取得できません。
DB_ERROR_CANNOT_REFRESH	Refresh メソッドは使用できません。

エラーコード	メッセージテキスト
DB_ERROR_CANNOT_USE_ABSTRACTNAME	データベース種別名が指定されていないためデータベース名は指定できません。
DB_ERROR_CANNOT_USE_DBCALLABLE	DBCallableStatement オブジェクトから作成された DBResultSet を使用する場合，このメソッドは実行できません。
DB_ERROR_CANNOT_USE_IN_CLOSE	ほかのスレッドで Execute メソッド又は Close メソッドを実行中のため，呼び出したメソッドは実行できません。
DB_ERROR_CANNOT_USE_IN_EXECUTE	ほかのスレッドで Execute メソッドを実行中のため，呼び出したメソッドは実行できません。
DB_ERROR_CANT_UPDATE	参照専用の DBResultSet オブジェクトは更新できません。
DB_ERROR_CAN_EXECUTE_SINGLE_SQL	今バージョンではエージェントで実行できる SQL 文は一つだけです。
DB_ERROR_CONNECTDBDEFINITIONNAME_LENGTH_IS_ZERO	指定したデータベース名の長さが 0 です。
DB_ERROR_CONNECTION_FAILED	DBMS との接続に失敗しました。
DB_ERROR_CONNECTSTRING_INVALID	接続文字列の形式が正しくありません。
DB_ERROR_CONSTRATION	更新レコードの内容は，データベースの整合性チェックに違反します。
DB_ERROR_CONVERT_ARRAY_TO_SCALAR	繰り返し構造のデータとスカラ型のデータは相互に変換できません。
DB_ERROR_CURRENT_RECORD_DELETED	カレントレコードは削除されています。
DB_ERROR_DAB_ACCESS_ERROR	DABroker でエラーが発生しました。詳細情報は DBSQLCA に設定されます。
DB_ERROR_DAB_ALREADY_LOADING	ドライバプログラムは既にロードしています。
DB_ERROR_DAB_ILLEGAL_VALUE	指定した引数が不正です。詳細情報は DBSQLCA に設定されます。
DB_ERROR_DAB_SYSTEMCALL_ERROR	システムコールでエラーが発生しました。詳細情報は DBSQLCA に設定されます。
DB_ERROR_DAB_UNINITIALIZE	データベースに接続していません。
DB_ERROR_DATA_CANNOT_CONVERT	設定されたデータを DBMS の正しいデータ型に変換できません。
DB_ERROR_DATA_TRUNCATED	データ変換でエラーが発生しました。GetParam メソッド,GetField メソッド,SetField メソッドの場合，取得するデータ型を修正する必要があります。
DB_ERROR_DBDEFINITION_IS_UNAVAILABLE_IN_XA	XA インタフェース使用時には接続先データベース定義は利用できません。

エラーコード	メッセージテキスト
DB_ERROR_DBDEFINITIONNAME_LENGTH_IS_ZERO	指定したデータベース種別名の長さが 0 です。
DB_ERROR_DRIVER_ERROR	DBMS でエラーが発生しました。詳細情報は DBSQLCA に設定されます。
DB_ERROR_DRIVER_NOT_LOADED	ドライバがロードされていません。
DB_ERROR_FIELD_NOT_DUPLICATABLE	指定されたフィールドは重複した値を設定できません。
DB_ERROR_FIELD_NOT_NULL	指定されたフィールドは NULL を設定できません。
DB_ERROR_FILE_NOT_FOUND	ファイルが見つかりません。
DB_ERROR_FILE_READ_ERROR	ファイルの読込みでエラーが発生しました。
DB_ERROR_FUNCION_NOT_READY	メソッドを呼び出すための準備ができていません。
DB_ERROR_INITIALIZE_ERROR	初期化に失敗しました。メモリ不足か環境設定の不正のため，このまま処理を続けることはできません。作成した DBDriverManager オブジェクトは使用できません。
DB_ERROR_INSTALLPATH_READ_ERROR	レジストリからのインストールパスの取得に失敗しました。
DB_ERROR_INVALID_ARGUMENT	メソッドの引数に誤りがあります。
DB_ERROR_IN_ASYNC_EXECUTE	SQL 文の非同期実行処理中です。
DB_ERROR_IN_TRANSACTION	トランザクションは既に開始されています。
DB_ERROR_IN_USE_OTHER_THREADS	オブジェクトはほかのスレッドで使用されているため，メソッドを実行できません。
DB_ERROR_MAXSIZE_NOT_EVEN	文字列の最大長が奇数の時に 2 バイト文字を使用することはできません。
DB_ERROR_MISSMATCH_DBMSKIND_IN_ENVFILE	環境情報[DAB_DBMSNAME]の値が間違っています。
DB_ERROR_NAME_ALREADY_USED	指定した名称は既に使用されています。
DB_ERROR_NAME_INVALID	指定した名前は不正です。
DB_ERROR_NOT_CONNECTED	データベースに接続していません。
DB_ERROR_NOT_ENOUGH_MEMORY	メモリ不足のため処理を続行できません。
DB_ERROR_NOT_EXECUTE_ASYNC	XA インターフェース使用時に非同期処理は指定できません。
DB_ERROR_NOT_FOUND	指定されたフィールドはありません。
DB_ERROR_NOT_FOUND_DBDEFINITION	接続先データベース定義情報が見つかりません。

