

Oracle Call Interface

プログラマーズ・ガイド

リリース 2 (9.2)

2002 年 7 月

部品番号 : J06295-01

ORACLE®

Oracle Call Interface プログラマーズ・ガイド, リリース 2 (9.2)

部品番号 : J06295-01

原本名 : Oracle Call Interface Programmer's Guide, Release 2 (9.2)

原本部品番号 : A96585-01 (Vol.1)、A96586-01 (Vol.2)

原本著者 : Jack Melnick

原本協力者 : Eric Belden, Phil Locke, G. Arora, A. Bande, Jenny Chai, S. Chandiramani, S. Chandrasekar, D. Chatterjee, Ernest Chen, L. Chidambaran, A. Downing, S. Gollapudi, R. Govindarajan, W. He, M. Joglekar, S. Kaluskar, R. Kambo, R. Kasamsetty, A. Katti, B. Khaladkar, S. Kotsovolos, V. Krishnamurthy, S. Krishnaswamy, Geoff Lee, Cindy Lim, Annie Liu, K. Mohan, R. Murthy, R. Pingte, D. Saha, B. Trute, S. Vedala, Wei Wang, Lik Wong, Jianping Yang, Valarie Moore

Copyright © 1996, 2002 Oracle Corporation. All rights reserved.

Printed in Japan.

制限付権利の説明

プログラム (ソフトウェアおよびドキュメントを含む) の使用、複製または開示は、オラクル社との契約に記された制約条件に従うものとします。著作権、特許権およびその他の知的財産権に関する法律により保護されています。

当プログラムのリバース・エンジニアリング等は禁止されています。

このドキュメントの情報は、予告なしに変更されることがあります。オラクル社は本ドキュメントの無謬性を保証しません。

* オラクル社とは、Oracle Corporation (米国オラクル) または日本オラクル株式会社 (日本オラクル) を指します。

危険な用途への使用について

オラクル社製品は、原子力、航空産業、大量輸送、医療あるいはその他の危険が伴うアプリケーションを用途として開発されておりません。オラクル社製品を上述のようなアプリケーションに使用することについての安全確保は、顧客各位の責任と費用により行ってください。万一かかる用途での使用によりクレームや損害が発生いたしましても、日本オラクル株式会社と開発元である Oracle Corporation (米国オラクル) およびその関連会社は一切責任を負いかねます。当プログラムを米国国防総省の米国政府機関に提供する際には、『Restricted Rights』と共に提供してください。この場合次の Notice が適用されます。

Restricted Rights Notice

Programs delivered subject to the DOD FAR Supplement are "commercial computer software" and use, duplication, and disclosure of the Programs, including documentation, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement. Otherwise, Programs delivered subject to the Federal Acquisition Regulations are "restricted computer software" and use, duplication, and disclosure of the Programs shall be subject to the restrictions in FAR 52.227-19, Commercial Computer Software - Restricted Rights (June, 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

このドキュメントに記載されているその他の会社名および製品名は、あくまでその製品および会社を識別する目的にのみ使用されており、それぞれの所有者の商標または登録商標です。

目次

はじめに	xxxix
対象読者	xxxix
このマニュアルの構成	xxxix
関連文書	xxxix
表記規則	xxxix

Oracle Call Interface の新機能	xxxix
Oracle9i リリース 2 (9.2) での Oracle Call Interface の新機能	xl
Oracle9i リリース 1 (9.0.1) での Oracle Call Interface の新機能	xli
Oracle8i リリース 8.1.6 での Oracle Call Interface の新機能	xlv

第 I 部 OCI の概念

1 概要およびアップグレード

OCI の概要	1-2
OCI の利点	1-3
OCI アプリケーションの構築	1-3
OCI の分類	1-5
手続き型および非手続き型要素	1-5
オブジェクトのサポート	1-6
SQL 文	1-7
カプセル化されたインタフェース	1-12
ユーザー認証およびパスワード管理の簡素化	1-12
アプリケーションのパフォーマンスおよび拡張性を改善する拡張機能	1-13

OCI によるオブジェクトのサポート	1-13
クライアント側のオブジェクト・キャッシュ	1-14
アソシエイティブ・インタフェースおよびナビゲーション・インタフェース	1-14
オブジェクト用のランタイム環境	1-15
型管理、マッピングおよび操作関数	1-15
Object Type Translator	1-16
OCI での Oracle Advanced Queuing のサポート	1-16
XA ライブラリ・サポート	1-17
既存アプリケーションのアップグレードの簡素化	1-17
互換性およびアップグレード	1-18
使用されなくなった OCI ルーチン	1-18
サポートされない OCI ルーチン	1-20
互換性	1-21
アップグレード	1-22

2 OCI プログラミングの基本

OCI プログラミングの概要	2-2
OCI プログラム構造	2-3
OCI データ構造	2-5
ハンドル	2-5
ハンドルの割当てと解放	2-7
環境ハンドル	2-9
エラー・ハンドル	2-9
サービス・コンテキスト・ハンドルとそれに対応付けられたハンドル	2-9
文ハンドル、バインド・ハンドルおよび定義ハンドル	2-11
記述ハンドル	2-11
複合オブジェクト検索ハンドル	2-12
スレッド・ハンドル	2-12
サブスクリプション・ハンドル	2-12
ダイレクト・パス・ハンドル	2-12
プロセス・ハンドル	2-13
接続プール・ハンドル	2-13
ハンドル属性	2-13
ユーザー・メモリーの割当て	2-14

記述子	2-15
スナップショット記述子	2-16
LOB/FILE データ型ロケータ	2-17
パラメータ記述子	2-18
ROWID 記述子	2-18
日時および時間隔の記述子	2-18
複合オブジェクト記述子	2-19
アドバンスト・キューイング記述子	2-19
LDAP ベースのパブリッシュ・サブスクライブ通知	2-19
ユーザー・メモリーの割当て	2-19
OCI プログラミング・ステップ	2-20
OCI 環境の初期化	2-21
OCI 環境の作成	2-21
共有データ・モード	2-22
ハンドルおよび記述子の割当て	2-25
アプリケーションの初期化、接続およびセッション作成	2-26
SQL 文の処理	2-29
コミットまたはロールバック	2-29
アプリケーションの終了	2-30
エラー処理	2-31
切捨ておよび NULL データのリターン・コードおよびエラー・コード	2-33
他の値を戻す関数	2-34
その他のコーディング・ガイドライン	2-35
パラメータの型	2-35
列への NULL の挿入	2-36
インジケータ変数	2-36
コールの取消し	2-38
位置指定の更新および削除	2-39
予約語	2-39
非ブロック化モード	2-41
ブロック化モードの設定	2-42
非ブロック化コールの取消し	2-42
非ブロック化の例	2-42
OCI プログラムでの PL/SQL 使用	2-44

OCI グローバリゼーション・サポート	2-45
UTF-16 環境	2-46
文字長セマンティクス	2-48
キャラクタ・セットのサポート	2-48
OCI からのクライアント・キャラクタ・セットの制御	2-48
OCI でキャラクタ・セットを制御するコードの例	2-49
文字制御と OCI インタフェース	2-50
OCI データベースのグローバリゼーション・サポート関数	2-51
OCI 文字列操作関数	2-51
OCI 文字分類関数	2-53
OCI 文字変換関数	2-54
OCI メッセージ関数	2-54

3 データ型

Oracle データ型	3-2
外部データ型コードの使用	3-4
内部データ型	3-4
LONG、RAW、LONG RAW、VARCHAR2	3-5
文字列およびバイト配列	3-5
UROWID	3-6
外部データ型	3-7
VARCHAR2	3-9
NUMBER	3-10
INTEGER	3-11
FLOAT	3-11
STRING	3-12
VARNUM	3-13
LONG	3-13
VARCHAR	3-13
DATE	3-14
RAW	3-15
VARRAW	3-15
LONG RAW	3-15
UNSIGNED	3-15
LONG VARCHAR	3-16
LONG VARRAW	3-16

CHAR	3-16
CHARZ	3-17
新しい外部データ型	3-18
名前付きデータ型（オブジェクト、VARRAY、NESTED TABLE）	3-18
REF	3-18
ROWID 記述子	3-19
LOB 記述子	3-19
日時および時間隔のデータ型記述子	3-22
C オブジェクト・リレーショナル・データ型マッピング	3-24
データ変換	3-24
LOB データ型記述子のデータ変換	3-26
日時および時間隔データ型のデータ変換	3-27
日時および日付のアップグレード規則	3-29
型コード	3-29
SQLT 値および OCI_TYPECODE 値の関係	3-31
oratypes.h の定義	3-33

4 OCI での SQL 文の使用

SQL 文の処理の概要	4-2
SQL 文の処理	4-2
文の準備	4-4
プリコンパイルされた SQL 文を複数のサーバーで使用方法	4-5
バインド	4-6
文の実行	4-7
実行スナップショット	4-7
実行モード	4-8
OCIStmtExecute() 用のバッチ・エラー・モード	4-9
選択リスト項目の記述	4-12
暗黙的記述	4-13
問合せの明示的記述	4-14
定義	4-15
結果のフェッチ	4-16
LOB データのフェッチ	4-16
プリフェッチ・カウンタの設定	4-17

スクロール・カーソル	4-17
OCI でのスクロール・カーソルのサポート	4-18
スクロール・カーソルでのアクセスの例	4-19

5 バインドと定義

バインド	5-2
名前付きバインドおよび定位置バインド	5-4
OCI 配列インタフェース	5-5
PL/SQL のプレースホルダのバインド	5-5
バインドで使用するステップ	5-6
PL/SQL の例	5-7
拡張バインド操作	5-10
名前付きデータ型のバインド	5-10
REF のバインド	5-10
LOB のバインド	5-10
OCI_DATA_AT_EXEC モードでのバインド	5-16
REF カーソル変数のバインド	5-17
バインド情報の概要	5-17
定義	5-18
定義に使用するステップ	5-19
拡張定義	5-21
拡張定義操作	5-21
名前付きデータ型出力変数の定義	5-21
REF 出力変数の定義	5-22
LOB 出力変数の定義	5-22
PL/SQL 出力変数の定義	5-24
ピース単位フェッチの定義	5-24
構造体配列のバインドと定義	5-25
スキップ・パラメータ	5-26
構造体配列で使われる OCI コール	5-28
構造体配列とインジケータ変数	5-28
RETURNING 句を使用した DML	5-29
RETURNING 句を使用した DML の使用方法	5-29
RETURNING...INTO 変数のバインド	5-30
エラー処理	5-31

RETURNING REF...INTO 句を使用した DML	5-31
コールバックに関するその他の注意	5-33
DML RETURNING 文に対する配列インタフェース	5-33
バインドおよび定義における文字変換の問題	5-34
キャラクタ・セットの選択	5-34
OCI でのクライアント・キャラクタ・セットの設定	5-35
OCI_ATTR_MAXDATA_SIZE 属性の使用	5-36
OCI_ATTR_MAXCHAR_SIZE 属性の使用	5-37
バインド時のバッファの拡張	5-37
定義時の制約チェック	5-39
文字長セマンティクスの互換性に関する一般的な問題	5-40
文字変換を伴うバインドと定義のコード例	5-40
PL/SQL REF CURSOR および NESTED TABLE	5-43
ランタイム・データ割当てとピース単位操作	5-44
ピース単位操作に有効なデータ型	5-44
LOB に対するバインドおよび定義	5-45
ピース単位操作の種類	5-45
実行時の INSERT または UPDATE データの提供	5-47
PL/SQL でのピース単位操作	5-50
実行時のフェッチ情報の提供	5-50
コールバックなしのピース単位操作に関する追加情報	5-52

6 スキーマ・メタデータの記述

スキーマ・メタデータの記述	6-2
OCIDescribeAny() の使用	6-2
制限	6-4
型および属性の注意	6-4
パラメータ属性	6-5
OCI_PTYPE_TABLE 型または OCI_PTYPE_VIEW 型	6-7
プロシージャ / ファンクション / サブプログラム属性	6-8
パッケージ属性	6-8
型属性	6-9
型属性の属性	6-11
型メソッド属性	6-12
コレクション属性	6-13

シノニム属性	6-14
順序属性	6-15
列属性	6-15
引数 / 結果属性	6-17
リスト属性	6-19
スキーマ属性	6-19
データベース属性	6-20
記述での文字長セマンティクスのサポート	6-21
OCIDescribeAny() の使用例	6-23
表用の列データ型の取出し	6-23
ストアド・プロシージャの記述	6-25
オブジェクト型の属性の取出し	6-27
名前付きコレクション型のコレクション要素のデータ型の取出し	6-29
文字長セマンティクスを使用した記述	6-31

7 LOB と FILE の操作

LOB での OCI 関数の使用	7-2
内部 LOB の作成と変更	7-2
表内の FILE とオペレーティング・システム・ファイルの関連付け	7-3
オブジェクトの LOB 属性	7-3
オブジェクトの LOB 属性への書込み	7-3
LOB 属性を持つ一時オブジェクト	7-4
LOB の配列インタフェース	7-4
LOB および FILE の関数	7-5
LOB の読み込み / 書き込みパフォーマンスを向上するための関数	7-10
LOB バッファリング関数	7-11
LOB のオープンおよびクローズのための関数	7-11
LOB 読み込みおよび書き込みコールバック	7-13
ストリーム転送のコールバック・インタフェース	7-14
コールバックを使用した LOB の読み込み	7-14
コールバックを使用した LOB の書き込み	7-16
一時 LOB のサポート	7-17
一時 LOB の作成および解放	7-18
一時 LOB の継続時間	7-18
一時 LOB の例	7-19

8 スケーラブルなプラットフォームの管理

OCI でのトランザクションのサポート	8-2
トランザクションの複雑度のレベル	8-3
トランザクションの例	8-9
関連する初期化パラメータ	8-10
パスワードおよびセッションの管理	8-11
認証管理	8-11
パスワード管理	8-12
セッション管理	8-13
中間層アプリケーション	8-14
中間層アプリケーションの属性	8-15
中間層の例	8-17
認証の属性	8-20
外部で初期化されたコンテキスト	8-20
外部で初期化されたコンテキストの OCI 属性	8-21
外部で初期化されたコンテキストによる OCISessionBegin() の使用	8-22

9 OCI プログラミングの高度なトピック

スレッド・セーフティ	9-2
OCI スレッド・セーフティの利点	9-2
スレッド・セーフティと 3 層アーキテクチャ	9-2
マルチスレッド開発の基本概念	9-3
スレッド・セーフティの実装	9-3
マルチスレッドの例	9-4
OCIThread パッケージ	9-5
初期化および終了	9-6
非アクティブなスレッドの基本形	9-7
アクティブなスレッドの基本形	9-10
OCIThread パッケージの使用方法	9-11
OCIThread の使用例	9-11
接続プーリング	9-13
OCI 接続プーリングの概念	9-13
接続プーリングの OCI コール	9-15
スクロール・カーソルのパフォーマンスの向上	9-19
接続プーリングの例	9-19

セッション・プーリング	9-23
OCI セッション・プーリングの機能	9-23
同種セッション・プールおよび異種セッション・プール	9-24
セッション・プーリングでのタグの使用	9-24
セッション・プーリング用のハンドル	9-24
OCI セッション・プーリングの使用	9-25
セッション・プーリングの OCI コール	9-26
OCI セッション・プーリングの例	9-28
文キャッシュ	9-28
セッション・プーリングを使用しない文キャッシュ	9-28
セッション・プーリングを使用する文キャッシュ	9-29
文キャッシュの規則	9-29
文キャッシュのコード例	9-30
ユーザー定義コールバック関数	9-30
ユーザー・コールバックの登録	9-31
外部プロシージャからの OCI コールバック	9-41
アプリケーション・フェイルオーバー・コールバック	9-41
フェイルオーバー・コールバックの概要	9-41
フェイルオーバー・コールバック構造およびパラメータ	9-42
フェイルオーバー・コールバックの登録	9-43
フェイルオーバー・コールバックの例	9-43
OCI_FO_ERROR の処理	9-45
OCI およびアドバンスト・キューイング	9-48
OCI アドバンスト・キューイング関数	9-49
OCI アドバンスト・キューイングの説明	9-49
OCI のアドバンスト・キューイングと PL/SQL	9-49
パブリッシュ・サブスクライブの通知	9-52
パブリッシュ・サブスクライブの登録関数	9-53
通知コールバック	9-58
通知プロシージャ	9-59
パブリッシュ・サブスクライブの直接登録の例	9-60
パブリッシュ・サブスクライブの LDAP 登録の例	9-65