エラーコード	メッセージテキスト
DB_ERROR_NOT_FOUND_DBMSKIND_IN_ENVFILE	環境情報に DBMS 種別を定義する変数[DAB_DBMSNAME]が定義されていません。
DB_ERROR_NOT_FOUND_USERID_IN_ENVFILE	環境情報に接続するユーザＩＤを定義する変数[DABCONNECT_USERID]が定義されていません。
DB_ERROR_NOT_IN_EDIT	更新のための準備ができていません。
DB_ERROR_NOT_IN_TRANSACTION	ステートメントごとの自動コミットが設定されています。
DB_ERROR_NOT_RELATIONSHIP_TO_DBRDATABASE	指定された DBRDatabase オブジェクトと関連付けることができませんでした。一つの DBRDatabase オブジェクトに対して関連付けられるオブジェクトは６４個までです。
DB_ERROR_NOT_SELECT_STATEMENT	指定された SQL 文は SELECT 文ではありません。SELECT 文を指定してください。
DB_ERROR_NOT_SUPPORTED	このバージョンではサポートされていません。
DB_ERROR_NOT_SUPPORT_XA	指定した Driver タイプでは，XA インターフェースを使用できません。
DB_ERROR_NOT_UPDATE_INSERT	SQL が UPDATE 文でも INSERT 文でもないためこのメソッドは無効です。
DB_ERROR_NO_AGENTBODY_INFORMATION	DBRAgentBody クラスに属する登録情報がありません。SetRegister メソッドを使用して登録情報を設定してください。
DB_ERROR_NO_AGENTCORE_INFORMATION	DBRAgentCore クラスに属する登録情報がありません。SetRegister メソッドを使用して登録情報を設定してください。
DB_ERROR_NO_AGENTOTHER_INFORMATION	DBRAgentOther クラスに属する登録情報がありません。SetRegister メソッドを使用して登録情報を設定してください。
DB_ERROR_NO_AGENTSQLLIST_INFORMATION	DBRAgentSQLList クラスに属する登録情報がありません。SetRegister メソッドを使用して登録情報を設定してください。
DB_ERROR_NO_CONNECTION_OBJECT	トランザクションに DBConnection が登録されていません。
DB_ERROR_NO_DATA_AREA	データ領域が存在しません。
DB_ERROR_NO_INSTANCE	指定された DBRArrayDataPtr または DBRArrayDataConstPtr オブジェクトは DBRArrayData クラスのインスタンスを保持していません。

エラーコード	メッセージテキスト
DB_ERROR_NO_REGISTER_INFORMATION	エージェントの登録情報がありません。SetRegister メソッドを使用して登録情報を設定してください。
DB_ERROR_OCCURRED_IN_CONSTRUCTOR	コンストラクタでエラーが発生しました。エラーの内容は GetErrorStatus メソッドで DBSQLCA オブジェクトを取得して参照してください。
DB_ERROR_OCCURRED_UNDER_ERROR_TRANSACTION	エラー処理中にメモリ不足になりました。
DB_ERROR_OUT_OF_RANGE	Next メソッド,PageNext メソッドが最後のレコードを超えて呼び出されました。
DB_ERROR_OUT_OF_RESULTSET	ResultSet の範囲を超えてカレントレコードを移動しようとしました。
DB_ERROR_OVER_LAST_AGENTRESULT	最後の実行結果の次の実行結果を取得しようとしました。
DB_ERROR_OVER_MAX_CURSORS	すべてのカーソルを使用中です。ステートメントを削除するなどして空きカーソルを増やしてから再度実行してください。
DB_ERROR_PARAM_NOT_FOUND	指定されたパラメタはありません。
DB_ERROR_PREPARE_ERROR	エージェントの実行結果取得の準備に失敗しました。設定情報を見直して再度 Execute メソッドを実行してください。
DB_ERROR_PROCEDURE_NOT_COMPLETE	プロシジャの実行が終了していません。
DB_ERROR_PROCEDURE_NOT_EXECUTE	プロシジャが実行されていません。
DB_ERROR_PROCEDURE_NOT_SET	プロシジャが指定されていません。
DB_ERROR_RESULTSET_NOT_EXIST	検索結果が取得できていません。
DB_ERROR_SELECT_NOT_EXECUTED	SELECT 文が実行されていません。Execute メソッドで SELECT 文を実行してから Open メソッドを呼び出してください。
DB_ERROR_STATEMENT_WITHOUT_PARAMETER	DBPreparedStatement オブジェクトを使用していないため，パラメタの取得・設定はできません。
DB_ERROR_SYNTAX_ERROR	指定された SQL 文に誤りがあります。詳細は DBSQLCA に設定されます。
DB_ERROR_TIMEOUT	タイムアウトが発生したため制御を戻します。
DB_ERROR_TOO_LARGE_DATA	指定したデータの値は大きすぎます。
DB_ERROR_USER_DEFINED_ERROR_NOT_LOADED	DBMS 非依存エラー情報は読み込まれていません。