第 II 部 OCI オブジェクトの概念

10 OCI オブジェクト・リレーショナル・プログラミング

OCI オブジェクトの概要	10-2
OCI でのオブジェクトの操作	10-3
基本的なオブジェクト・プログラム構造体	10-3
永続オブジェクト、一時オブジェクトおよび値	10-5
OCI オブジェクト・アプリケーションの開発	10-7
C アプリケーションでのオブジェクトの表現	10-8
環境およびオブジェクト・キャッシュの初期化	10-9
データベース接続の実行	10-10
サーバーからのオブジェクト参照の取出し	10-10
オブジェクトの確保	10-11
オブジェクト属性の操作	10-13
オブジェクトのマークおよび変更のフラッシュ	10-14
埋込みオブジェクトのフェッチ	10-15
オブジェクトのメタ属性	10-16
複合オブジェクト検索	10-20
COR ブリフェッチ	10-25
OCI と SQL のオブジェクトへのアクセス	10-28
確保カウントおよび確保解除	10-29
NULL かどうか	10-30
オブジェクトの作成	10-32
オブジェクトの解放およびコピー	10-34
オブジェクト参照と型参照	10-35
オブジェクト・ビューまたはユーザー定義 OID に基づいたオブジェクトの作成	10-35
オブジェクト・アプリケーションでのエラー処理	10-36
型の継承	10-36
代入性	10-38
NOT INSTANTIABLE の型とメソッド	10-38
OCI での型の継承のサポート	10-39
OTT での型の継承のサポート	10-40
型の変更	10-41

11 オブジェクト・リレーショナル・データ型

オブジェクトに対する OCI 関数の概要	11-2
Oracle データ型の C へのマッピング	11-3
OCI 型マッピングの方法論	11-4
OCI での C データ型の操作	11-4
Oracle 数値操作の精度	11-6
日付 (OCIDate)	11-6
日付変換関数	11-6
日付割当ておよび日付検索関数	11-7
日付算術および日付比較関数	11-7
日付情報アクセッサ関数	11-7
日付の妥当性チェック関数	11-8
日付の例	11-8
日時および時間隔 (OCIDateTime、OCIInterval)	11-10
日時関数	11-10
日時の例	11-12
時間隔関数	11-13
数値 (OCINumber)	11-14
数値算術関数	11-14
数値変換関数	11-15
指数関数および対数関数	11-15
三角関数	11-16
数値の代入、比較および評価関数	11-16
数値の例	11-17
固定長または可変長文字列 (OCIStrng)	11-19
文字列関数	11-19
文字列の例	11-19
ロー (OCIRaw)	11-20
ロー関数	11-20
ローの例	11-21
コレクション (OCITable、OCIArray、OCIColl、OCIIter)	11-21
汎用コレクション関数	11-22
コレクション・データ操作関数	11-22
コレクション・スキャン関数	11-23
VARRAY/ コレクション・イテレータの例	11-23

NESTED TABLE 操作関数	11-25
NESTED TABLE のロケータ	11-26
マルチレベル・コレクション型	11-26
マルチレベル・コレクション型の例	11-27
REF (OCIRef)	11-28
REF 操作関数	11-28
REF の例	11-28
オブジェクト型情報の格納およびアクセス	11-29
記述子オブジェクト	11-29
AnyType インタフェース、AnyData インタフェースおよび AnyDataSet インタフェース	11-30
型インタフェース	11-30
OCIAnyData インタフェース	11-34
OCIAnyData 関数の NCHAR 型コード	11-34
OCIAnyDataSet インタフェース	11-35
名前付きデータ型のバインド	11-36
名前付きデータ型のバインド	11-36
REF のバインド	11-37
名前付きデータ型および REF バインドの情報	11-37
名前付きデータ型の定義	11-38
名前付きデータ型出力変数の定義	11-38
REF 出力変数の定義	11-39
名前付きデータ型、REF 定義および PL/SQL OUT バインドの情報	11-39
Oracle C データ型のバインドおよび定義	11-41
バインドおよび定義の例	11-42
給与更新の例	11-44
SQLT_NTY のバインドおよび定義の例	11-47
バインドの例	11-47
定義の例	11-48

12 ダイレクト・パス・ロード

ダイレクト・パス・ロードの概要	12-2
ダイレクト・パス・ロードでサポートされるデータ型	12-4
ダイレクト・パス・ハンドル	12-5
ダイレクト・パス・インタフェースの関数	12-8
ダイレクト・パス・ロード・インタフェースに関する制限と制約	12-9

スカラー列に対するダイレクト・パス・ロードの例	12-9
OCI のダイレクト・パス・ロードでの日付キャッシュの使用	12-14
オブジェクト型のダイレクト・パス・ロード	12-16
NESTED TABLE のダイレクト・パス・ロード	12-16
列オブジェクトのダイレクト・パス・ロード	12-17
SQL 文字列の列のダイレクト・パス・ロード	12-20
REF 列のダイレクト・パス・ロード	12-23
NOT FINAL オブジェクトと REF 列	12-28
オブジェクト表のダイレクト・パス・ロード	12-29
NOT FINAL オブジェクト表のダイレクト・パス・ロード	12-30
ピース単位のダイレクト・パス・ロード	12-31
オブジェクト型のピース単位のロード	12-32
ダイレクト・パス・コンテキスト・ハンドルとオブジェクト型の属性	12-33
ダイレクト・パス・コンテキストの属性	12-33
ダイレクト・パス関数コンテキストと属性	12-33
ダイレクト・パス列パラメータの属性	12-37
非スカラー列のダイレクト・パス関数列配列ハンドル	12-40

13 オブジェクトのキャッシュおよびナビゲーション

オブジェクト・キャッシュおよびメモリー管理	13-2
キャッシュの一貫性	13-5
オブジェクト・キャッシュ・パラメータ	13-5
オブジェクト・キャッシュ操作	13-6
オブジェクト・コピーのロードと削除	13-7
オブジェクト・コピーの変更	13-10
オブジェクト・コピーとサーバーとの同期化	13-11
オブジェクトのロック	13-13
オブジェクト・アプリケーションでのコミットおよびロールバック	13-15
オブジェクト継続時間	13-15
インスタンスのメモリー・レイアウト	13-17
オブジェクト・ナビゲーション	13-18
単純なオブジェクト・ナビゲーション	13-18
OCI ナビゲーション関数	13-20
確保 / 確保解除 / 解放関数	13-21
フラッシュおよびリフレッシュ関数	13-21

マークおよびマーク解除関数	13-22
オブジェクトのメタ属性アクセッサ関数	13-22
その他の関数	13-23
型の変更とオブジェクト・キャッシュ	13-23

14 Object Type Translator (OTT)

OTT の概要	14-2
Object Type Translator の使用方法	14-2
データベースでの型の作成	14-5
OTT の起動	14-5
OTT のコマンドライン	14-6
OTT コマンドライン起動の例	14-6
Intype ファイル	14-8
OTT データ型マッピング	14-10
オブジェクト・データ型の C へのマッピング	14-11
OTT 型マッピングの例	14-12
NULL インジケータ構造体	14-15
OTT による型の継承のサポート	14-16
Outtype ファイル	14-20
OCI アプリケーションでの OTT の使用方法	14-21
OCI でのオブジェクトへのアクセスおよび操作	14-23
初期化関数のコール	14-23
初期化関数の作業	14-25
OTT リファレンス	14-25
OTT コマンドラインの構文	14-26
OTT パラメータ	14-27
OTT パラメータの指定可能な場所	14-32
Intype ファイルの構造体	14-33
ネストされたインクルード・ファイルの生成	14-35
SCHEMA_NAMES の使用方法	14-37
デフォルトの名前のマッピング	14-39
OTT でのファイル名比較の制限事項	14-40

第 III 部 OCI リファレンス

15 OCI リレーショナル関数

リレーショナル関数の概要	15-2
関数の構文	15-2
OCI 関数のコール	15-3
LOB 関数用のサーバー・ラウンドトリップ	15-3
接続関数、認証関数および初期化関数	15-4
OCIConnectionPoolCreate()	15-5
OCIConnectionPoolDestroy()	15-8
OCIEnvCreate()	15-9
OCIEnvInit()	15-12
OCIEnvNlsCreate()	15-14
OCIInitialize()	15-18
OCILogoff()	15-21
OCILogon()	15-22
OCILogon2()	15-24
OCIServerAttach()	15-27
OCIServerDetach()	15-29
OCISessionBegin()	15-30
OCISessionEnd()	15-34
OCISessionGet()	15-35
OCISessionPoolCreate()	15-39
OCISessionPoolDestroy()	15-43
OCISessionRelease()	15-44
OCITerminate()	15-46
ハンドル関数および記述子関数	15-47
OCIAttrGet()	15-48
OCIAttrSet()	15-50
OCIDescriptorAlloc()	15-52
OCIDescriptorFree()	15-54
OCIHandleAlloc()	15-56
OCIHandleFree()	15-59
OCIParmGet()	15-61
OCIParmSet()	15-63

バインド関数、定義関数および記述関数	15-64
OCIBindArrayOfStruct()	15-65
OCIBindByName()	15-66
OCIBindByPos()	15-71
OCIBindDynamic()	15-75
OCIBindObject()	15-79
OCIDefineArrayOfStruct()	15-81
OCIDefineByPos()	15-82
OCIDefineDynamic()	15-86
OCIDefineObject()	15-89
OCIDescribeAny()	15-91
OCISstmtGetBindInfo()	15-94

16 その他の OCI リレーショナル関数

その他のリレーショナル関数の概要	16-2
関数の構文	16-2
OCI 関数のコール	16-3
LOB 関数用のサーバー・ラウンドトリップ	16-3
文関数	16-4
OCISstmtExecute()	16-5
OCISstmtFetch()	16-8
OCISstmtFetch2()	16-10
OCISstmtGetPieceInfo()	16-13
OCISstmtPrepare()	16-15
OCISstmtPrepare2()	16-17
OCISstmtRelease()	16-19
OCISstmtSetPieceInfo()	16-20
LOB 関数	16-22
OCIDurationBegin()	16-24
OCIDurationEnd()	16-25
OCILobAppend()	16-26
OCILobAssign()	16-28
OCILobCharSetForm()	16-30
OCILobCharSetId()	16-31
OCILobClose()	16-32
OCILobCopy()	16-34
OCILobCreateTemporary()	16-36

OCILobDisableBuffering()	16-38
OCILobEnableBuffering()	16-39
OCILobErase()	16-40
OCILobFileClose()	16-42
OCILobFileCloseAll()	16-43
OCILobFileExists()	16-44
OCILobFileGetName()	16-45
OCILobFileIsOpen()	16-47
OCILobFileOpen()	16-48
OCILobFileSetName()	16-49
OCILobFlushBuffer()	16-51
OCILobFreeTemporary()	16-53
OCILobGetChunkSize()	16-54
OCILobGetLength()	16-56
OCILobIsEqual()	16-57
OCILobIsOpen()	16-58
OCILobIsTemporary()	16-60
OCILobLoadFromFile()	16-61
OCILobLocatorAssign()	16-63
OCILobLocatorIsInit()	16-65
OCILobOpen()	16-67
OCILobRead()	16-69
OCILobTrim()	16-74
OCILobWrite()	16-76
OCILobWriteAppend()	16-81
アドバンスト・キューイング関数およびパブリッシュ・サブスクライブ関数	16-85
OCIAQDeq()	16-86
OCIAQEnq()	16-88
OCIAQListen()	16-100
OCISubscriptionDisable()	16-102
OCISubscriptionEnable()	16-103
OCISubscriptionPost()	16-104
OCISubscriptionRegister()	16-106
OCISubscriptionUnRegister()	16-109
ダイレクト・パス・ロード関数	16-110
OCIDirPathAbort()	16-111
OCIDirPathColArrayEntryGet()	16-112
OCIDirPathColArrayEntrySet()	16-114

OCIDirPathColArrayRowGet()	16-116
OCIDirPathColArrayReset()	16-117
OCIDirPathColArrayToStream()	16-118
OCIDirPathDataSave()	16-120
OCIDirPathFinish()	16-121
OCIDirPathFlushRow()	16-122
OCIDirPathLoadStream()	16-123
OCIDirPathPrepare()	16-124
OCIDirPathStreamReset()	16-125
スレッド管理関数	16-126
OCIThreadClose()	16-128
OCIThreadCreate()	16-129
OCIThreadHandleGet()	16-131
OCIThreadHndDestroy()	16-132
OCIThreadHndInit()	16-133
OCIThreadIdDestroy()	16-134
OCIThreadIdGet()	16-135
OCIThreadIdInit()	16-136
OCIThreadIdNull()	16-137
OCIThreadIdSame()	16-138
OCIThreadIdSet()	16-139
OCIThreadIdSetNull()	16-140
OCIThreadInit()	16-141
OCIThreadIsMulti()	16-142
OCIThreadJoin()	16-143
OCIThreadKeyDestroy()	16-144
OCIThreadKeyGet()	16-145
OCIThreadKeyInit()	16-146
OCIThreadKeySet()	16-147
OCIThreadMutexAcquire()	16-148
OCIThreadMutexDestroy()	16-149
OCIThreadMutexInit()	16-150
OCIThreadMutexRelease()	16-151
OCIThreadProcessInit()	16-152
OCIThreadTerm()	16-153
トランザクション関数	16-154
OCITransCommit()	16-155
OCITransDetach()	16-158

OCITransForget()	16-159
OCITransMultiPrepare()	16-160
OCITransPrepare()	16-161
OCITransRollback()	16-162
OCITransStart()	16-163
その他の関数	16-171
OCIBreak()	16-172
OCIErrorGet()	16-173
OCILdaToSvcCtx()	16-175
OCINlsEnvironmentVariableGet()	16-176
OCIPasswordChange()	16-178
OCIReset()	16-180
OCIRowidToChar()	16-181
OCIServerVersion()	16-182
OCISvcCtxToLda()	16-183
OCIUserCallbackGet()	16-184
OCIUserCallbackRegister()	16-186

17 OCI のナビゲーション関数と型関数

ナビゲーション関数および型関数の概要	17-2
オブジェクト型と存続期間	17-2
用語	17-3
関数の構文	17-4
ナビゲーション関数の戻り値	17-5
キャッシュ関数およびオブジェクト関数用のサーバー・ラウンドトリップ	17-5
ナビゲーション関数エラー・コード	17-5
OCI フラッシュまたはリフレッシュ関数	17-8
OCICacheFlush()	17-9
OCICacheRefresh()	17-11
OCIObjectFlush()	17-13
OCIObjectRefresh()	17-14
OCI オブジェクトおよびキャッシュへのマークまたはマーク解除関数	17-16
OCICacheUnmark()	17-17
OCIObjectMarkDelete()	17-18
OCIObjectMarkDeleteByRef()	17-19
OCIObjectMarkUpdate()	17-20

OCIObjectUnmark()	17-22
OCIObjectUnmarkByRef()	17-23
OCI のオブジェクト・ステータス取得関数	17-24
OCIObjectExists()	17-25
OCIObjectGetProperty()	17-26
OCIObjectIsDirty()	17-30
OCIObjectIsLocked()	17-31
その他の OCI オブジェクト関数	17-32
OCIObjectCopy()	17-33
OCIObjectGetAttr()	17-35
OCIObjectGetInd()	17-37
OCIObjectGetObjectRef()	17-38
OCIObjectGetTypeRef()	17-39
OCIObjectLock()	17-40
OCIObjectLockNoWait()	17-41
OCIObjectNew()	17-43
OCIObjectSetAttr()	17-47
OCI の確保関数、確保解除関数および解放関数	17-49
OCICacheFree()	17-50
OCICacheUnpin()	17-51
OCIObjectArrayPin()	17-52
OCIObjectFree()	17-54
OCIObjectPin()	17-56
OCIObjectPinCountReset()	17-59
OCIObjectPinTable()	17-60
OCIObjectUnpin()	17-62
OCI の型情報アクセッサ関数	17-64
OCITypeArrayByName()	17-65
OCITypeArrayByRef()	17-68
OCITypeByName()	17-70
OCITypeByRef()	17-73

18 OCI のデータ型マッピング関数および操作関数

データ型マッピング関数および操作関数の概要	18-2
関数の構文	18-2
データ型マッピング関数および操作関数の戻り値	18-3
その他の値を戻す関数	18-3

データ型マッピング関数および操作関数のサーバー・ラウンドトリップ回数	18-4
例	18-4
OCI コレクションおよびイテレータ関数	18-5
OCICollAppend()	18-6
OCICollAssign()	18-7
OCICollAssignElem()	18-9
OCICollGetElem()	18-11
OCICollIsLocator()	18-14
OCICollMax()	18-15
OCICollSize()	18-16
OCICollTrim()	18-18
OCIIterCreate()	18-19
OCIIterDelete()	18-20
OCIIterGetCurrent()	18-21
OCIIterInit()	18-22
OCIIterNext()	18-23
OCIIterPrev()	18-25
OCI の日付関数、日時関数および時間隔関数	18-27
OCIDateAddDays()	18-30
OCIDateAddMonths()	18-31
OCIDateAssign()	18-32
OCIDateCheck()	18-33
OCIDateCompare()	18-35
OCIDateDaysBetween()	18-36
OCIDateFromText()	18-37
OCIDateGetDate()	18-39
OCIDateGetTime()	18-40
OCIDateLastDay()	18-41
OCIDateNextDay()	18-42
OCIDateSetDate()	18-43
OCIDateSetTime()	18-44
OCIDateSysDate()	18-45
OCIDateToText()	18-46
OCIDateTimeAssign()	18-48
OCIDateTimeCheck()	18-49
OCIDateTimeCompare()	18-51
OCIDateTimeConstruct()	18-52
OCIDateTimeConvert()	18-54

OCIDateTimeFromArray()	18-55
OCIDateTimeFromText()	18-57
OCIDateTimeGetDate()	18-59
OCIDateTimeGetTime()	18-60
OCIDateTimeGetTimeZoneName()	18-62
OCIDateTimeGetTimeZoneOffset()	18-63
OCIDateTimeIntervalAdd()	18-64
OCIDateTimeIntervalSub()	18-65
OCIDateTimeSubtract()	18-66
OCIDateTimeSysTimeStamp()	18-67
OCIDateTimeToArray()	18-68
OCIDateTimeToText()	18-70
OCIDateZoneToZone()	18-72
OCIIntervalAdd()	18-74
OCIIntervalAssign()	18-75
OCIIntervalCheck()	18-76
OCIIntervalCompare()	18-78
OCIIntervalDivide()	18-79
OCIIntervalFromNumber()	18-80
OCIIntervalFromText()	18-81
OCIIntervalFromTZ()	18-83
OCIIntervalGetDaySecond()	18-84
OCIIntervalGetYearMonth()	18-86
OCIIntervalMultiply()	18-87
OCIIntervalSetDaySecond()	18-88
OCIIntervalSetYearMonth()	18-90
OCIIntervalSubtract()	18-91
OCIIntervalToNumber()	18-92
OCIIntervalToText()	18-93
OCI 数値関数	18-95
OCINumberAbs()	18-97
OCINumberAdd()	18-98
OCINumberArcCos()	18-99
OCINumberArcSin()	18-100
OCINumberArcTan()	18-101
OCINumberArcTan2()	18-102
OCINumberAssign()	18-103
OCINumberCeil()	18-104

OCINumberCmp()	18-105
OCINumberCos()	18-106
OCINumberDec()	18-107
OCINumberDiv()	18-108
OCINumberExp()	18-109
OCINumberFloor()	18-110
OCINumberFromInt()	18-111
OCINumberFromReal()	18-112
OCINumberFromText()	18-113
OCINumberHypCos()	18-115
OCINumberHypSin()	18-116
OCINumberHypTan()	18-117
OCINumberInc()	18-118
OCINumberIntPower()	18-119
OCINumberIsInt()	18-120
OCINumberIsZero()	18-121
OCINumberLn()	18-122
OCINumberLog()	18-123
OCINumberMod()	18-124
OCINumberMul()	18-125
OCINumberNeg()	18-126
OCINumberPower()	18-127
OCINumberPrec()	18-128
OCINumberRound()	18-129
OCINumberSetPi()	18-130
OCINumberSetZero()	18-131
OCINumberShift()	18-132
OCINumberSign()	18-133
OCINumberSin()	18-134
OCINumberSqrt()	18-135
OCINumberSub()	18-136
OCINumberTan()	18-137
OCINumberToInt()	18-138
OCINumberToReal()	18-139
OCINumberToText()	18-140
OCINumberTrunc()	18-142

OCI ロー関数	18-143
OCIRawAllocSize()	18-144
OCIRawAssignBytes()	18-145
OCIRawAssignRaw()	18-146
OCIRawPtr()	18-147
OCIRawResize()	18-148
OCIRawSize()	18-149
OCI REF 関数	18-150
OCIRefAssign()	18-151
OCIRefClear()	18-152
OCIRefFromHex()	18-153
OCIRefHexSize()	18-154
OCIRefsEqual()	18-155
OCIRefsIsNull()	18-156
OCIRefToHex()	18-157
OCI 文字列関数	18-158
OCIStringAllocSize()	18-159
OCIStringAssign()	18-160
OCIStringAssignText()	18-161
OCIStringPtr()	18-162
OCIStringResize()	18-163
OCIStringSize()	18-164
OCI 表関数	18-165
OCITableDelete()	18-166
OCITableExists()	18-167
OCITableFirst()	18-168
OCITableLast()	18-169
OCITableNext()	18-170
OCITablePrev()	18-171
OCITableSize()	18-172

19 OCI カートリッジ関数

外部プロシージャ関数およびカートリッジ・サービス関数の概要	19-2
関数の構文	19-2
リターン・コード	19-3
With_Context 型	19-3

カートリッジ・サービス – OCI 外部プロシージャ	19-4
OCIExtProcAllocCallMemory()	19-5
OCIExtProcRaiseExcp()	19-6
OCIExtProcRaiseExcpWithMsg()	19-7
OCIExtProcGetEnv()	19-8
カートリッジ・サービス – メモリー・サービス	19-9
OCIDurationBegin()	19-10
OCIDurationEnd()	19-11
OCIMemoryAlloc()	19-12
OCIMemoryResize()	19-14
OCIMemoryFree()	19-15
カートリッジ・サービス – コンテキストのメンテナンス	19-16
OCIContextSetValue()	19-17
OCIContextGetValue()	19-19
OCIContextClearValue()	19-20
OCIContextGenerateKey()	19-21
カートリッジ・サービス – パラメータ・マネージャ・インタフェース	19-22
OCIExtractInit()	19-23
OCIExtractTerm()	19-24
OCIExtractReset()	19-25
OCIExtractSetNumKeys()	19-26
OCIExtractSetKey()	19-27
OCIExtractFromFile()	19-29
OCIExtractFromStr()	19-30
OCIExtractToInt()	19-31
OCIExtractToBool()	19-32
OCIExtractToStr()	19-33
OCIExtractToOCINum()	19-34
OCIExtractToList()	19-35
OCIExtractFromList()	19-36
カートリッジ・サービス – ファイル I/O インタフェース	19-38
OCIFileObject	19-38
OCIFileInit()	19-39
OCIFileTerm()	19-40
OCIFileOpen()	19-41
OCIFileClose()	19-43
OCIFileRead()	19-44
OCIFileWrite()	19-45

OCIFileSeek()	19-46
OCIFileExists()	19-48
OCIFileGetLength()	19-49
OCIFileFlush()	19-50
カートリッジ・サービス – 文字列のフォーマット・インタフェース	19-51
OCIFormatInit()	19-52
OCIFormatTerm()	19-53
OCIFormatString()	19-54
フォーマット修飾子	19-56
書式コード	19-57
例	19-59

20 OCI の任意型関数および任意データ関数

任意型インタフェースおよび任意データ・インタフェースの概要	20-2
関数の構文	20-2
関数戻り値	20-3
OCI の型インタフェース関数	20-4
OCITypeAddAttr()	20-5
OCITypeBeginCreate()	20-6
OCITypeEndCreate()	20-8
OCITypeSetBuiltin()	20-9
OCITypeSetCollection()	20-10
OCI 任意データ・インタフェース関数	20-11
OCIAnyDataAccess()	20-12
OCIAnyDataAttrGet()	20-14
OCIAnyDataAttrSet()	20-17
OCIAnyDataBeginCreate()	20-20
OCIAnyDataCollAddElem()	20-22
OCIAnyDataCollGetElem()	20-24
OCIAnyDataConvert()	20-26
OCIAnyDataDestroy()	20-28
OCIAnyDataEndCreate()	20-29
OCIAnyDataGetCurrAttrNum()	20-30
OCIAnyDataGetType()	20-31
OCIAnyDataIsNull()	20-32
OCIAnyDataTypeCodeToSqlit()	20-33

OCI 任意データ・セット・インタフェース関数	20-34
OCIAnyDataSetAddInstance()	20-35
OCIAnyDataSetBeginCreate()	20-37
OCIAnyDataSetDestroy()	20-39
OCIAnyDataSetEndCreate()	20-40
OCIAnyDataSetGetCount()	20-41
OCIAnyDataSetGetInstance()	20-42
OCIAnyDataSetGetType()	20-43

第 IV 部 付録

A ハンドルおよび記述子の属性

規則	A-2
環境ハンドル属性	A-3
エラー・ハンドル属性	A-10
サービス・コンテキスト・ハンドル属性	A-11
サーバー・ハンドル属性	A-13
認証情報ハンドル	A-16
ユーザー・セッション・ハンドル属性	A-17
接続プール・ハンドル属性	A-21
セッション・プール・ハンドル属性	A-23
トランザクション・ハンドル属性	A-26
文ハンドル属性	A-27
バインド・ハンドル属性	A-34
定義ハンドル属性	A-37
記述ハンドル属性	A-40
パラメータ記述子属性	A-40
LOB ロケータ属性	A-41
複合オブジェクト属性	A-41
複合オブジェクト検索ハンドル属性	A-41
複合オブジェクト検索記述子の属性	A-42
アドバンスト・キューイング記述子の属性	A-43
OCIAQEnqOptions 記述子の属性	A-43
OCIAQDeqOptions 記述子の属性	A-44
OCIAQMsgProperties 記述子の属性	A-48
OCIAQAgent 記述子の属性	A-52
OCIServerDNs 記述子の属性	A-53

サブスクリプション・ハンドル属性	A-54
ダイレクト・パス・ロード・ハンドル属性	A-58
ダイレクト・パス・コンテキスト・ハンドル (OCIDirPathCtx) の属性	A-58
ダイレクト・パス関数のコンテキスト・ハンドル (OCIDirPathFuncCtx) 属性	A-64
ダイレクト・パスおよびダイレクト・パス関数の列配列ハンドル (OCIDirPathColArray)	
属性	A-66
ダイレクト・パス・ストリーム・ハンドル (OCIDirPathStream) 属性	A-67
ダイレクト・パス列パラメータ属性	A-68
プロセス・ハンドル属性	A-75

B OCI デモ・プログラム

C OCI 関数のサーバー・ラウンドトリップ

サーバー・ラウンドトリップの概要	C-2
リレーショナル関数のラウンドトリップ	C-2
LOB 関数のラウンドトリップ	C-3
オブジェクト関数およびキャッシュ関数のラウンドトリップ	C-5
記述操作のラウンドトリップ	C-6
データ型マッピング関数および操作関数によるラウンドトリップ	C-7
任意型関数および任意データ関数のラウンドトリップ回数	C-7
その他のローカル関数	C-8

索引

はじめに

Oracle Call Interface (OCI) は、Application Program Interface (API) の 1 つです。OCI を使用すると、C または C++ 言語で作成したアプリケーションで 1 つ以上の Oracle データベース・サーバーと対話できます。OCI では、SQL 文の処理およびオブジェクト操作など、Oracle データベース・サーバーで可能なすべてのデータベース操作をプログラム上で実行できます。

ここでは、次の項目について説明します。

- [対象読者](#)
- [このマニュアルの構成](#)
- [関連文書](#)
- [表記規則](#)

対象読者

『Oracle Call Interface プログラマーズ・ガイド』は、Oracle 環境で実行するための新規アプリケーションの開発または既存のアプリケーションの変換を行うプログラマのためのマニュアルです。OCI について包括的に説明しているため、システム・アナリストやプロジェクト・マネージャをはじめ、データベース・アプリケーションの開発に関心のある他のユーザーにも役に立ちます。

このマニュアルを効果的に活用するには、C 言語によるアプリケーション・プログラミングに関する実践知識、および SQL リレーショナル・データベース言語の知識が必要です。さらに、このマニュアルの一部の章では、オブジェクト指向プログラミングの基本概念の知識が必要です。

関連項目：

- SQL の詳細は、『Oracle9i SQL リファレンス』および『Oracle9i データベース管理者ガイド』を参照してください。
- Oracle データベースについての基本概念の詳細は、『Oracle9i データベース概要』を参照してください。
- Standard Edition と Enterprise Edition の違いおよび使用可能なすべての機能とオプションの詳細は、『Oracle9i データベース新機能』を参照してください。

このマニュアルの構成

『Oracle Call Interface プログラマーズ・ガイド』は 4 部で構成されています。各章および付録の内容は、次のとおりです。

第 I 部：OCI の概念

第 I 部（第 1 章～第 9 章）では、OCI を使用してプログラミングを行い、Oracle データベースのリレーショナル・データにアクセスできるスケーラブルなアプリケーション・ソリューションを作成する方法の概要について説明します。

第 1 章「概要およびアップグレード」

この章では、OCI の概要およびこのインタフェースを説明する上での特殊な用語や表記上の規則について説明します。今回のリリースから新たに追加された機能についても説明します。

第 2 章「OCI プログラミングの基本」

この章では、OCI プログラムを開発する上で必要な基本概念を説明します。OCI プログラムに組み込む必要のある基本的なステップと、エラー・メッセージを取り出して解読する方法について説明します。

第3章「データ型」

OCI を使用するには、Oracle の表とホスト・プログラムの変数の間でデータが変換される仕組みを理解する必要があります。この章では、Oracle 内部と外部のデータ型、およびデータ変換について説明します。

第4章「OCI での SQL 文の使用」

この章では、OCI を使用する SQL 文のステップについて説明します。

第5章「バインドと定義」

この章では、OCI バインド操作および定義操作について詳しく説明します。さらに、拡張バインド操作および拡張定義操作についても説明します。

第6章「スキーマ・メタデータの記述」

この章では、スキーマ・オブジェクトとそれに対応する要素の情報を、OCIDescribeAny() コールを使用して取得する方法を説明します。

第7章「LOB と FILE の操作」

この章では、LOB データ型、FILE データ型および一時 LOB データ型に対する OCI サポートについて説明します。

第8章「スケーラブルなプラットフォームの管理」

この章では、パスワード管理、セッション管理およびスレッド・セーフティについて説明します。

第9章「OCI プログラミングの高度なトピック」

この章では、OCI スレッド・サポート、ユーザーのコールバック、アプリケーション・フェイルオーバー・コールバック、アドバンスド・キューイング、パブリッシュ・サブスクライブの通知など、より高度な OCI プログラミングのトピックについて説明します。

第II部：OCI オブジェクトの概念

第II部（第10章～第14章）では、OCI を使用してオブジェクト・リレーショナル・データにアクセスするための OCI 機能について説明します。

第10章「OCI オブジェクト・リレーショナル・プログラミング」

この章では、OCI を使用して Oracle データベース・サーバー内のオブジェクトにアクセスする場合の概要について説明します。基本的なオブジェクトの概念とオブジェクト・ナビゲーション・アクセス、およびオブジェクト・リレーショナル・アプリケーションの基本構造についても説明します。

第 11 章「オブジェクト・リレーショナル・データ型」

この章では、OCI プログラミングで使用するオブジェクト・データ型の概要を説明します。また、Oracle データベース内のユーザー定義データ型と C のデータ型のマッピング、およびこのようなデータを操作する関数について説明します。さらに、これらの C マッピングを使用したバインドと定義についても説明します。

第 12 章「ダイレクト・パス・ロード」

この章では、ダイレクト・パス・ロード API を使用して、ファイルからスカラー列とオブジェクト列にデータ（スカラー、オブジェクト）をロードする方法を説明します。

第 13 章「オブジェクトのキャッシュおよびナビゲーション」

この章では、OCI を使用して Oracle データベース・サーバー内のオブジェクトにアクセスする場合の概要について説明します。また、オブジェクト・キャッシュについて説明し、サーバーから取り出されたオブジェクトを操作する OCI ナビゲーション・コールの使用方法についても説明します。