8.2.2　DB_ERROR_DAB_ILLEGAL_VALUE での詳細コード

エラーコードが DB_ERROR_DAB_ILLEGAL_VALUE の場合は，さらに詳しい情報を取得できます。その情報は詳細コードとして取得できます。

詳細コードとそれに対応するメッセージテキストを表 8.2 に示します。

表 8-2　詳細コードとそれに対応するメッセージテキスト

詳細コード	メッセージテキスト
DB_DDCI_ERROR_ACCESS_UNSUPPORT_DB_ERROR	指定した DBMS は今バージョンではサポートしていません。
DB_DDCI_ERROR_AGENT_BODY_INF_IS_NOT_NULL_ERROR	DBRAgentBody オブジェクトが指定されています。
DB_DDCI_ERROR_AGENT_GROUPID_LENGTH_IS_NOT_ZERO_ERROR	グループＩＤの長さが 0 ではありません。
DB_DDCI_ERROR_AGENT_HEADER_COUNT_IS_ZERO_ERROR	DBRAgentHeader オブジェクトに指定したヘッダの個数が 0 です。
DB_DDCI_ERROR_AGENT_HEADER_INF_IS_NOT_NULL_ ERROR	DBRAgentHeader オブジェクトが指定されています。
DB_DDCI_ERROR_AGENT_LIST_TYPE_MISSMATCHER_ ERROR	エージェント登録名を省略したときの取得情報属性の値が間違っています。
DB_DDCI_ERROR_AGENT_OTHER_INF_IS_NOT_NULL_ERROR	DBRAgentOther オブジェクトが指定されています。
DB_DDCI_ERROR_AGENT_REGISTER_DB_KIND_LENGTH_IS_ZERO_ERROR	接続先 DBMS 識別名の長さが 0 です。
DB_DDCI_ERROR_AGENT_REGISTER_NAME_IS_NULL_ ERROR	エージェント登録名に NULL が指定されています。
DB_DDCI_ERROR_AGENT_REGISTER_NAME_LENGTH_IS_ ZERO_ERROR	エージェント登録名の長さが 0 です。
DB_DDCI_ERROR_AGENT_SQLLIST_INF_IS_NOT_NULL_ ERROR	DBRAgentSQLList オブジェクトが指定されています。
DB_DDCI_ERROR_AGENT_SQL_COUNT_IS_ZERO_ERROR	実行する SQL が一つも指定されていません。
DB_DDCI_ERROR_AGENT_VARIABLE_COUNT_IS_ZERO_ ERROR	可変値情報が一つも指定されていません。
DB_DDCI_ERROR_DBDEFINITIONNAME_MISSMATCHER_ERROR	一致するデータベース種別名が見つかりません。
DB_DDCI_ERROR_DBMSHOSTNAME_MISSMATCHER_ERROR	ホスト名称の指定が不正です。
DB_DDCI_ERROR_DBMSNAMELENGTH_IS_NOT_ZERO_ERROR	DBMS 名の指定が不正です。
DB_DDCI_ERROR_DBMSNAME_MISSMATCHER_ERROR	DBMS 名の指定が不正です。
DB_DDCI_ERROR_PASSWORD_MISSMATCHER_ERROR	パスワードの指定が不正です。

詳細コード	メッセージテキスト
DB_DDCI_ERROR_PRK_COLUMN_NAME_MISSMATCHER_ERROR	プライマリキーのフィールド名の指定が不正です。
DB_DDCI_ERROR_SQL_LENGTH_IS_ZERO_ERROR	指定した SQL 文は不正です。SQL 文の長さが 0 です。
DB_DDCI_ERROR_TOO_LONG_AGENT_HEADER_TEXT _ERROR	エージェントのヘッダは長すぎます。
DB_DDCI_ERROR_TOO_LONG_AGENT_REGISTER_COMMENT_ERROR	エージェントのコメントは長すぎます。
DB_DDCI_ERROR_TOO_LONG_AGENT_REGISTER_DB_HOST_ERROR	エージェントの DBMS 識別名は長すぎます。
DB_DDCI_ERROR_TOO_LONG_AGENT_REGISTER_DB_KIND_ERROR	エージェントのホスト名は長すぎます。
DB_DDCI_ERROR_TOO_LONG_AGENT_REGISTER_GROUPID_ERROR	エージェントのグループＩＤは長すぎます。
DB_DDCI_ERROR_TOO_LONG_AGENT_REGISTER_NAME_ ERROR	エージェントの登録名は長すぎます。
DB_DDCI_ERROR_TOO_LONG_AGENT_REGISTER_OWNER_ ERROR	エージェントの所有者名は長すぎます。
DB_DDCI_ERROR_TOO_LONG_AGENT_REGISTER_OWNER_ PASSWD_ERROR	エージェントの所有者のパスワードは長すぎます。
DB_DDCI_ERROR_TOO_LONG_AGENT_SQL_TEXT_ERROR	エージェントの SQL 文字列は長すぎます。
DB_DDCI_ERROR_TOO_LONG_AGENT_VARIABLE_DATA_ ERROR	エージェントの可変値のデータは長すぎます。
DB_DDCI_ERROR_TOO_LONG_AGENT_VARIABLE_NAME_ ERROR	エージェントの可変値名称は長すぎます。
DB_DDCI_ERROR_UNJUST_ACTIVEDB_ERROR	DBMS 種別の指定が不正です。
DB_DDCI_ERROR_UNJUST_AGENT_EXECUTE_OPTION_ ERROR	エージェントの実行時オプションの値が不正です。
DB_DDCI_ERROR_UNJUST_AGENT_LIST_CONDITION_ERROR	エージェントの登録情報取得時の検索条件の値が不正です。
DB_DDCI_ERROR_UNJUST_AGENT_LIST_TYPE_ERROR	エージェントの登録情報取得時の取得情報属性の値が不正です。
DB_DDCI_ERROR_UNJUST_AGENT_OPTION_TYPE_ERROR	エージェントの登録情報の上書き許可の指定が不正です。
DB_DDCI_ERROR_UNJUST_AGENT_REGISTER_CONNECT_ DB_ERROR	エージェント実行時の接続先 DBMS の指定が不正です。
DB_DDCI_ERROR_UNJUST_AGENT_REGISTER_EXECUTE_AUTHORIZATION_ERROR	エージェントの実行権限の指定が不正です。
DB_DDCI_ERROR_UNJUST_AGENT_REGISTER_REFERENCE_AUTHORIZATION_ERROR	エージェントの参照権限の指定が不正です。