第 14 章「Object Type Translator (OTT)」

この章では、Object Type Translator を使用してデータベース・オブジェクト定義を C 構造体の表現に変換し、OCI アプリケーションで使用方法を説明します。

第 III 部：OCI リファレンス

第 III 部（第 15 章～第 20 章）では、OCI ライブラリ内にある OCI ファンクション・コールのリストを示しています。

第 15 章「OCI リレーショナル関数」

この章では、OCI のリレーショナル関数のリストを示し、構文、コメント、パラメータ記述やその他の役立つ情報について説明します。

第 16 章「その他の OCI リレーショナル関数」

第 15 章に続き、OCI のリレーショナル関数について説明します。この章では、文関数および LOB、スレッド、トランザクション管理などに使用する関数について説明します。

第 17 章「OCI のナビゲーション関数と型関数」

この章では、OCI ナビゲーション関数のリストを示し、構文、コメント、パラメータ記述やその他の役立つ情報について説明します。

第 18 章「OCI のデータ型マッピング関数および操作関数」

この章では、OCI のデータ型マッピングと操作関数のリストを示し、構文、コメント、パラメータ記述やその他の役立つ情報について説明します。

第 19 章「OCI カートリッジ関数」

この章では、外部プロシージャおよびカートリッジ機能で使用する特殊な OCI 関数について説明します。

第 20 章「OCI の任意型関数および任意データ関数」

この章では、OCI の任意型関数および任意データ関数について説明します。

第 IV 部：付録

第 IV 部（付録 A ～付録 C）には、OCI プログラミングに関する追加の参照情報が含まれています。

付録 A「ハンドルおよび記述子の属性」

この付録では、OCI コールによって設定や読取りができる OCI アプリケーション・ハンドルの属性について説明します。

付録 B「OCI デモ・プログラム」

この付録では、Oracle のインストールに含まれる重要な OCI デモ・プログラムの名前を記載しています。

付録 C「OCI 関数のサーバー・ラウンドトリップ」

この付録では、様々な OCI アプリケーションで必要と考えられるサーバー・ラウンドトリップ回数を示す表を記載しています。

このマニュアルの読み方

OCI には拡張機能が多いため、初心者のみでなく、経験のあるプログラマの方も第 I 部の概念的な説明を参照してください。

OCI の現行バージョンをよく理解している方およびオブジェクト機能に興味のある方は、第 I 部は軽く目を通して、第 II 部から読み始めることも可能です。

OCI 関数の構文やハンドル属性の説明などの参照情報は、第 III 部と第 IV 部を参照してください。

関連文書

リリース・ノート、インストレーション・マニュアル、ホワイト・ペーパーまたはその他の関連文書は、OTN-J（Oracle Technology Network Japan）に接続すれば、無償でダウンロードできます。OTN-Jを使用するには、オンラインでの登録が必要です。次の URL で登録できます。

<http://otn.oracle.co.jp/membership/>

OTN-J のユーザー名とパスワードを取得済みであれば、次の OTN-J Web サイトの文書セクションに直接接続できます。

<http://otn.oracle.co.jp/document/>

このマニュアルでは、Standard Edition 製品および Enterprise Edition 製品に組み込まれている OCI の特性および機能がすべて説明されているわけではありません。

Oracle C++ Call Interface

Oracle C++ Call Interface では、C++ プログラマ向けに C++ プログラム用の OCI 機能を提供し、（ユーザー定義型の）データベース・オブジェクトを C++ オブジェクトとして操作できます。

OCIに関するその他の参照先

OCIに関するその他の参照先は、次のとおりです。

関連項目：

- C++ Call Interface の詳細は、『Oracle C++ Call Interface プログラマーズ・ガイド』を参照してください。
- カートリッジ・サービスおよびデータ・カートリッジの開発に関連する OCI コールの詳細は、『Oracle9i Data Cartridge Developer's Guide』を参照してください。
- 各国語サポートおよびグローバリゼーション・サポートに関連する OCI コールの詳細は、『Oracle9i Database グローバリゼーション・サポート・ガイド』を参照してください。
- アドバンスト・キューイングに関連する OCI コールの詳細は、『Oracle9i アプリケーション開発者ガイド - アドバンスト・キューイング』を参照してください。
- XA ライブラリで OCI を使用する場合は、『Oracle9i アプリケーション開発者ガイド - 基礎編』を参照してください。
- OCI コールを使用した LOB の操作の詳細およびコード例は、『Oracle9i アプリケーション開発者ガイド - ラージ・オブジェクト』を参照してください。
- オブジェクト型の詳しい説明は、『Oracle9i アプリケーション開発者ガイド - オブジェクト・リレーショナル機能』を参照してください。

表記規則

このマニュアルで使用する表記規則は、次のとおりです。

...

コード例が省略されている部分は、省略記号で表記されます。構文の中の省略記号は、前の項目が繰り返されていることを示します。

固定幅フォント

SQL と C 言語のコード例、OCI 関数名、データベース・オブジェクト、パッケージ、ユーザー名、ファイル名およびディレクトリ名は、固定幅フォントで表記されます。構文例でも、固定幅フォントが使用されます。

固定幅フォントのイタリック

本文中に OCI パラメータおよびユーザー指定のデータ・フィールドがある場合は、固定幅フォントのイタリックが使用されます。これらの項目が表やリストで使用されるときは、イタリックで表記されません。

固定幅フォントの大文字

固定幅フォントの大文字は、SELECT または UPDATE など、SQL や PL/SQL のキーワードを参照する場合に使用されます。

関連項目： SQL および PL/SQL 用のキーワードおよび予約語のリストを確認するには、『Oracle9i SQL リファレンス』および『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

太字

太字は、**ub4**、**sword**、**OCINumber** などの C データ型の名前を識別する場合に使用されま
す。太字は、コード例の中で強調の目的で使用される場合もあります。

このマニュアルでは、読者の注意を促すため、一部の情報に特殊なテキスト・フォーマット
を使用しています。インデントされて太字のテキスト・ラベルで始まる段落は、特別な内容
です。このように表記される情報は、次のとおりです。

注意： この「注意」というフラグは、一般的な問題を回避するため、ま
たは概念を理解するために読者にとって特に重要な情報であることを示し
ています。また、アプリケーションが正常に動作するように、OCI プログ
ラマが注意する必要がある事項を示しています。

7.x アップグレードの注意： 「7.x アップグレードの注意」と記された項目
は、通常、OCI リリース 8.x 以上と OCI リリース 7.x に大きな違いがある
場合にプログラマが注意する必要がある事項です。

関連項目： 「関連項目」と記されたテキストは、このマニュアル内で現
在の説明内容に関する追加情報が記載されている他の項または他の文書を
示しています。

Oracle Call Interface の新機能

この章では、次の各項で Oracle Call Interface の新機能について説明します。

- [Oracle9i リリース 2 \(9.2\) での Oracle Call Interface の新機能](#)
- [Oracle9i リリース 1 \(9.0.1\) での Oracle Call Interface の新機能](#)
- [Oracle8i リリース 8.1.6 での Oracle Call Interface の新機能](#)

Oracle9i リリース 2 (9.2) での Oracle Call Interface の新機能

■ セッション・プーリング

関連項目：

- 9-23 ページ「セッション・プーリング」
- 15-4 ページ「接続関数、認証関数および初期化関数」
- A-23 ページ「セッション・プール・ハンドル属性」

■ 文キャッシュ

関連項目：

- 9-28 ページ「文キャッシュ」
- 15-4 ページ「接続関数、認証関数および初期化関数」

■ 任意データ関数の拡張機能

関連項目： 拡張機能

- 20-33 ページ「OCIAnyDataTypeCodeToSqlit()」
- 11-34 ページ「OCIAnyData 関数の NCHAR 型コード」

■ オブジェクトに対する NCHAR およびコードポイントのサポート

オブジェクトに、NCHAR、NCLOB および NVARCHAR2 の各属性を設定できます。

関連項目：

- 2-45 ページ「OCI グローバリゼーション・サポート」
- 表 14-1「オブジェクト型属性のオブジェクト・データ型マッピング」
- 10-2 ページ「OCI オブジェクトの概要」
- 10-33 ページの表 10-1「新しいオブジェクトの属性値」
- 表 14-1「オブジェクト型属性のオブジェクト・データ型マッピング」
- 15-14 ページ「OCIEnvNlsCreate()」
- 16-176 ページ「OCINlsEnvironmentVariableGet()」

- クライアント・キャラクタ・セットの制御

関連項目： 2-48 ページ「[OCI からクライアント・キャラクタ・セットの制御](#)」

- データベース・グローバリゼーション・サポート

関連項目： 2-51 ページ「[OCI データベースのグローバリゼーション・サポート関数](#)」

- date_cache を使用したダイレクト・ロード

関連項目：

- 12-14 ページ「[OCI のダイレクト・パス・ロードでの日付キャッシュの使用](#)」
- A-58 ページ「[ダイレクト・パス・コンテキスト・ハンドル \(OCIDirPathCtx\) の属性](#)」

- 新規 OTT オプション: URL

関連項目： 14-32 ページ「[URL](#)」

- このドキュメントの構成の変更

このマニュアルの印刷版の 2 分冊構成を保つために、第 15 章にあった「[文関数](#)」は第 16 章 (16-4 ページ) に移されました。

Oracle9i リリース 1 (9.0.1) での Oracle Call Interface の新機能

- LOB 出力変数の定義

説明も変更されています。

関連項目： 5-22 ページ「[LOB 出力変数の定義](#)」

- UTF-16 Unicode のサポート

説明も変更されています。

関連項目：

- 2-45 ページ「[OCI グローバリゼーション・サポート](#)」
- 5-34 ページ「[バインドおよび定義における文字変換の問題](#)」
- A-34 ページ「[バインド・ハンドル属性](#)」
- A-37 ページ「[定義ハンドル属性](#)」

■ **アドバンスト・キューイング**

パブリッシュ・サブスクライブ通知のインタフェースおよび OCI 関数の `OCISubscriptionRegister()` が変更されています。いくつかのサブスクリプション・ハンドル属性が変更または追加されています。また、パブリッシュ・サブスクライブのオープン登録が追加されています。

関連項目：

- 9-52 ページ「[パブリッシュ・サブスクライブの通知](#)」
- 9-53 ページ「[パブリッシュ・サブスクライブの登録関数](#)」
- 16-106 ページ「[OCISubscriptionRegister\(\)](#)」
- A-54 ページ「[サブスクリプション・ハンドル属性](#)」
- A-53 ページ「[OCIServerDNs 記述子の属性](#)」
- A-3 ページ「[環境ハンドル属性](#)」
- 9-65 ページ「[パブリッシュ・サブスクライブの LDAP 登録の例](#)」

■ **ダイレクト・パス・ロード**

オブジェクト列およびスカラー列へのデータのダイレクト・パス・ロードがサポートされています。ダイレクト・パス・ロードについては、オブジェクトとその使用方法について説明した後、第 12 章で説明します。オブジェクト・データ型のバインドと定義については、第 11 章の最後で説明しています。

関連項目：

- 第 12 章「[ダイレクト・パス・ロード](#)」
- A-58 ページ「[ダイレクト・パス・ロード・ハンドル属性](#)」

■ 接続プーリング

この機能によって、単一の物理接続で複数の論理接続を多重化できます。

関連項目：

- 9-13 ページ [「接続プーリング」](#)
- 15-4 ページ [「接続関数、認証関数および初期化関数」](#)
- A-21 ページ [「接続プール・ハンドル属性」](#)

■ スクロール・カーソル

結果セットのメンバーに、順不同でアクセスできます。

関連項目：

- 4-17 ページ [「スクロール・カーソル」](#)
- 16-4 ページ [「文関数」](#)

■ グローバリゼーション・サポート

様々な OCI コールで、SQL 文、データ、メタデータ、オブジェクトおよびエラー・メッセージについて UTF-16 がサポートされています。

関連項目： 2-45 ページ [「OCI グローバリゼーション・サポート」](#)

■ 中間層アプリケーション

クライアント認証に関する新規の属性が追加されています。

関連項目： 8-14 ページ [「中間層アプリケーション」](#)

■ 新規のデータ型

日時と時間隔および夏時間調整のデータ型については、次の各項で説明します。

関連項目：

- 3-22 ページ [「日時および時間隔のデータ型記述子」](#)
- 11-10 ページ [「日時および時間隔（OCIDateTime、OCIInterval）」](#)
- 3-27 ページ [「日時および時間隔データ型のデータ変換」](#)
- 18-27 ページ [「OCI の日付関数、日時関数および時間隔関数」](#)

- 任意型、AnyData および AnyDataSet

OCIAnyData は、型情報とその型のデータ・インスタンス（つまり、自己記述データ）をカプセル化します。OCIAnyDataSet は、型情報およびその型のインスタンス・セットをカプセル化します。

関連項目： 11-30 ページの「[AnyType インタフェース](#)、[AnyData インタフェース](#)および [AnyDataSet インタフェース](#)」、および対応する新機能については、[第 20 章「OCI の任意型関数および任意データ関数](#)」を参照してください。

- LONG 列のかわりに LOB 列を使用

関連項目： 5-44 ページの「[ランタイム・データ割当てとピース単位操作](#)」は、LOB をサポートするための新機能の説明に変更されています。

- オブジェクトのサブタイプが定義可能

関連項目：

- 10-36 ページ「[型の継承](#)」
- 14-16 ページ「[OTT による型の継承のサポート](#)」

- 型の変更

型の属性を変更できます。

関連項目： 10-41 ページ「[型の変更](#)」

- マルチレベル・コレクション型

要素がコレクションのコレクションです。

関連項目： 11-26 ページ「[マルチレベル・コレクション型](#)」

- 外部で初期化されたコンテキスト

外部で初期化されたコンテキストとは、属性を OCI から初期化できるアプリケーション・コンテキストです。

関連項目：

- 8-20 ページ「[外部で初期化されたコンテキスト](#)」
- A-17 ページ「[ユーザー・セッション・ハンドル属性](#)」

- このドキュメントの構成の変更
 - リリース 8.1.6 の第 15 章は、第 15 章と 16 章に分割されています。
 - 第 15 章と 16 章に含まれる項は、より論理的な順序に並べ替えられています。
 - 第 17 章、18 章、19 章は、リリース 8.1.6 の第 16 章、17 章、18 章に該当します。
 - 第 20 章が追加されています。

関連項目：

- 既存のルーチンを置き換える新しいコールの詳細は、1-18 ページの「[互換性およびアップグレード](#)」を参照してください。
- 新機能のエントリについては、目次および索引を参照してください。

Oracle8i リリース 8.1.6 での Oracle Call Interface の新機能

リリース 8.1 の OCI には次の新機能が備えられており、パフォーマンスが向上しています。

- メカニズムを改善したコールバック
- 中間層の認証属性の説明
- カートリッジ・サービス機能の説明
- NULL 以外の属性値を含む新規オブジェクトの作成機能
- ユニバーサル ROWID のサポート
- 固定幅 Unicode のサポート
- スレッド操作の OCIThread パッケージ
- ユーザー作成コールバック関数の登録機能
- アプリケーション・フェイルオーバー処理の拡張機能
- パブリッシュ / サブスクライブ通知のサポート
- オブジェクトに対する非待機ロック・オプション
- フラッシュ時にオブジェクトの変更を検出する機能
- 一時 LOB のサポート
- LOB サポートの拡張機能
- すべてのエラーをバッチに戻すことができる、配列 DML 文実行の拡張機能
- 拡張 DML...RETURNING サポート
- オブジェクト・ビューまたはユーザー作成のオブジェクト ID に基づくオブジェクトの作成機能

- 非ブロック化モードのサポート
- 機能およびパフォーマンスの拡張の追加
- クライアントのイベント通知用のパブリッシュ / サブスクライブ機能
- Oracle サーバーのダイレクト・ブロック・フォーマッタへのアクセスを提供する、ダイレクト・パス・ロード・コール
- 実行時のメモリー使用量の削減
- コードの削減による実行時パフォーマンスの向上
- フェッチ・プロトコルの合理化および効率化による、問合せのパフォーマンスの向上

第 I 部

OCI の概念

第 I 部は、OCI プログラミングの概念を説明した章で構成されています。

- [第 1 章「概要およびアップグレード」](#) では、OCI の概要を説明し、このリリースの新機能について説明します。
- [第 2 章「OCI プログラミングの基本」](#) では、OCI プログラミングの基本概念を説明します。
- [第 3 章「データ型」](#) では、OCI アプリケーションおよびサーバー内で使用されるデータ型について説明します。
- [第 4 章「OCI での SQL 文の使用」](#) では、OCI を使用した SQL 文の処理方法について説明します。
- [第 5 章「バインドと定義」](#) では、バインド操作および定義操作について詳しく説明します。
- [第 6 章「スキーマ・メタデータの記述」](#) では、`OCIDescribeAny()` 関数について説明します。
- [第 7 章「LOB と FILE の操作」](#) では、データベース内のラージ・オブジェクト (Large Object: LOB) および外部 LOB の操作を行う OCI 関数について説明します。
- [第 8 章「スケーラブルなプラットフォームの管理」](#) では、パスワード管理、セッション管理、中間層アプリケーションおよび外部で初期化されたコンテキストについて説明します。
- [第 9 章「OCI プログラミングの高度なトピック」](#) では、スレッド、接続プーリング、セッション・プーリング、ユーザー定義のコールバック、アドバンスド・キューイング、パブリッシュ・サブスクライブ通知など、OCI プログラミングのより高度なトピックについて説明します。

概要およびアップグレード

この章では、Oracle Call Interface (OCI) について説明します。OCI を使用したアプリケーションの開発に必要な基本情報を記載しています。また、OCI を説明するときに使用する特殊な用語についても解説します。互換性およびアップグレードについても説明します。

この章は、次の項目で構成されています。

- [OCI の概要](#)
- [互換性およびアップグレード](#)

OCI の概要

Oracle Call Interface (OCI) は、Application Program Interface (API) の 1 つです。OCI を使用すると、第三世代言語固有のプロシージャまたはファンクション・コールを使用して Oracle データベース・サーバーにアクセスし、SQL 文の実行のすべてのフェーズを制御するアプリケーションを作成できます。OCI では、C 言語と C++ 言語のデータ型、コール規則、構文およびセマンティックをサポートします。

関連項目：

- C++ Call Interface の詳細は、『Oracle C++ Call Interface プログラマーズ・ガイド』を参照してください。
- xxxvii ページ「[OCI に関するその他の参照先](#)」

OCI は次の機能を備えています。

- システム・メモリーおよびネットワーク接続の効率的な使用によるパフォーマンスと拡張性の向上
- 2 層クライアント / サーバーまたは複数層環境での動的セッションおよびトランザクション管理に適した一貫性のあるインタフェース
- N 層認証
- Oracle オブジェクトを使用したアプリケーション開発の包括的サポート
- 外部データベースへの接続
- ハードウェアへの追加投資をすることなく、ユーザー数および要求数の増加に対応できるアプリケーション

OCI では、C 言語などのホスト・プログラミング言語を使用して Oracle データベース内のデータおよびスキーマを操作できます。標準的なデータベース・アクセスと検索機能のライブラリが動的ランタイム・ライブラリ (OCI ライブラリ) の形で用意されており、実行時にアプリケーションにリンクできます。このため、3GL プログラム内に SQL や PL/SQL を埋め込む必要はありません。

OCI は多くの新機能を備えており、主に次のように分類できます。

- カプセル化されたインタフェース / 不透明（詳細が不明な）インタフェース
- ユーザー認証およびパスワード管理の簡素化
- アプリケーションのパフォーマンスおよび拡張性を向上させる拡張機能
- トランザクション管理に適した一貫性のあるインタフェース
- クライアント側からの Oracle オブジェクトに対するアクセスをサポートする OCI 拡張機能

OCI の利点

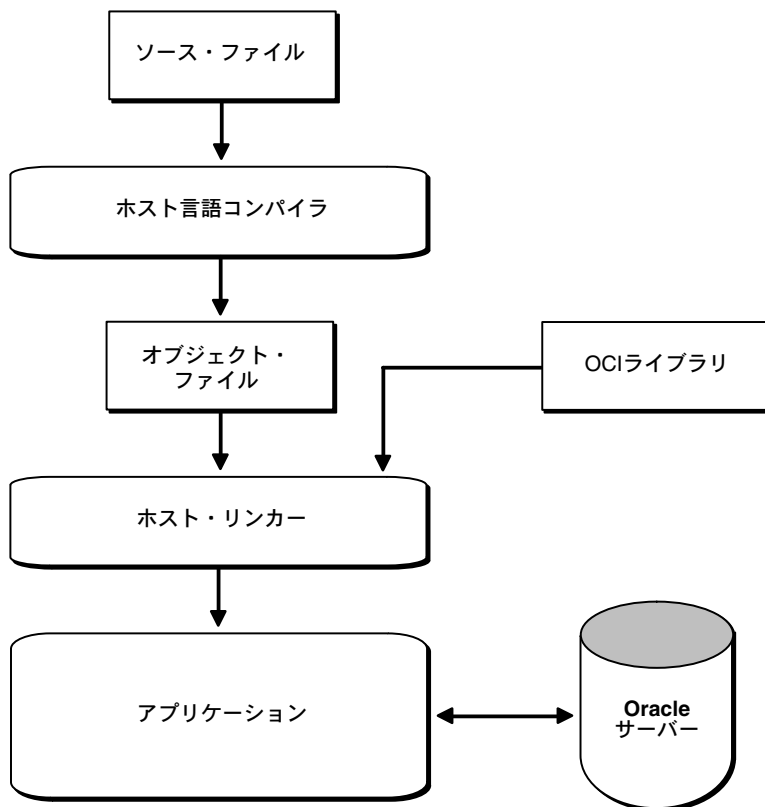
Oracle データベースに接続する場合、OCI には次のような大きな利点があります。

- アプリケーション設計のあらゆる側面におけるきめ細かい制御
- プログラム実行に対する高度な制御
- 使い慣れた 3GL プログラミング手法およびブラウザやデバッガなどのアプリケーション開発ツールの使用
- スケーラブルなアプリケーション作成を可能にする接続プーリング、セッション・プーリングおよび文キャッシュ
- 動的 SQL のサポート
- 様々なプラットフォームに対応した Oracle プログラム・インタフェースの可用性
- コールバックを使用した動的バインドおよび定義
- サーバー・メタデータのレイヤーを公開する記述機能
- 登録されたクライアント・アプリケーションへの非同期イベント通知
- 配列の挿入、更新、削除を行うための高度な配列データ操作言語（DML）機能
- コミット要求の実行への関連付けによるラウンドトリップ回数の削減
- 透過的プリフェッチ・バッファを使用した問合せの最適化によるラウンドトリップ回数の削減
- OCI ハンドルに対する相互排他ロック（mutex）を不要にするスレッド・セーフティ

OCI アプリケーションの構築

図 1-1 に示すように、OCI プログラムは非データベース・アプリケーションと同様の方法でコンパイルおよびリンクできます。処理前のステップやプリコンパイル・ステップを別途行う必要はありません。

図 1-1 OCI の開発過程



Oracle では、一般に普及しているほとんどのサード・パーティのコンパイラをサポートしています。OCI プログラムのリンクの詳細は、システムによって異なります。一部のプラットフォームでは、OCI プログラムを正常にリンクさせるために、OCI ライブラリに加えて他のライブラリの組み込みが必要になる場合があります。特定のプラットフォームでの OCI アプリケーションのコンパイルとリンクの詳細は、使用システムに関する Oracle マニュアルとインストール・ガイドを参照してください。

OCI の分類

OCI は次の機能を備えています。

- 多数のユーザーを安全にサポートできるスケーラブルなマルチスレッド・アプリケーションを設計するための API
- データベース・アクセス、SQL 文の処理および Oracle データベース・サーバーから取り出したオブジェクトの操作を行うための SQL アクセス関数
- Oracle 型のデータ属性を操作するためのデータ型マッピングおよび操作関数
- SQL 文を使用せずにデータベースにデータを直接ロードするためのデータ・ロード関数
- PL/SQL から C 言語のコールバックを書き込むための外部プロシージャ関数

手続き型および非手続き型要素

OCI を使用すると、SQL の非プロシージャ型データ・アクセス機能と、C や C++ 言語の手続き型機能をあわせ持つ複数層アーキテクチャ上で、スケーラブルなマルチスレッド・アプリケーションを開発できます。

- 非手続き型言語のプログラムでは、操作対象となる一連のデータが指定されますが、実行する操作の種類や操作の方法は指定されません。SQL は非手続き型という性質上、習得しやすく、データベース・トランザクションを実行しやすい言語です。また、SQL は、先進のリレーショナル・データベース・システムおよびオブジェクト・リレーショナル・データベース・システムのデータに対するアクセスや操作に使用する標準言語となっています。
- 手続き型言語のプログラムでは、大部分の文の実行は、前または後続の文、およびループや条件ブランチなど SQL では利用できない制御構造に依存しています。手続き型という性質上、このような言語は SQL よりも複雑ですが、柔軟性に富み、非常に強力です。

OCI プログラムでは、非手続き型言語と手続き型言語の要素が組み合されているため、構造化プログラミング環境で Oracle データベースに容易に接続できます。

OCI では、Oracle データベース・サーバーを通じて使用可能な SQL のデータ定義、データ操作、問合せおよびトランザクション制御の機能をすべてサポートしています。たとえば、OCI プログラムでは、Oracle データベースに対する問合せを実行できます。問合せ文では、次の例のように、入力（バインド）変数を使用して、データベースにデータを渡すようにプログラムに対して指示できます。

```
SELECT name FROM employees WHERE empno = :empnumber
```

前述の SQL 文の :empnumber は、アプリケーションが指定する値に対するプレースホルダです。

また、Oracle が開発した SQL の手続き型拡張要素である PL/SQL も利用できます。PL/SQL により、SQL のみで作成されたアプリケーションよりも、さらに強力で柔軟性のあるアプリケーションを開発できます。OCI は、Oracle データベース・サーバー内のオブジェクトに対するアクセスおよび操作のための機能も提供します。

オブジェクトのサポート

OCI には、オブジェクト型とオブジェクトを処理する機能があります。オブジェクト型は、ユーザー定義のデータ構造体で、現実社会の実体を抽象的に表します。たとえば、データベースに `person`（個人）というオブジェクトの定義が含まれているとします。このオブジェクトは、個人を識別するための特徴を表した属性 `first_name`、`last_name` および `age` を持つことができます。

- オブジェクト型の定義は、オブジェクト型のインスタンスを表すオブジェクトの作成の基礎となります。オブジェクト型を構造的な定義として使用することで、たとえば、ある `person` オブジェクトを属性 `'John'`、`'Bonivento'` および `'30'` から作成することができます。オブジェクト型にはメソッド、つまりそのオブジェクト型の動作を表現するプログラム関数を含めることもできます。

関連項目： オブジェクト型およびオブジェクトの詳細は、『Oracle9i データベース概要』および『Oracle9i アプリケーション開発者ガイド - オブジェクト・リレーショナル機能』を参照してください。

OCI には、Oracle データベース・サーバー内のオブジェクト処理用に OCI の機能を拡張した関数が組み込まれています。具体的には、次の機能が OCI に追加されました。

- オブジェクト・データとスキーマ情報を操作する SQL 文の実行のサポート
- SQL 文の入力変数としてオブジェクト参照およびオブジェクトのインスタンスを渡す機能のサポート
- SQL 文の出力を受け取る変数として、オブジェクト参照およびオブジェクトのインスタンスの宣言のサポート
- データベースからオブジェクト参照およびオブジェクトのインスタンスのフェッチのサポート
- オブジェクトのインスタンスと参照を戻す SQL 文のプロパティの記述のサポート
- オブジェクト・パラメータまたは結果を指定した PL/SQL プロシージャまたはファンクションの記述のサポート
- オブジェクトとリレーショナル機能を同期化するための、コミット・コールとロールバック・コールの拡張

その他の OCI コールは、SQL 文を通してアクセスされた後、オブジェクトの操作をサポートするために提供されます。拡張機能および新機能の詳細は、1-12 ページの「[カプセル化されたインタフェース](#)」を参照してください。

SQL 文

OCI アプリケーションの主なタスクの 1 つは、SQL 文を処理することです。SQL 文の種類が異なると、プログラムでは異なる処理ステップが必要になります。OCI アプリケーションをコーディングする際には、そのことを考慮に入れることが重要です。Oracle では、次の種類の SQL 文を認識します。

- [データ定義言語 \(DDL\)](#)
- [制御文](#)
 - [トランザクション制御](#)
 - [セッション制御](#)
 - [システム制御](#)
- [データ操作言語 \(DML\)](#)
- [問合せ](#)

注意： 多くの場合、問合せは DML 文として分類されますが、OCI アプリケーションでは異なる方法で問合せを処理するため、ここでは両者を別個に考慮しています。

- [PL/SQL](#)
- [埋込み SQL](#)

関連項目： [第 4 章「OCI での SQL 文の使用」](#)

データ定義言語

データ定義言語 (DDL) 文は、データベース内のスキーマ・オブジェクトを管理します。DDL 文は、新規の表を作成し、古い表を削除し、その他のスキーマ・オブジェクトを設定します。また、スキーマ・オブジェクトに対するアクセスを制御します。

表を作成し、その表へのアクセスを指定する例を次に示します。

```
CREATE TABLE employees
  (name      VARCHAR2(20),
   ssn       VARCHAR2(12),
   empno     NUMBER(6),
   mgr       NUMBER(6),
   salary    NUMBER(6))
```

```
GRANT UPDATE, INSERT, DELETE ON employees TO donna
REVOKE UPDATE ON employees FROM jamie
```

DDL 文によって、Oracle データベース内のオブジェクトを操作することもできます。次の例は、オブジェクト表を作成する一連の文です。

```
CREATE TYPE person_t AS OBJECT (  
    name      VARCHAR2(30),  
    ssn       VARCHAR2(12),  
    address   VARCHAR2(50))  
  
CREATE TABLE person_tab OF person_t
```

制御文

- OCI アプリケーションは、トランザクション制御文およびセッション制御文、システム制御文を DML 文のように処理します。

関連項目： これらの文の詳細は、『Oracle9i SQL リファレンス』を参照してください。

データ操作言語

データ操作言語（DML）文は、データベース表にあるデータを変更できます。たとえば、DML 文を使用して次の処理を実行します。

- 表に新しい行を挿入します。
- 既存の行の列値を更新します。
- 表から行を削除します。
- データベース内の表をロックします。
- SQL 文の実行計画を説明します。
- DML 文では、入力（バインド）変数を使用して、データベースにデータを渡すようにアプリケーションに対して指示できます。

関連項目： 入力バインド変数の詳細は、4-6 ページの「[バインド](#)」を参照してください。

また、DML 文では Oracle データベース内のオブジェクトを操作できます。次の例では、person_t 型のインスタンスがオブジェクト表 person_tab に挿入されます。

```
INSERT INTO person_tab  
VALUES (person_t('Steve May','123-45-6789','146 Winfield Street'))
```

問合せ

問合せは、データベースからデータを取り出すための文です。1つの問合せで、0、1行および複数行のデータを戻すことができます。すべての問合せは、次の例のように、SQL キーワード `SELECT` で始まります。

```
SELECT dname FROM dept
      WHERE deptno = 42
```

問合せは、表の中のデータにアクセスするため、多くの場合 `DML` 文として分類されます。ただし、OCI アプリケーションでは問合せの処理方法が異なるため、このマニュアルでは問合せを別個のものとして扱います。

問合せ文では入力（バインド）変数を使用して、データベースにデータを渡すようにプログラムに指示できます。たとえば次のとおりです。

```
SELECT name
      FROM employees
      WHERE empno = :empnumber
```

前述の SQL 文の `:empnumber` は、アプリケーションが指定する値に対するプレースホルダです。

- 問合せを処理する際、OCI アプリケーションでは、戻される結果を受け取るための出力変数も定義する必要があります。上の文では、問合せから戻される `name` の値を受け取るために出力変数を定義する必要があります。

関連項目：

- 入力バインド変数の詳細は、5-2 ページの「[バインド](#)」を参照してください。出力変数の定義の詳細は、5-18 ページの「[定義](#)」を参照してください。
- OCI プログラム内での SQL 文の処理方法の詳細は、第 4 章「[OCI での SQL 文の使用](#)」を参照してください。

PL/SQL

PL/SQL は、Oracle が開発した SQL 言語の手続き型拡張要素です。PL/SQL は、単純な問合せや SQL データ操作言語文よりも複雑なタスクを処理します。PL/SQL を使用すると、多数の構文を単一のブロックにグループ化し、1 つの単位として実行できます。これらの単位には次の部分が含まれます。