詳細コード	メッセージテキスト
DB_DDCI_ERROR_UNJUST_AGENT_REGISTER_RESERVE_DATE_ERROR	エージェントの実行日付の指定が不正です。
DB_DDCI_ERROR_UNJUST_AGENT_REGISTER_RESERVE_DAYOFWEEK_ERROR	エージェントの実行曜日の指定が不正です。
DB_DDCI_ERROR_UNJUST_AGENT_REGISTER_RESERVE_TIME_ERROR	エージェントの実行時刻の指定が不正です。
DB_DDCI_ERROR_UNJUST_AGENT_REGISTER_VARIABLE_QUESTION_POSITION_ERROR	エージェントの SQL 位置の指定が不正です。
DB_DDCI_ERROR_UNJUST_AGENT_REGISTER_VARIABLE_SQL_POSITION_ERROR	エージェントの？パラメタの位置の指定が不正です。
DB_DDCI_ERROR_UNJUST_AGENT_REQUEST_COUNT_ERROR	エージェントの結果情報取得時の件数の指定が不正です。
DB_DDCI_ERROR_UNJUST_AGENT_RESULT_WRITE_MODE_ERROR	エージェントの実行結果の保存方法の指定が不正です。
DB_DDCI_ERROR_UNJUST_AGENT_VARIABLE_PRECISION_ERROR	エージェントの可変値の精度と属性の値が一致しません。
DB_DDCI_ERROR_UNJUST_AGENT_VARIABLE_SCALE_ERROR	エージェントの可変値の位取りと属性の値が一致しません。
DB_DDCI_ERROR_USERID_MISSMATCHER_ERROR	ユーザ ID の指定が不正です。

付録

付録 A　セットアップ

付録 A.1　DABroker for C++の組み込み(UNIX の場合)

UNIX の場合のセットアップは，次の手順で実行してください。

1. 日立 PP インストーラを使って，DABroker for C++で提供するファイルをマシンに組み込む。
2. 組み込まれたファイルの中から,セットアップコマンド（dabcppsetup）を実行する。

(1)　組み込み

DABroker for C++の組み込みは,日立 PP インストーラで実行します。

自マシン上の任意のディレクトリを管理ディレクトリとして設定できます。リモートファイルシステム上のディレクトリには設定できません。

(2)　DABroker for C++が提供するファイル

DABroker for C++の組み込みによってハードディスク上に作成されるファイル・ディレクトリを表 A-1 に示します。

表 A-1　DABroker for C++が提供するファイル（UNIX の場合）

ディレクトリ	ファイル	権限	説明
/opt/DABroker		drwxrwxrwx	DABroker ディレクトリ
	.HTC_8820.inf	-r--r--r--	Network Objectplaza 用の PP 固有情報ファイル
/opt/DABroker/bin		drwxr-xr-x	実行ファイル　ディレクトリ※1
	dabcppsetup	-r-x------	DABroker for C++　セットアップコマンド
/opt/DABroker/lib		drwxrwxrwx	動的ライブラリ　ディレクトリ※1
	libdabcpp10.sl(HP-UX(PA-RISC)版)	-rwxr-xr-x	DABroker for C++ 共用ライブラリ※1
	libdabcpp14.sl(HP-UX(PA-RISC)版)	-rwxr-xr-x	※1
	libdabcpp20.sl(HP-UX(PA-RISC)版)	-rwxr-xr-x	※1
	libdabcpp20st.sl(HP-UX(PA-RISC)版)	-rwxr-xr-x	※1
	libdabcpp32.so(HP-UX(IPF)版)	-rwxr-xr-x	※1
	libdabcpp32st.so(HP-UX(IPF)版)	-rwxr-xr-x	※1
	libdabcpp20.a(AIX版)	-rwxr-xr-x	※1
	libdabcpp20st.a(AIX版)	-rwxr-xr-x	※1
	libdabcpp30.so(Red Hat Linux版)	-rwxr-xr-x	※1
	libdabcpp40.so(Red Hat Linux版)	-rwxr-xr-x	※1
/opt/DABroker/cpp		drwxr-xr-x	DABroker for C++ ディレクトリ※1
/opt/DABroker/cpp/include		drwxr-xr-x	DABroker for C++ ヘッダ　ディレクトリ※1
	xxxxx.h	-r--r--r--	DABroker for C++ ヘッダ※1