- 1 つ以上の SQL 文
- 変数宣言
- 代入文
- プロシージャ型制御文 (IF...THEN...ELSE 文とループ)
- 例外処理

OCI プログラムで PL/SQL ブロックを使用すると、次の処理を実行できます。

- Oracle ストアド・プロシージャおよびストアド・ファンクションのコール
- プロシージャ型制御文を複数の SQL 文と結合し、1 つの単位として実行
- レコード、表、カーソル FOR ループ、例外処理などの特殊な PL/SQL 機能へのアクセス
- カーソル変数の使用
- Oracle データベース・サーバー内のオブジェクトへのアクセスおよび操作

次の PL/SQL 例では、特定の従業員番号をキーにして、従業員表から値を取り出す SQL 文が発行されます。この例は、PL/SQL 文でのプレースホルダの使用方法も示しています。

```
BEGIN
    SELECT ename, sal, comm INTO :emp_name, :salary, :commission
    FROM emp
    WHERE ename = :emp_number;
END;
```

- この文のプレースホルダは、PL/SQL 変数ではないことに注意してください。これらは、文の処理時に、Oracle に渡される入力値を示しています。これらのプレースホルダは、プログラム内で C 言語変数にバインドする必要があります。

関連項目：

- PL/SQL ブロックのコーディングの詳細は、『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。
- PL/SQL でのプレースホルダの使用方法の詳細は、5-5 ページの「[PL/SQL のプレースホルダのバインド](#)」を参照してください。

埋込み SQL

OCI は、アプリケーションが実行時に Oracle に渡すテキスト文字列として SQL 文を処理します。Oracle プリコンパイラ (Pro*C/C++, Pro*COBOL、Pro*FORTRAN) を使用することによって、SQL 文をアプリケーション・コードに直接埋め込むことができます。その後、実行可能なアプリケーションを生成するために別個のプリコンパイル・ステップが必要です。

- OCI コールと埋込み SQL はプリコンパイラ・プログラムでは混在できません。

関連項目： 詳細は、『Pro*C/C++ Precompiler プログラマーズ・ガイド』を参照してください。

OCI/SQL の特殊用語

このマニュアルでは、SQL 文の様々な部分を参照するために特殊な用語を使用しています。たとえば次のような SQL 文があります。

```
SELECT customer, address
FROM customers
WHERE bus_type = 'SOFTWARE'
AND sales_volume = :sales
```

この SQL 文は、次の部分で構成されています。

- SQL コマンド - SELECT
- 2つの選択リスト項目 - customer および address
- FROM 句内の表名 - customers
- WHERE 句内の2つの列名 - bus_type および sales_volume
- WHERE 句内のリテラル入力値 - 'SOFTWARE'
- WHERE 句内の入力変数用のプレースホルダ - :sales

OCI アプリケーションを開発する際は、プログラム中の入出力変数のアドレス (位置) を Oracle データベース・サーバーに指定しているルーチンをコールします。このマニュアルでは、データ入力用のプレースホルダ変数のアドレスを指定することをバインド操作と呼びます。また、選択リスト項目を受け取る変数のアドレスを指定することを定義操作と呼びます。

PL/SQL では、入力の指定も出力の指定もバインド操作と呼びます。これらの用語と操作の詳細は、[第4章「OCI での SQL 文の使用」](#)を参照してください。

カプセル化されたインタフェース

OCI で使用されるすべてのデータ構造は、ハンドルと呼ばれる不透明なインタフェースの形にカプセル化されます。ハンドルとは、OCI ライブラリによって割り当てられる記憶領域を指す不透明なポインタで、SQL 文または PL/SQL 文のコンテキスト情報、接続情報、エラー情報またはバインド情報を格納します。クライアントでは、特定の種類のハンドルを割り当て、正しく定義されたインタフェースを通じて 1 つ以上のハンドルを移入し、それらのハンドルを使用してサーバーに要求を送信します。その結果、アプリケーションは、アクセス関数を使用してそのハンドルに含まれている特定の情報にアクセスできます。OCI ライブラリでは、ハンドルの階層が管理されます。これらのハンドルを使用して OCI インタフェースをカプセル化することには、アプリケーション開発者にとって次のような利点があります。

- 保持する必要があるサーバー側の状態情報の量が少なくなり、サーバー側のメモリ使用量を削減できます。
- グローバル変数を使用する必要がなくなること、エラー・レポートを作成しやすくなること、OCI 変数のアクセスおよび使用方法に一貫性を保てることにより、アプリケーション開発者の生産性が向上します。
- OCI 構造をハンドルの形にカプセル化することにより OCI 構造が不透明になるため、アプリケーションに影響を与えずに基礎となる構造を変更できます。

ユーザー認証およびパスワード管理の簡素化

OCI では、アプリケーション開発者が、次のような簡素化されたユーザー認証およびパスワード管理を実行できます。

- 単一の OCI アプリケーションで、複数のユーザーを認証およびメンテナンスできます。
- OCI アプリケーションで、ユーザーのパスワードを更新できます。これは、認証の処理でパスワード期限切れのメッセージが戻された場合に特に役立ちます。

OCI では、次の 2 種類のログイン・セッションをサポートします。

- シングル・ユーザーがログイン名とパスワードを使用してデータベースに接続するセッションのための、簡易ログイン関数。
- ログイン・セッションを分割することにより、単一の OCI アプリケーションで複数のセッションの認証とメンテナンスを行う設定。ログイン・セッションは、ユーザーが自分で作成したその他すべてのセッションであるユーザー・セッションから、Oracle データベースにログインするときに作成されるセッションです。これはリリース 7.3 との重要な違いです。リリース 7.3 では、ユーザーがデータベースにログインすると、新しいトランザクションを開始することにより、セッションは暗黙的に作成されました。このプロセスはセッション・クローニングと呼ばれます。リリース 7.3 のこれらのユーザー・セッションでは、ログイン・セッションからの権限およびセキュリティ・コンテキストが継承されていました。OCI では、クライアントが、各ユーザー・セッションに必要なすべての認証情報を提供する必要があります。これにより、OCI アプリケーションでは複数のユーザーをサポートできます。

アプリケーションのパフォーマンスおよび拡張性を改善する拡張機能

OCI には、アプリケーションのパフォーマンスおよび拡張性を改善する、いくつかの拡張機能があります。クライアントとサーバー間のラウンドトリップ回数を減らすことにより、アプリケーションのパフォーマンスが改善されました。さらに、サーバー側で保持する必要がある状態情報の量を減らすことにより拡張性も向上しています。これらの機能には、次のものがあります。

- クライアント側の処理の増加、およびサーバー側の問合せ処理の削減
- 記述ラウンドトリップをふり落とし、ラウンドトリップ回数およびメモリー使用量を削減する、`SELECT` 文結果セットの暗黙的なプリフェッチ
- オープン・カーソルおよびクローズ・カーソルのラウンドトリップの排除
- マルチスレッド環境のサポートの改善
- 接続上のセッション多重化
- 標準の 2 層クライアント / サーバー構成、サーバー / サーバー・トランザクション調整、3 層 TP モニター構成など、様々な構成に対する一貫したサポート
- XA インタフェースの `TM_JOIN` 操作のサポートを含む、ローカル・トランザクションおよびグローバル・トランザクションに対する一貫したサポート
- 接続中のユーザー接続、処理およびセッションを集結する機能を提供し、グローバル・トランザクションのブランチごとに個別のセッションを作成する必要性をなくしたことによる、拡張性の向上
- アプリケーションによる複数のユーザーの認証、および複数のユーザーのためのトランザクションの開始

OCI によるオブジェクトのサポート

OCI は、Oracle サーバーのオブジェクト機能を使用するプログラマに、最も包括的な Application Program Interface (API) を提供します。その機能は、次の 6 つの主要なカテゴリに分類できます。

- クライアント側のオブジェクト・キャッシュ
- オブジェクトへのアクセスおよび操作のためのアソシエイティブ・インタフェースおよびナビゲーション・インタフェース
- オブジェクト用のランタイム環境
- Oracle データベース内のオブジェクト型の情報にアクセスするための型管理関数
- Oracle 型のデータ属性を操作するための型マッピングおよび操作関数
- Oracle 内部スキーマ情報をクライアント側の言語バインド変数にマッピングする Object Type Translator (OTT) ユーティリティ

クライアント側のオブジェクト・キャッシュ

オブジェクト・キャッシュは、オブジェクトの参照とメモリー管理をサポートするクライアント側のメモリー・バッファです。オブジェクト・キャッシュは、OCI アプリケーションがサーバーからクライアント側にフェッチしたオブジェクト・インスタンスを格納し、追跡します。オブジェクト・キャッシュは、OCI 環境を初期化するときに作成されます。同じサーバーに対して実行中の複数のアプリケーションには、それぞれ専用のオブジェクト・キャッシュがあります。オブジェクト・キャッシュは、メモリーに現在あるオブジェクトの追跡、オブジェクトへの参照のメンテナンス、自動オブジェクト・スワッピングの管理、オブジェクトのメタ属性または型情報の追跡を行います。オブジェクト・キャッシュは OCI アプリケーションに次のものを提供します。

- オブジェクトのフェッチおよび操作に必要なクライアント / サーバー間のラウンドトリップ回数の削減によるアプリケーションのパフォーマンスの改善
- クライアント側のキャッシュからのオブジェクト・スワッピングのサポートによる拡張性の向上
- オブジェクト・レベルのロックのサポートによる並行性の向上

アソシエイティブ・インタフェースおよびナビゲーションル・インタフェース

OCI を使用したアプリケーションでは、次のような様々な種類のインタフェースを介して、Oracle サーバー内のオブジェクトにアクセスできます。

- SQL の SELECT 文、INSERT 文および UPDATE 文の使用
- 対応するスマート・ポインタまたは REF を横断することによりクライアント側のキャッシュ内のオブジェクトにアクセスする、C スタイルのポインタ追跡スキームの使用

OCI には、SQL の SELECT 文、INSERT 文および UPDATE 文を使用したオブジェクト操作をサポートする拡張機能を備えた、一連の関数が用意されています。Oracle オブジェクトにアクセスするために、これらの SQL 文では、リレーショナル表にアクセスする場合のような一貫した一連のステップが使用されます。OCI は、SQL 文を使用してオブジェクトにアクセスするために必要な一連の関数を提供します。関数は次の目的で使用されます。

- オブジェクト型のインスタンスと参照を SQL 文の入出力変数としてバインドおよび定義します。
- オブジェクト型のインスタンスと参照を含む SQL 文を実行します。
- オブジェクト型のインスタンスと参照をフェッチします。
- Oracle オブジェクト型の選択リスト項目を記述します。

また、OCI は、C スタイルのポインタ追跡スキームを使用してクライアント側のキャッシュ内のオブジェクトにアクセスする一連の関数を提供しています。これらのオブジェクトは、対応するスマート・ポインタまたは REF を横断することにより、クライアント側のキャッシュ内にフェッチされます。このナビゲーション・インタフェースでは、関数が次の目的で使用されます。

- 参照可能な永続オブジェクト、つまりオブジェクト ID を持つ永続オブジェクトのスマート・ポインタまたは REF を確保することにより、そのオブジェクトのコピーをクライアント側のキャッシュにインスタンス化します。
- 互いを指す REF を横断することにより、相互に接続されている一連のオブジェクトを横断します。
- オブジェクト属性の値を動的に取得し、設定します。

オブジェクト用のランタイム環境

OCI は、オブジェクト用のランタイム環境を提供します。このランタイム環境では、Oracle オブジェクトのクライアント側での使用方法を管理する一連の関数を使用できます。これらの関数は、次のタスクの実行に必要な機能を提供します。

- セッションの初期化、データベース・サーバーへのログイン、接続の登録などのオブジェクト機能にアクセスするための、Oracle サーバーへの接続
- クライアント側のオブジェクト・キャッシュの設定およびそのパラメータのチューニング
- エラーおよび警告メッセージの取得
- サーバー内のオブジェクトにアクセスするトランザクションの制御
- SQL によるオブジェクトへの結合アクセス
- パラメータまたは結果が Oracle タイプ、システムのタイプである PL/SQL プロシージャまたはファンクションの記述

型管理、マッピングおよび操作関数

OCI は、Oracle オブジェクトを操作するための 2 つの関数セットを提供します。

- 型マッピング関数を使用すると、アプリケーションで、数値型、日付型、文字列型など Oracle 内部データ型としてサーバー内に表されている Oracle スキーマの属性を、整数、月、日などの対応するホスト言語型にマップできます。
- 型操作関数を使用すると、ホスト言語のアプリケーションで、Oracle スキーマの個別の属性を操作できます。属性の値を設定または取得したりサーバーにフラッシュできます。

また、OCIDescribeAny() 関数を使用すると、データベース内に格納されたオブジェクトの情報を取得できます。

Object Type Translator

Object Type Translator (OTT) ユーティリティは、Oracle オブジェクト型のスキーマ情報をクライアント側の言語バインドに変換します。つまり、Oracle OTT によって、型情報が構造体やクラスなどのホスト言語変数の宣言に変換されます。OTT は、Oracle スキーマ・オブジェクトのメタデータ情報が含まれる `intype` ファイルを入力として使用します。OTT は、次に `outtype` ファイルおよび必要なヘッダー・ファイルと実装ファイルを生成します。このヘッダー・ファイルと実装ファイルは、オブジェクト・スキーマに対して実行する C アプリケーションに組み込む必要があります。OCI アプリケーションおよび Pro*C/C++ プリコンパイラ・アプリケーションには、OTT で生成したコードを組み込むことができます。OTT には次のような多くの利点があります。

- アプリケーション開発者の生産性が向上します。OTT を使用すると、アプリケーション開発者はスキーマ・オブジェクトに対応するホスト言語変数を手作業で記述する必要がなくなります。
- SQL を、選択したデータ定義言語としてメンテナンスします。OTT では、SQL を使用して作成した Oracle スキーマ・オブジェクトをホスト言語変数に自動的にマップできるため、SQL がデータ定義言語として使用しやすくなります。その結果、Oracle によって、ユーザー・データの一貫したモデルを企業全体でサポートできます。
- オブジェクト型のスキーマ展開を容易にします。OTT では、スキーマが変更されたときにインクルードされたヘッダー・ファイルを再生成できるため、Oracle アプリケーションでスキーマ展開をサポートできます。

通常、OTT は、コマンドラインから `intype` ファイル、`outtype` ファイルおよび特定のデータベース接続を指定することにより起動されます。Oracle の OTT では、OCI プログラムと Pro*C/C++ プリコンパイラ・プログラムのどちらでも使用できる C 構造体のみが生成されます。

OCI での Oracle Advanced Queuing のサポート

OCI では、Oracle Advanced Queuing (AQ) 機能へのインタフェースを提供します。Oracle AQ は、Oracle サーバーと統合されたメッセージ・キューイングです。Oracle AQ のメッセージ・キューイングでは、キューイング・システムをデータベースと統合し、メッセージ対応データベースを作成する機能を提供しています。Oracle AQ では、統合されたソリューションを提供することにより、開発者がメッセージ交換インフラストラクチャを構築する必要をなくし、特定のビジネス・ロジックに専念できるようにします。

関連項目： OCI AQ 機能の詳細は、9-48 ページの「[OCI およびアドバンスト・キューイング](#)」を参照してください。

XA ライブラリ・サポート

関連項目： Oracle XA ライブラリ・サポートの詳細は、『Oracle9i アプリケーション開発者ガイド - 基礎編』を参照してください。

既存アプリケーションのアップグレードの簡素化

OCI では、多くの機能が大幅に改善されました。OCI リリース 7.x のクライアントとこのリリースのサーバー、およびこのリリースのクライアントと Oracle7 データベース・サーバーにはそれぞれ相互運用性があるため、OCI リリース 7.x を使用するために作成したアプリケーションは、このリリースの OCI に容易にアップグレードできます。

具体的には次のとおりです。

- OCI リリース 7.3 を使用したアプリケーションは、このリリースのサーバーに対して変化なく動作します。
- このリリースの OCI を使用したアプリケーションは、OCI またはサーバーの新機能を使用していない場合、Oracle7 サーバーに対して動作します。
- OCI リリース 7.x のコールおよびこのリリースの OCI のコールは、同じ文の中で混在させないかぎり、同一アプリケーションおよび同一トランザクション内で使用できます。

このため、既存の OCI リリース 7.x のアプリケーションをアップグレードする場合は、次の 3 つの方法があります。

- Oracle7 OCI クライアントを保持します。既存の Oracle7 OCI アプリケーションは、修正することなく保持できます。このアプリケーションは、カレント・サーバーに対しても動作します。
- 現行の OCI クライアントにアップグレードしますが、アプリケーションは修正しません。Oracle7 OCI クライアントからカレント・リリースの OCI クライアントにアップグレードする場合は、新しいバージョンの OCI ライブラリの再リンクのみを実行します。アプリケーションを再コンパイルする必要はありません。再リンクした Oracle7 OCI アプリケーションは、カレント・サーバーに対して変化なく動作します。
- Oracle9i OCI クライアントにアップグレードし、アプリケーションを修正します。ただし、新しい OCI で提供されるパフォーマンスと拡張性の利点を活用するには、新しい OCI プログラミング・パラダイムを使用できるように既存のアプリケーションを修正して、そのアプリケーションを新しい OCI ライブラリに再リンクし、カレント・リリースのサーバーに対して実行する必要があります。

また、カレント・リリースのサーバーのオブジェクト機能を使用する必要がある場合は、クライアントをアップグレードして、このリリースの OCI を使用する必要があります。

互換性およびアップグレード

このリリースの OCI では、OCI リリース 7.x またはリリース 8.x 以上の OCI で作成されたアプリケーションをサポートしています。この項では、異なるバージョンの OCI およびサーバー間の互換性の問題、OCI ライブラリ・ルーチンの変更点および OCI リリース 7.x からこのリリースの OCI へのアプリケーションのアップグレードについて説明します。

関連項目： 互換性およびアップグレードの最新情報は、『Oracle9i データベース移行ガイド』を参照してください。

使用されなくなった OCI ルーチン

Oracle Call Interface リリース 8.0 では、リリース 7.3 で利用できなかった新しい関数セットを導入しました。リリース 8.1 では、新しい機能が追加されています。Oracle9i OCI では、これらの新機能を引き続きサポートし、さらに新しいコールも追加されています。以前のリリース 7.x コールも使用できますが、パフォーマンスを向上し、機能性を高めるために、既存のアプリケーションで新しいコールを使用することをお薦めします。

表 1-1 「使用されなくなった OCI ルーチン」は、リリース 7.x の OCI コールおよびリリース 8.x 以上での等価コールのリストです。OCI コールの詳細は、このマニュアルの第 III 部の関数の説明を参照してください。リリース 7.x コールの詳細は、『Oracle Call Interface プログラマーズ・ガイド』を参照してください。これらのリリース 7.x コールは廃止されています。つまり、OCI ではこれらが新しいコールで置き換えられています。廃止されたコールはこの時点ではサポートされていますが、OCI の将来のバージョンではサポートされない可能性があります。

注意： 多くの場合、新しい OCI ルーチンはリリース 7.x のルーチンに直接マッピングを行わないため、ファンクション・コールとパラメータ・リストを単純に別のものに置き換えることができない場合があります。この場合、新しいコールを作成する前後に、追加のプログラム・ロジックが必要になります。詳細は、このマニュアルの残りの章を参照してください。

表 1-1 使用されなくなった OCI ルーチン

7.x OCI ルーチン	8.x 以上での等価または類似の OCI ルーチン
obindps(), obndra(), obndrn(), obndrv()	OCIBindByName(), OCIBindByPos() (注意： 一部のデータ型には追加のバインド・コールが必要です。)
obreak()	OCIBreak()
ocan()	なし
oclose()	注意： リリース 8.x 以上では、カーソルは使用されていません。
ocof(), ocon()	OCI_COMMIT_ON_SUCCESS モードでの OCIStmtExecute()

表 1-1 使用されなくなった OCI ルーチン（続き）

7.x OCI ルーチン	8.x 以上での等価または類似の OCI ルーチン
<code>ocom()</code>	<code>OCITransCommit()</code>
<code>odefin()</code> 、 <code>odefinps()</code>	<code>OCIDefineByPos()</code> （ 注意 ：一部のデータ型には追加の定義コールが必要です。）
<code>odescr()</code>	注意 ：スキーマ・オブジェクトは <code>OCIDescribeAny()</code> で記述されます。通常、記述は、リリース 7.x で使用されているように、SQL 文の実行後に <code>OCIAttrGet()</code> を文ハンドルでコールすることにより行われます。
<code>odessp()</code>	<code>OCIDescribeAny()</code>
<code>oerhms()</code>	<code>OCIErrorGet()</code>
<code>oexec()</code> 、 <code>oexn()</code>	<code>OCIStmtExecute()</code>
<code>oexfet()</code>	<code>OCIStmtExecute()</code> 、 <code>OCIStmtFetch()</code> （ 注意 ：結果セット行は暗黙的にプリフェッチできます。）
<code>ofen()</code> 、 <code>ofetch()</code>	<code>OCIStmtFetch()</code>
<code>oflng()</code>	なし
<code>ogetpi()</code>	<code>OCIStmtGetPieceInfo()</code>
<code>olog()</code>	<code>OCILogon()</code>
<code>ologof()</code>	<code>OCILogoff()</code>
<code>onbclr()</code> 、 <code>onbset()</code> 、 <code>onbtst()</code>	注意 ：非ブロック化モードは、 <code>OCIAttrSet()</code> または <code>OCIAttrGet()</code> をサーバー・コンテキスト・ハンドルまたはサービス・コンテキスト・ハンドルでコールすることにより、設定またはチェックすることができます。
<code>oopen()</code>	注意 ：リリース 8.x 以上では、カーソルは使用されていません。
<code>oopt()</code>	なし
<code>oparse()</code>	<code>OCIStmtPrepare()</code> 。ただし、すべてローカル。
<code>opinit()</code>	<code>OCIEnvCreate()</code>
<code>orol()</code>	<code>OCITransRollback()</code>
<code>osetpi()</code>	<code>OCIStmtSetPieceInfo()</code>
<code>sqlld2()</code>	<code>xaoSvcCtx()</code> または <code>xaoEnv()</code>
<code>sqllda()</code>	<code>SQLSvcCtxGet()</code> または <code>SQLEnvGet()</code>
<code>odsc()</code>	注意 ：前述の <code>odescr()</code> を参照してください。

表 1-1 使用されなくなった OCI ルーチン（続き）

7.x OCI ルーチン	8.x 以上での等価または類似の OCI ルーチン
oerrmsg()	OCIErrorGet()
olon()	OCILogon()
orlon()	OCILogon()
oname()	注意: 前述の odescr() を参照してください。
osql3()	注意: 前述の oparse() を参照してください。

関連項目: このリストにない新しい関数によって提供される追加機能の詳細は、このマニュアルの他の章に記載されていますので参照してください。

サポートされない OCI ルーチン

OCI の以前のバージョンで使用可能だった OCI ルーチンの一部は、Oracle8i または Oracle9i ではサポートされません。それらのルーチンのリストを、表 1-2 「サポートされない OCI ルーチン」 に示します。

表 1-2 サポートされない OCI ルーチン

OCI ルーチン	8.x 以上での等価または類似の OCI ルーチン
obind()	OCIBindByName(), OCIBindByPos() (注意: 一部のデータ型には追加のバインド・コールが必要です。)
obindn()	OCIBindByName(), OCIBindByPos() (注意: 一部のデータ型には追加のバインド・コールが必要です。)
odfinn()	OCIDefineByPos() (注意: 一部のデータ型には追加の定義コールが必要です。)
odsrbn()	注意: 表 1-1 の odescr() を参照してください。
ologon()	OCILogon()
osql()	注意: 表 1-1 の oparse() を参照してください。

互換性

この項では、異なるバージョンの OCI および Oracle サーバー間の互換性について説明します。

リリース 8.x 以上の新しい OCI コールを含まない既存のリリース 7.x アプリケーションでは、次の 2 つの選択肢があります。

- アプリケーションの再リンクを行わない
- リリース 8.x 以上の新しい OCI ライブラリへの再リンクを行う

いずれの場合も、関数 `ocom()` が `ocon()` で代用される点を除いて、アプリケーションは Oracle7 と Oracle8i 以上の両方に対して動作します。`ocon()` によって AUTOCOMMIT (DML 文ごとの自動コミット) が使用可能になるため、後続のフェッチ文でエラーが発生します。

このアプリケーションでは、Oracle8i 以上のオブジェクト機能が使用できません。また、これらの OCI リリースで提供されるパフォーマンスや拡張性も利用できません。

OCI で作成された新しいアプリケーションは、次の例外を除いて、Oracle7 および Oracle8i 以上に対してシームレスに動作します。

- Oracle7 Server に対して、Oracle のオブジェクト機能はいずれもサポートされません。また、次のデータ型はサポートされません。
 - `SQLT_NTY` – 名前付きデータ型
 - `SQLT_REF` – ホスト言語表現の名前付きデータ型に対する参照
 - `SQLT_CLOB` – キャラクタ LOB データ型
 - `SQLT_BLOB` – バイナリ LOB データ型
 - `SQLT_BFILE` – バイナリ FILE LOB データ型
 - `SQLT_RSET` – 結果セット・データ型
 - `SQLT_DATE` – ANSI DATE
 - `SQLT_TIMESTAMP` – TIMESTAMP
 - `SQLT_TIMESTAMP_TZ` – TIMESTAMP WITH TIME ZONE
 - `SQLT_TIMESTAMP_LTZ` – TIMESTAMP WITH LOCAL TIME ZONE
 - `SQLT_INTERVAL_DS` – INTERVAL DAY TO SECOND
 - `SQLT_INTERVAL_YM` – INTERVAL YEAR TO MONTH
- Oracle7 Server に対して、次のコールまたは機能はサポートされないか、制限付きでサポートされます。

表 1-3 Oracle7 Server に対して Oracle8i 以上の OCI を実行した場合の制限

関数	制限
OCIBindObject()	未サポート
OCIPasswordChange()	未サポート
OCIDefineObject()	未サポート
OCIDescribeAny()	選択リストまたはストアド・プロシージャの記述のみをサポート
OCIErrorGet()	Oracle エラー・コードのサブセットのみを戻すことが可能
OCIStmtFetch()	プリフェッチ・オプションは未サポート
OCILob*()	LOB/FILE コールは未サポート
OCIAttrSet()	NCHAR 属性の設定は未サポート
OCIAttrGet()	NCHAR 属性の取得は未サポート

アップグレード

プログラマがリリース 8.x 以上の機能を既存の OCI アプリケーションに取り込む場合は、次の 2 つの選択肢があります。

- アプリケーションを完全にリライトして、新しい OCI コールのみを使用（推奨）
- リリース 8.x 以上の新しい OCI コールをアプリケーションに取り込む一方で、一部の操作にはリリース 7.x コールを使用

このマニュアルでは、既存のアプリケーションをリライトして新しい OCI コールのみを使用するために必要な情報を提供しています。

リリース 8.x 以上の OCI コールのリリース 7.x アプリケーションへの追加

次のガイドラインは、プログラマが新しい OCI コールを使用して新しい Oracle のデータ型および機能を取り込む一方で、一部の操作にはリリース 7.x コールを使用する場合に適用されます。

- 既存のログインを変更して、olog()（または他のログイン・コール）のかわりに OCILogon() を使用します。サービス・コンテキスト・ハンドルは、新しい OCI コールで使用できます。または、リリース 7.x の OCI コールで使用する Lda_Def に変換できます。

関連項目： 複数のセッションをメンテナンスしているアプリケーションに必要なログイン・コールの詳細は、15-27 ページの「[OCIserverAttach\(\)](#)」および 15-30 ページの「[OCISessionBegin\(\)](#)」の説明を参照してください。

- サーバー・コンテキスト・ハンドルを初期化した後は、リリース 8.x 以上の OCI コールで使用できます。
- Oracle7 OCI コールを使用するには、`OCISvcCtxToLda()` を使用してサーバー・コンテキスト・ハンドルを **Lda_Def** に変換し、結果の **Lda_Def** をリリース 7.x コールに渡します。

注意： 同じサーバー・ハンドルを共有する複数のサービス・コンテキストがある場合、Oracle7 モードでは常に 1 つのサービス・コンテキストのみ可能です。

- リリース 8.x 以上の OCI コールの使用を再開する場合は、アプリケーションで `OCILdaToSvcCtx()` を使用して **Lda_Def** をサーバー・コンテキスト・ハンドルに変換しなおす必要があります。
- アプリケーションでは、必要に応じて、**Lda_Def** とサーバー・コンテキストを切り替えることができます。

このアプローチを行うと、アプリケーションで、単一の接続で 2 つの異なる API を使用して異なるタスクを達成できます。

リリース 7.x の OCI コールとリリース 8.x 以上の OCI コールを混在および一致させることは、トランザクション内では可能ですが、文内ではできません。これにより、ある SQL 文または PL/SQL 文をリリース 7.x の OCI コールで実行し、そのトランザクション内で、次の SQL 文または PL/SQL 文を Oracle8.x 以上の OCI コールで実行できます。

注意： カーソルをオープンして、リリース 7.x の OCI コールで解析し、次にリリース 8.x 以上の OCI コールで文を実行することはできません。

OCI プログラミングの基本

この章では、Oracle Call Interface (OCI) を使用してプログラムを作成する際に必要となる基本概念の概要を説明します。この章は、次の項目で構成されています。

- OCI プログラミングの概要
- OCI プログラム構造
- OCI データ構造
- ハンドル
- 記述子
- OCI プログラミング・ステップ
- OCI 環境の初期化
- SQL 文の処理
- コミットまたはロールバック
- アプリケーションの終了
- エラー処理
- その他のコーディング・ガイドライン
- OCI プログラムでの PL/SQL 使用
- OCI グローバリゼーション・サポート
- OCI データベースのグローバリゼーション・サポート関数

OCI プログラミングの概要

この章では、OCI アプリケーションの開発における基本的な概念とプロシージャについて説明します。この章を読み終えると、基本的な OCI アプリケーションの作成に必要なツールについてほとんど習得したことになります。

この章は、次の項に大きく分かれています。

- [OCI プログラム構造](#) – OCI アプリケーションの基本構造と作成の大まかなステップを説明します。
- [OCI データ構造](#) – ハンドルおよび記述子について説明します。
- [OCI プログラミング・ステップ](#) – OCI アプリケーションのコーディングに必要な各ステップを詳しく説明します。
- [エラー処理](#) – OCI アプリケーションでのエラー処理を説明します。
- [その他のコーディング・ガイドライン](#) – OCI アプリケーションのコーディング時に役立つ注意事項を説明します。
- [非ブロック化モード](#) – 非ブロック化モードを使用した Oracle データベース・サーバーへの接続を説明します。
- [OCI プログラムでの PL/SQL 使用](#) – OCI アプリケーションで PL/SQL を扱う際の重要事項を説明します。

新規ユーザーは、この章の情報に特に注意してください。この章は、以降の章を理解する基本となります。以降の章は、この章の内容を補足しています。

関連項目：

- 多言語環境に適用される OCI 関数の詳細は、『Oracle9i Database グローバリゼーション・サポート・ガイド』を参照してください。
- カートリッジ・サービスに適用される OCI 関数の詳細は、『Oracle9i Data Cartridge Developer's Guide』を参照してください。

OCI プログラム構造

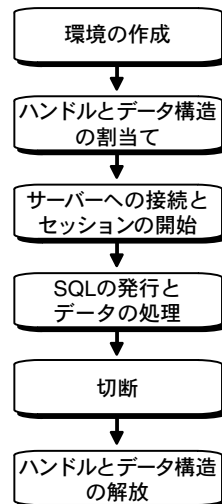
OCI アプリケーションの一般的な目標は、マルチ・ユーザーのために操作を行うことです。N 層構造では、マルチ・ユーザーがクライアント・アプリケーションに HTTP 要求を送信します。クライアント・アプリケーションでは、データ交換やデータ処理など、いくつかのデータ操作を実行する必要があります。

OCI では、次のプログラム基本構造を採用しています。

1. OCI プログラミング環境およびスレッドの初期化
2. 必要なハンドルの割当て、およびサーバー接続とユーザー・セッションの確立
3. サーバー上での SQL 文の実行、データベース・サーバーとの間でのデータ交換、必要なアプリケーション・データ処理の実行
4. プリコンパイルされた SQL 文の再実行、または新規に実行する文の準備
5. ユーザー・セッションおよびサーバー接続の終了
6. フリー・ハンドル

図 2-1 「基本的な OCI プログラムの流れ」は、OCI アプリケーションのステップの流れを示しています。各ステップの詳細は 2-20 ページの「OCI プログラミング・ステップ」を参照してください。