ディレクトリ	ファイル	権限	説明
/opt/DABroker/cpp/patch_cpp		drwxr-xr-x	DABroker for C++ パッチ履歴ディレクトリ (バージョン 02-10 以降で提供されます)
/opt/DABroker/msg		drwxr-xr-x	メッセージ　ディレクトリ※1
/opt/DABroker/msg/C		drwxr-xr-x	C コード　メッセージ　ディレクトリ※1
	dabcppme.txt	-r--r--r--	※1
	dabdef_oracle7e.txt	-r--r--r--	※1
	dabdef_oracle7_xae.txt	-r--r--r--	※1
	dabdef_hirdbe.txt	-r--r--r--	※1
	dabdef_hirdb_xae.txt	-r--r--r--	※1
	dabdef_rdb1e.txt	-r--r--r--	※1
	dabdef_sqlke.txt	-r--r--r--	※1
	dabdef_xdmrde.txt	-r--r--r--	※1
	dabdef_xdmrd_dbse.txt	-r--r--r--	※1
	dabdef_xdmsd_dbse.txt	-r--r--r--	※1
	dabdef_sqlk_dbse.txt	-r--r--r--	※1
/opt/DABroker/msg/SJIS		drwxr-xr-x	sjis コード　メッセージ　ディレクトリ (Red Hat Linux 版では提供されません)※1
	dabcppmj.txt	-r--r--r--	※1
	dabdef_oracle7j.txt	-r--r--r--	※1
	dabdef_oracle7_xaj.txt	-r--r--r--	※1
	dabdef_hirdbj.txt	-r--r--r--	※1
	dabdef_hirdb_xaj.txt	-r--r--r--	※1
	dabdef_rdb1j.txt	-r--r--r--	※1
	dabdef_sqlkj.txt	-r--r--r--	※1
	dabdef_xdmrdj.txt	-r--r--r--	※1
	dabdef_xdmrd_dbsj.txt	-r--r--r--	※1
	dabdef_xdmsd_dbsj.txt	-r--r--r--	※1
	dabdef_sqlk_dbsj.txt	-r--r--r--	※1
/etc/.hitachi		drwxr-xr-x	インストーラ管理ディレクトリ
/etc/.hitachi/delete/		drwxr-xr-x	削除ファイル　ディレクトリ
/etc/.hitachi/delete/delete_8820		-rw-------	削除ファイル
/etc/.hitachi/remove		drwxr-xr-x	削除時実行プログラム　ディレクトリ
/etc/.hitachi/remove/remove_8820		-rwx------	削除時実行プログラム

注※1

セットアップコマンド(dabcppsetup)を実行したときに，運用ディレクトリへ移されるディレクトリ，及びコマンドを示します。

- /etc/.hitachi/delete/delete_8820 ファイルについて

このファイルは，日立 PP インストーラで DABroker for C++を削除する場合に，どのファイルを削除するかを記述します。このファイルには，次に示すファイルを記述します。

```
/opt/DABroker/.HTC_8820.inf
/opt/DABroker/bin/dabcppsetup
/opt/DABroker/lib/libdabcpp10.sl(HP-UX(PA-RISC)版)
/opt/DABroker/lib/libdabcpp14.sl(HP-UX(PA-RISC)版)
/opt/DABroker/lib/libdabcpp20.sl(HP-UX(PA-RISC)版)
/opt/DABroker/lib/libdabcpp20st.sl(HP-UX(PA-RISC)版)
/opt/DABroker/lib/libdabcpp32.so(HP-UX(IPF)版)
/opt/DABroker/lib/libdabcpp32st.so(HP-UX(IPF)版)
/opt/DABroker/lib/libdabcpp20.a(AIX版)
/opt/DABroker/lib/libdabcpp20st.a(AIX版)
/opt/DABroker/lib/libdabcpp30.so(Red Hat Linux版)
/opt/DABroker/lib/libdabcpp40.so(Red Hat Linux版)
/opt/DABroker/msg/C/dabcppme.txt
/opt/DABroker/msg/SJIS/dabcppmj.txt
```

上記以外のファイルは，/etc/.hitachi/remove/remove_8820 ファイル（シェル）で削除します。

- /etc/.hitachi/remove/remove_8820 ファイルについて

  このファイルは日立 PP インストーラで DABroker for C++を削除する場合に起動されるファイル（シェル）です。

  このファイル（シェル）では次のことが実行されます。

  1. /opt/DABroker/bin ディレクトリ下にディレクトリ・ファイルがなければ/opt/DABroker/bin ディレクトリを削除します。
  2. /opt/DABroker/lib ディレクトリ下にディレクトリ･ファイルがなければ/opt/DABroker/lib ディレクトリを削除します。
  3. /opt/DABroker/cpp ディレクトリを削除します。
  4. DABroker for COBOL が組み込まれていない場合は，/opt/DABroker/msg/C ディレクトリ下のファイル"dabdef_hirdbe.txt"，"dabdef_hirdb_xae.txt"，"dabdef_oracle7e.txt"，"dabdef_oracle7_ xae.txt"，"dabdef_xdmrde.txt"，"dabdef_rdb1e.txt"，"dabdef_sqlke.txt"，"dabdef_xdmrd_dbse.txt"，"dabdef_xdmsd_dbse.txt"，"dabdef_sqlk_dbse.txt"を削除します。
  5. /opt/DABroker/msg/C ディレクトリ下にディレクトリ・ファイルがなければ/opt/DABroker/msg/C を削除します。
  6. DABroker for COBOL が組み込まれていない場合は/opt/DABroker/msg/SJIS ディレクトリ下のファイル"dabdef_hirdbj.txt"，"dabdef_hirdb_xaj.txt"，"dabdef_oracle7j.txt"，"dabdef_oracle7_xaj.txt"，"dabdef_xdmrdj.txt"，"dabdef_rdb1j.txt"，"dabdef_sqlkj.txt"，"dabdef_xdmrd_dbsj.txt"，"dabdef_xdmsd_dbsj.txt"，"dabdef_sqlk_dbsj.txt"を削除します。
  7. /opt/DABroker/msg/SJIS ディレクトリ下にディレクトリ・ファイルがなければ/opt/DABroker/msg/SJIS ディレクトリを削除します。
  8. /opt/DABroker/msg ディレクトリ下にディレクトリ・ファイルがなければ/opt/DABroker/msg ディレクトリを削除します。
  9. /opt/DABroker ディレクトリ下にディレクトリ・ファイルがなければ/opt/DABroker ディレクトリを削除します。

(3)　セットアップコマンドの実行

dabcppsetup [-d]

インストールした DABroker for C++を運用ディレクトリに移動します。

DABroker 運用ディレクトリは，DABroker のセットアップ（dabsetup コマンド）で指定した同じディレクトリです。

-d：DABroker for C++を運用ディレクトリから削除する場合に指定します。

注意

- DABroker for C++の組み込み後に，DABroker 本体のセットアップ(dabsetup)を実行した場合は，dabcppsetup を実行する必要はありません。
- セットアップコマンド実行前に DABroker 組み込みディレクトリ・ファイルを DABroker 管理者ユーザ／グループに変更してください。
- セットアップコマンドは，スーパユーザで実行してください。
- 運用ディレクトリに移動対象ファイルと同じ名称のファイルがあった場合，ファイルを上書きします。このとき，ファイルの上書きを禁止した属性などで，ファイルが上書きができない場合は，セットアップを終了します。また，ディレクトリ・ファイルはセットアップ前の状態に戻します。
- -d オペランド指定時，テキストビジーなどの原因で削除に失敗することがあります。その場合は OS の rm コマンドで，残ったファイルを削除してください。
- OS から DABroker for C++を完全に削除する場合は，日立 PP インストーラで削除してください。
- -d オペランド指定時, DABroker 運用ディレクトリ下にある DABroker for C++以外は削除されません。このため，必要に応じて，システム管理者がファイルを削除してください。
- DABroker for C++を DABroker 運用ディレクトリへの移動する場合，本セットアップコマンド実行前に，DABroker のセットアップが終了している必要があります。
- 次に示す順序で作成された環境で dabsetup コマンドを実行すると, dabsetup コマンドの実行に失敗します。この場合，dabsetup コマンドを実行する前に DABroker for C++ を再度インストールしてから，再度 dabsetup コマンドを実行してください。
  1. DABroker ，及び DABroker for C++ をインストールする。
  2. DABroker 運用ディレクトリを，/opt/DABroker 以外に設定する。
  3. DABroker だけを上書きインストールする。

(4)　ロケールへの対応

DABroker for C++では，実行環境のロケールに応じてメッセージや日付の表記を動的に切り替えることができます。次の環境情報に応じて切り替えてください。

- ロケール情報の設定

  環境変数 LANG には，使用するユーザの環境に合わせて国際化情報のロケールを設定する必要があります。

  各環境変数の詳細については，各 OS のマニュアルを参照してください。
- メッセージ情報の格納先
  - 組み込み先ディレクトリ/msg/SJIS(日本語シフト JIS コードメッセージ)
  - 組み込み先ディレクトリ/msg/C(英語 ASCII コードメッセージ)

    Red Hat Linux 版では英語 ASCII だけを提供します。

付録 A.2　DABroker for C++の組み込み(Windows の場合)

(1)　インストール

Windows の場合は，次の手順でインストールしてください。

DABroker for C++のインストールは，日立総合インストーラを使用して行います。

インストールを実行すると，ダイアログボックスが表示されますので，ガイダンスに従ってください。

1. Administrator グループのメンバであるユーザアカウントでログオンする。
2. 日立総合インストーラを使用して，DABroker for C++のインストールを開始する。
（先に DABroker 本体が組み込まれていること）
3. 表示されるウィザードに従ってインストールする。

(2)　DABroker for C++が提供するファイル

DABroker for C++の組み込によってハードディスク上に作成されるファイル・フォルダを表 A-2 に示します。¥DABroker は DABroker のインストール先フォルダです。

表 A-2　DABroker for C++が提供するファイル（Windows の場合）

ディレクトリ	ファイル	権限	説明
¥DABroker		Administrator	DABroker 本体組込みフォルダ
¥DABroker¥lib		-	動的ライブラリ　フォルダ
	dabcpp10.dll	Administrator	DABroker for C++ 動的ライブラリ
	dabcpp10d.dll	Administrator	DABroker for C++ 動的ライブラリ（デバッグ版）※
	dabcpp14.dll	Administrator	DABroker for C++ 動的ライブラリ
	dabcpp14d.dll	Administrator	DABroker for C++ 動的ライブラリ（デバッグ版）※
	dabcpp20.dll	Administrator	DABroker for C++ 動的ライブラリ
	dabcpp20d.dll	Administrator	DABroker for C++ 動的ライブラリ（デバッグ版）※
¥DABroker¥cpp¥lib		－	DABroker for C++ ライブラリフォルダ
	dabcpp10.lib	Administrator	DABroker for C++ ライブラリ
	dabcpp10d.lib	Administrator	DABroker for C++ ライブラリ（デバッグ版）※
	dabcpp14.lib	Administrator	DABroker for C++ ライブラリ
	dabcpp14d.lib	Administrator	DABroker for C++ ライブラリ（デバッグ版）※
	dabcpp20.lib	Administrator	DABroker for C++ ライブラリ
	dabcpp20d.lib	Administrator	DABroker for C++ ライブラリ（デバッグ版）※
¥DABroker¥cpp¥include		-	DABroker for C++ ヘッダ　フォルダ
	xxxxx.h	Administrator	DABroker for C++ ヘッダ
¥DABroker¥msg		-	メッセージ　フォルダ
¥DABroker¥msg¥C		-	C コード　メッセージ　フォルダ
	dabcppme.txt	Administrator	
	dabdef_oracle7e.txt	Administrator	
	dabdef_oracle7_xae.txt	Administrator	
	dabdef_hirdbe.txt	Administrator	
	dabdef_hirdb_xae.txt	Administrator	
	dabdef_rdb1e.txt	Administrator	
	dabdef_sqlke.txt	Administrator	
	dabdef_xdmrde.txt	Administrator	

ディレクトリ	ファイル	権限	説明
	dabdef_sqlanye.txt	Administrator	C コード　メッセージ　フォルダ
	dabdef_sqlservere.txt	Administrator	
	dabdef_adaptivee.txt	Administrator	
	dabdef_xdmrd_dbse.txt	Administrator	
	dabdef_xdmsd_dbse.txt	Administrator	
	dabdef_sqlk_dbse.txt	Administrator	
¥DABroker¥msg¥SJIS		-	sjis コード　メッセージ　フォルダ
	dabcppmj.txt	Administrator	
	dabdef_oracle7j.txt	Administrator	
	dabdef_oracle7_xaj.txt	Administrator	
	dabdef_hirdbj.txt	Administrator	
	dabdef_hirdb_xaj.txt	Administrator	
	dabdef_rdb1j.txt	Administrator	
	dabdef_sqlkj.txt	Administrator	
	dabdef_xdmrdj.txt	Administrator	
	dabdef_sqlanyj.txt	Administrator	
	dabdef_sqlserverj.txt	Administrator	
	dabdef_adaptivej.txt	Administrator	
	dabdef_xdmrd_dbsj.txt	Administrator	
	dabdef_xdmsd_dbsj.txt	Administrator	
	dabdef_sqlk_dbsj.txt	Administrator	
¥DABroker¥cpp¥patch_cpp		-	DABroker for C++ パッチ履歴フォルダ
	PATCHLOG.TXT	Administrator	パッチ履歴ファイル

注※ UAP のデバック時に使用します。

(3)　アンインストール

アンインストールは，コントロールパネルの「アプリケーションの追加と削除」－「インストールと削除」で行います。DABroker for C++を指定し，表示されるウィザードに従ってアンインストールしてください。

DABroker 本体も一緒に削除する場合は，先に DABroker for C++をアンインストールしてください。

(4)　ロケールへの対応

DABroker for C++では，実行環境のロケールに応じてメッセージや日付の表記を動的に切り替えることを可能とします。次のロケール情報に応じて切り替えてください。

- ロケール情報の設定
  コントロールパネルの「地域」の設定に従います。
- メッセージ情報の格納先
  - 組み込み先フォルダ¥msg¥SJIS(日本語シフト JIS コードメッセージ)
  - 組み込み先フォルダ¥msg¥C(英語 ASCII コードメッセージ)

付録 B　用語解説

(記号)

?パラメタ

SQL 文,又はストアドプロシジャに対して，実行時にアプリケーションからパラメタ値を渡せます。SQL の文字列の中で，アプリケーションから値を渡される個所に「?」を指定しておきます。実行すると「?」に対してパラメタを渡せます。

この「?」を?パラメタといいます。

ORACLE のデータベースでは，「プレースホルダ」のことを指します。

(英字)

BLOB(Binaly Large Object)

binary データの中でも，フィールド値が大きい（大容量のデータを扱えるフィールドの属性を持つ）データ型のこと。

このマニュアルでは，次のデータ型も同じ扱いになります。

HiRDB：BINARY

ORACLE：LONGRAW

SQL Anywhere：LONG BINARY,IMAGE

Adaptive Server Anywhere：LONG BINARY,IMAGE,java serialization,java.lang.Object

SQL Server：IMAGE

一般には，画像，音声などのマルチメディアデータを扱うデータ型です。

Database Connection Server

Windows 上から，メインフレーム系データベースにアクセスできるシステムのこと。別途購入が必要です。

DBMS(Databese Management System)

データベース管理システムのこと。

データベースを効率的に管理・利用するためのプログラムです。

NULL 値

データベースでいう NULL は不定の値のことで，ほかのどの値とも同じではないものを指します。NULL を認めないというのは，データの入力が必須であることを意味しています。

RDA Link for Gateway

Windows，又は HP-UX 上から，メインフレーム系データベースにアクセスできるシステムのこと。別途購入が必要です。

ResultSet

検索した結果を入れる仮想の表のこと。検索結果は，カーソルでレコードを特定し，Getxxx メソッドでフィールドから取り出します。

TPBroker

DABroker では，TPBroker で行う 2 相コミットを利用して，トランザクションを実現します。TPBroker を利用することで，DABroker のトランザクションと，グローバルトランザクションの同期がとれるようになります。

XA インタフェース

X/Open XA インタフェースは，分散トランザクション処理システムでのトランザクションマネージャとリソースマネージャの接続インタフェースを規定した X/Open の標準仕様です。

XA インタフェースを使って，リソースマネージャのトランザクション処理をトランザクションマネージャで制御できます。

実際には，TPBroker と DBMS 間のインタフェースと考えることができ，マルチスレッド対応の XA インタフェースを利用することで，DABroker，TPBroker を使って，複数 DBMS にまたがるトランザクション制御ができるようになります。

(ア行)

オブジェクト

クラスを基に生成された実態であり，new 演算子などで生成されます。

インスタンスとも呼ばれます。

(カ行)

カレントレコード

カーソルの位置付いているレコードのこと。カレントレコードが操作の対象になります。

キャスト

一時的に，指定したデータ型に変換すること。

クラス

あるデータ型(プロパティ)とそれに対する処理(メソッド)を一まとまりにした型の定義です。

コミット

トランザクションを終了し，そのトランザクションのデータベースに対するすべての操作を有効に（保存）すること。

(サ行)

自動コミット

アプリケーションからトランザクション制御を指定しなかった場合，SQL を実行するごとに（自動的に）コミットします。これを自動コミットと呼んでいます。

ストアドプロシジャ

ストアドプロシジャとは，SQL で記述した処理手続きをデータベースに登録したものです。

ストアドプロシジャは，CREATE PROCEDURE で手続きとして定義し，実行時にアプリケーションから，登録されている処理を呼び出します。

ストアドプロシジャを使うと，SQL 文を 1 行ずつ実行するのと比べて，次のようなメリットがあります。

- SQL 文を実行すると，1 行ごとにアプリケーションと DBMS 間で処理の流れが発生します。しかし，ストアドプロシジャを利用すれば，一度にまとまった単位の SQL を実行できるため，処理速度の向上が望めます。
- 一つストアドプロシジャを作成しておけば，複数のアプリケーションから使用できるため，アプリケーションの作成効率も向上できます。
- 手続き中の SQL 文はコンパイルされた形式（SQL オブジェクト）でサーバ側に登録されているため，SQL 解析のオーバヘッドも削減できます。

(タ行)

抽象データ型データ

SQL の CREATE TYPE で定義するデータ型を示します。データ型中に属性定義，ルーチンなどをユーザが任意に定義できます。

データ操作

データベースに対して，データの追加，更新，削除を行うこと。

デッドロック

複数のトランザクションが複数の資源を競合し合い，お互いに相手のトランザクションによる資源の占有解除を待っている状態です。

同期処理

DABroker はアプリケーションから SQL 文の処理要求を受け付けると，アプリケーションに制御を戻さないまま，DBMS に対して SQL 文の処理を要求します。アプリケーションに制御が戻るのは，SQL 文の処理が完了した時点です。

トランザクション処理

論理的な仕事の単位で，一連のデータベースの操作などの集まりのこと。また，回復や排他制御の基本単位でもあります。

(ナ行)

2 相コミット

トランザクションが二つ以上に分散された環境で，同期点での処理をプリペア処理（更新準備）とコミット処理（更新処理）という 2 段階に分けて行う方法です。

2 相コミットではトランザクション処理中に障害が発生した場合でも，すべてのリソースオブジェクトを矛盾なく自動的にロールバックできます。

DABroker から 2 相コミットを実行するためには，TPBroker(トランザクションマネージャ)が必要です。

2 相コミットは 2 フェーズコミットとも呼ばれています。

(ハ行)

排他エラー

アプリケーションがデータベースにアクセスしようとしたときに，アクセス対象のレコードが，他のアプリケーションによってロックされている状態を指します。

非同期処理

アプリケーションから SQL 文の処理要求を受け取ると，制御をいったんアプリケーションに戻し，同時に DBMS に対して SQL 文の処理を要求します。したがって，アプリケーションでは，SQL 文の処理結果が戻る前に，次の処理を行うことができます。

このため，複数のデータベースアクセスを並行して行えます。

フィールド

このマニュアルでは，列やカラムのことを指します。これに対して，行やローのことをレコードと呼んでいます。ただし，ストアドプロシジャの説明では，フィールドのことをパラメタと表記しています。

フィールドは，フィールド名，又は先頭フィールドを 1 としたフィールド番号によって特定できます。

プロパティ

クラスのメンバとして宣言された変数であり，C++ではメンバ変数と呼ばれるものを指します。

(マ行)

マルチスレッド

複数の独立したプログラム部（スレッド）を同時に実行すること。

メソッド

クラスのメンバとして宣言された関数であり，C++ではメンバ関数と呼ばれるものを指します。

(ラ行)

レコード

このマニュアルでは，行やローのことを指します。これに対して，列やカラムのことをフィールドと呼んでいます。

レコードは，レコード名の指定，レコードへのカーソルの位置付け，又は先頭レコードを 1 としたレコード番号の指定によって特定できます。

ロック

アプリケーションから，検索レコードのロックと，テーブルのロックを掛けられます。検索レコードのロックは，クラスライブラリのメソッドで制御できます。テーブルのロックは SQL 文を使います。

ロールバック

トランザクションを終了し，そのトランザクションによるデータベースに対するすべての操作を破棄し，データベースをトランザクション開始時の状態まで戻す（回復する）こと。

索引

記号

数字

A

B

C

D

E

F

G

H

I

K

N

O

P

Q

R

S

T

U

V

W

X

あ

い

え

お

か

き

く

け

こ

し

す

せ

ち

て

と

は

ひ

ふ

へ

ま

め

り

れ

ろ