図 2-1 基本的な OCI プログラムの流れ

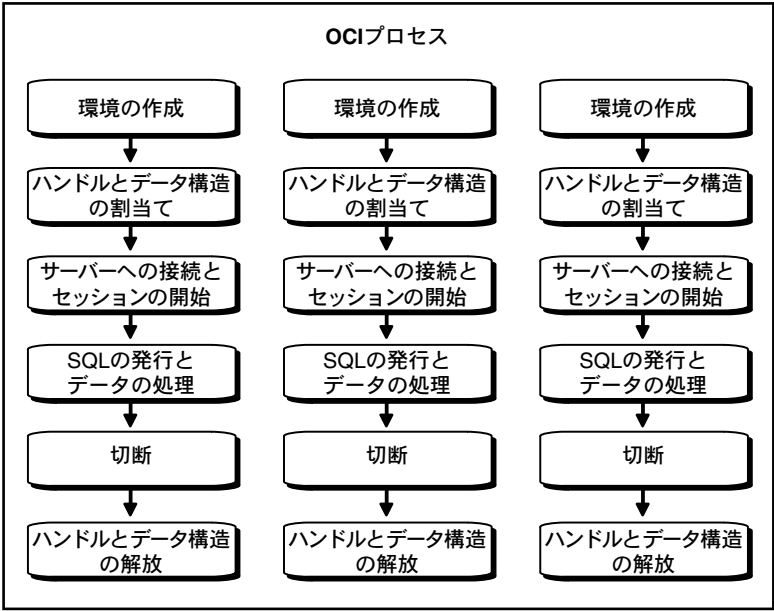


図およびステップ・リストは、OCI プログラミングのステップを簡潔に表したものです。さらに、プログラムの機能性に応じて様々なバリエーションが考えられます。複数セッションおよび複数トランザクションの管理やオブジェクトの使用など、より高度な機能が組み込まれる OCI アプリケーションには、追加のステップが必要です。

すべての OCI ファンクション・コールは、ある環境のコンテキスト内で実行されます。図 2-2 「OCI プロセスでの複数環境」に示すように、1 つの OCI プロセス内に複数の環境が存在することが可能です。環境でプロセスレベルの初期化が必要になった場合には、自動的に実行されます。

注意： 以前のリリースでは、個別のプロセスレベルの初期化を明示的に行う必要がありました。この初期化は自動化されたため、明示的なプロセスレベルの初期化は必要ありません。

図 2-2 OCI プロセスでの複数環境



注意： OCI アプリケーションには、アクティブな接続および文を 2 つ以上含めることができます。

関連項目： オブジェクトへのアクセスおよび操作の詳細は、[第 10 章「OCI オブジェクト・リレーショナル・プログラミング」](#) およびそれ以降の章を参照してください。

OCI データ構造

ハンドルと記述子は、OCI アプリケーション内で定義される不透明なデータ構造であり、明示的な割当てコールによって直接割り当てるか、あるいは OCI 関数を使用して暗黙的に割り当てることができます。

7.x アップグレードの注意： リリース 7.x OCI アプリケーションを作成した経験のあるプログラマは、大部分の OCI コールに使用されているこれらの新しいデータ構造に慣れる必要があります。

ハンドルおよび記述子を使用して、データ、接続またはアプリケーション動作に関する情報を格納します。ハンドルについては、次の項で詳しく定義します。記述子の詳細は、2-15 ページの「[記述子](#)」を参照してください。

ハンドル

ほとんどの場合、OCI コールのパラメータ・リストには、ハンドルが 1 つ以上含まれています。ハンドルとは、OCI ライブラリによって割り当てられる記憶領域を指す不透明なポインタです。ハンドルを使用すると、コンテキストや接続に関する情報（環境コンテキスト・ハンドルやサービス・コンテキスト・ハンドルなど）を格納できます。また OCI 関数やデータに関する情報（エラー・ハンドルや記述ハンドルなど）を格納する場合があります。つまり、ハンドルを使用するとアプリケーションではなくライブラリでこのデータを維持できるため、プログラミングが簡単になります。

ほとんどの OCI アプリケーションでは、ハンドルに格納されている情報へのアクセスが必要となります。この情報にアクセスするには、属性の入手および設定のための OCI コールである OCIAttrGet() および OCIAttrSet() を使用します。

関連項目： ハンドル属性の使用方法の詳細は、2-13 ページの「[ハンドル属性](#)」を参照してください。

次の表は、OCI 用に定義されているハンドルを示しています。それぞれのハンドル・タイプについて、OCI コールでハンドル・タイプを識別するために使用される、C データ型およびハンドル・タイプ定数をリストしています。

表 2-1 OCI ハンドル・タイプ

説明	C 型	ハンドル・タイプ
OCI 環境ハンドル	OCIEnv	OCI_HTYPE_ENV
OCI エラー・ハンドル	OCLError	OCI_HTYPE_ERROR
OCI サービス・コンテキスト・ハンドル	OCISvcCtx	OCI_HTYPE_SVCCTX
OCI 文ハンドル	OCIStmt	OCI_HTYPE_STMT
OCI バインド・ハンドル	OCIBind	OCI_HTYPE_BIND
OCI 定義ハンドル	OCIDefine	OCI_HTYPE_DEFINE
OCI 記述ハンドル	OCIDescribe	OCI_HTYPE_DESCRIBE
OCI サーバー・ハンドル	OCIServer	OCI_HTYPE_SERVER
OCI ユーザー・セッション・ハンドル	OCISession	OCI_HTYPE_SESSION
OCI 認証情報ハンドル	OCIAuthInfo	OCI_HTYPE_AUTHINFO
OCI 接続プール・ハンドル	OCICPool	OCI_HTYPE_CPOOL
OCI セッション・プール・ハンドル	OCISPool	OCI_HTYPE_SPOOL
OCI トランザクション・ハンドル	OCITrans	OCI_HTYPE_TRANS
OCI 複合オブジェクト検索 (COR) ハンドル	OCIComplexObject	OCI_HTYPE_COMPLEXOBJECT
OCI スレッド・ハンドル	OCIThreadHandle	該当なし
OCI サブスクリプション・ハンドル	OCISubscription	OCI_HTYPE_SUBSCRIPTION
OCI ダイレクト・パス・コンテキスト・ ハンドル	OCIDirPathCtx	OCI_HTYPE_DIRPATH_CTX
OCI ダイレクト・パス関数コンテキスト・ ハンドル	OCIDirPathFuncCtx	OCI_HTYPE_DIRPATH_FN_CTX
OCI ダイレクト・パス列配列ハンドル	OCIDirPathColArray	OCI_HTYPE_DIRPATH_COLUMN_ARRAY
OCI ダイレクト・パス・ストリーム・ ハンドル	OCIDirPathStream	OCI_HTYPE_DIRPATH_STREAM
OCI プロセス・ハンドル		OCI_HTYPE_PROC

ハンドルの割当てと解放

アプリケーションでは、すべてのハンドル（バインド・ハンドル、定義ハンドルおよびスレッド・ハンドルは除く）が、特定の環境ハンドルに対して割り当てられます。環境ハンドルをパラメータの1つとして、ハンドル割当てコールに渡します。こうして、割り当てられたハンドルは、その特定の環境に固有のものとなります。

バインド・ハンドルおよび定義ハンドルは、文ハンドルを親として割り当てられ、そのハンドルによって表される文についての情報を含みます。

注意： バインド・ハンドルおよび定義ハンドルは、OCI ライブラリによって暗黙的に割り当てられるため、ユーザーによる割当ては不要です。

図 2-3 「ハンドルの階層」は、様々なハンドル・タイプ間の関係を示しています。

ユーザー割当てハンドルは、すべて OCI ハンドル割当てコール `OCIHandleAlloc()` によって割り当てられます。

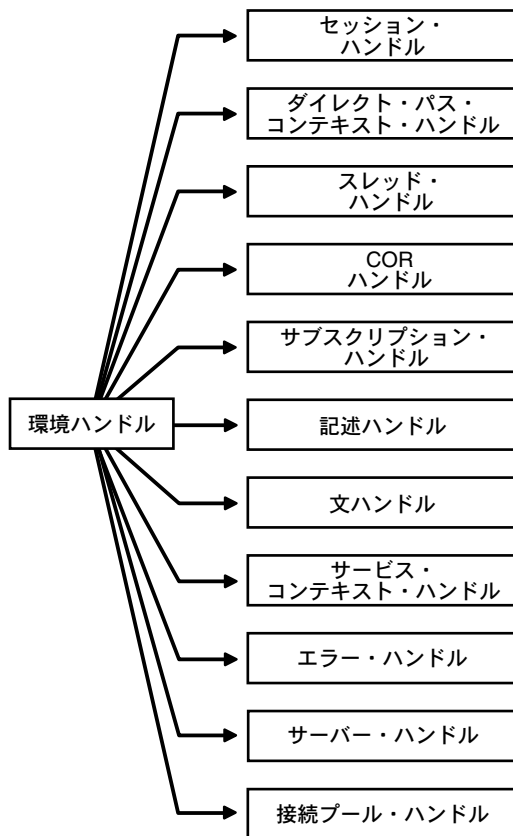
注意： 環境ハンドルは `OCIEnvCreate()` のコールによって割り当てられ、初期化されます。このコールは、すべての OCI アプリケーションで必要です。

スレッド・ハンドルは、`OCIThreadHndInit()` コールによって割り当てられます。

アプリケーションは、ハンドルが不要になった時点で全ハンドルを解放する必要があります。ハンドルは `OCIHandleFree()` 関数によって解放されます。

注意： 親ハンドルが解放されると、それに対応付けられた子ハンドルもすべて解放され、それ以降使用できなくなります。たとえば、文ハンドルが解放されると、それに関連付けられたバインド・ハンドルおよび定義ハンドルも解放されます。

図 2-3 ハンドルの階層



ハンドルを使用することにより、必要なグローバル変数が少なくなります。また、エラー・レポートが簡単になります。エラー・ハンドルは、エラーと診断情報を戻すために使用します。

関連項目： OCI ハンドルの割当てと使用方法を示したコード例は、[付録 B「OCI デモ・プログラム」](#) のプログラム例を参照してください。

以降の項では、各種のハンドルについて詳しく説明します。

環境ハンドル

環境ハンドルは、すべての OCI 関数をコールするコンテキストを定義します。各環境ハンドルにはメモリ・キャッシュがあり、これによってメモリ・アクセスが高速になります。環境ハンドルに基づくメモリの割当ては、このキャッシュで行われます。キャッシュへのアクセスは、複数のスレッドが同じ環境ハンドルに基づいてメモリの割当てを試行する場合にシリアル化されます。複数のスレッドで単一の環境ハンドルを共有する場合、スレッドはキャッシュへのアクセスをブロックすることがあります。

環境ハンドルは、*parent* パラメータとして `OCIHandleAlloc()` コールに渡され、その他のすべてのハンドル・タイプを割り当てます。バインド・ハンドルと定義ハンドルは、暗黙的に割り当てられます。

エラー・ハンドル

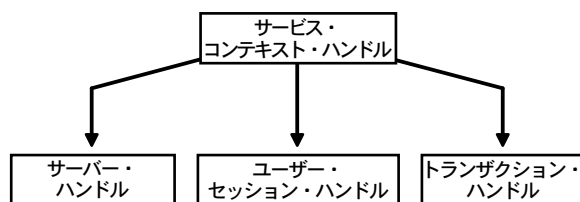
エラー・ハンドルは、ほとんどの OCI コールにパラメータとして渡されます。エラー・ハンドルは、OCI 操作中に発生したエラーについての情報をメンテナンスします。コールでエラーが発生した場合、エラー・ハンドルを `OCIErrorGet()` に渡して、発生したエラーに関する追加情報を取得することができます。

ほとんどの OCI コールではパラメータの 1 つとしてエラー・ハンドルが必要であるため、エラー・ハンドルの割当ては、OCI アプリケーションの最初のステップです。

サービス・コンテキスト・ハンドルとそれに対応付けられたハンドル

サービス・コンテキスト・ハンドルは、サーバーに対する OCI コールの操作コンテキストを決定する属性を定義します。サービス・コンテキストには、サーバー接続、ユーザー・セッションおよびトランザクションを示す 3 つのハンドルが属性として含まれています。これらの属性を、[図 2-4 「サービス・コンテキストのコンポーネント」](#) に示します。

図 2-4 サービス・コンテキストのコンポーネント



- サーバー・ハンドルは、データベースとの接続を識別します。これは、接続指向のトランSPORT・メカニズムによって物理接続に変換されます。
- ユーザー・セッション・ハンドルは、ユーザーのロールと権限（ユーザーのセキュリティ・ドメインとも呼ばれます）およびコールを実行するための操作コンテキストを定義します。
- トランザクション・ハンドルは、SQL 操作を実行するためのトランザクションを定義します。トランザクション・コンテキストには、フェッチ状態やパッケージのインスタンス化などの、ユーザー・セッション状態の情報が含まれています。

この方法によるサービス・コンテキストの分析は、拡張性を提供し、プログラマは、洗練された3層アプリケーションとトランザクション処理（TP）モニターを作成して、複数のアプリケーション・サーバーと異なるトランザクション・コンテキスト上の複数のユーザーの要求を実行できます。

サービス・コンテキスト・ハンドルを使用する際は、その前に `OCIHandleAlloc()` または `OCILogon()` によって、割当てと初期化を行う必要があります。サービス・コンテキスト・ハンドルは、`OCIHandleAlloc()` によって明示的に割り当てられます。また、サーバー・ハンドル、セッション・ハンドルおよびトランザクション・ハンドルでは、`OCIAttrSet()` を使用して初期化できます。サービス・コンテキスト・ハンドルが `OCILogon()` によって暗黙的に割り当てられている場合には、すでに初期化されています。

データベース接続ごとに常時シングル・ユーザー・セッションのみをメンテナンスするアプリケーションでは、`OCILogon()` をコールして、初期化されたサービス・コンテキスト・ハンドルを取得できます。

より複雑なセッション管理を必要とするアプリケーションでは、サービス・コンテキストを明示的に割り当てる必要があります。また、サーバー・ハンドルおよびユーザー・セッション・ハンドルは、サービス・コンテキストに明示的に設定する必要があります。

`OCIServerAttach()` コールと `OCISessionBegin()` コールは、サーバー・ハンドルとユーザー・セッション・ハンドルを初期化します。

グローバル・トランザクションの場合、あるいはセッションに対してアクティブなトランザクションが複数存在する場合、アプリケーションではトランザクションを明示的に定義するのみです。また、アプリケーションでは、データベースの内容を変更した際に OCI によって自動的に作成された暗黙的トランザクションを正しく処理できます。

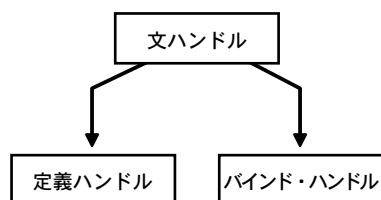
関連項目：

- トランザクションの詳細は、8-2 ページの「[OCI でのトランザクションのサポート](#)」を参照してください。
- サーバー接続およびユーザー・セッションの確立の詳細は、2-21 ページの「[OCI 環境の初期化](#)」および 8-11 ページの「[パスワードおよびセッションの管理](#)」を参照してください。

文ハンドル、バインド・ハンドルおよび定義ハンドル

文ハンドルは、SQL 文または PL/SQL 文と、それぞれに関連付けられた属性を識別するコンテキストです。

図 2-5 文ハンドル



入力バインド変数および出力バインド変数に関する情報は、バインド・ハンドルに格納されます。OCI ライブラリでは、`OCIBindByName()` または `OCIBindByPos()` 関数でバインドされた各プレースホルダのバインド・ハンドルが割り当てられます。ユーザーがバインド・ハンドルを割り当てる必要はありません。バインド・ハンドルは、バインド・コールによって暗黙的に割り当てられます。

問合せ (`select` 文) でフェッチされて戻されたデータは、定義ハンドルの指定に従って変換および格納されます。また OCI ライブラリによって、`OCIDefineByPos()` で定義された各出力変数の定義ハンドルが割り当てられます。ユーザーが定義ハンドルを割り当てる必要はありません。定義ハンドルは、定義コールによって暗黙的に割り当てられます。

バインド・ハンドルおよび定義ハンドルが解放されるのは、文ハンドルが解放されたとき、または文ハンドルに新しい文が準備されたときです。

文コンテキスト・データ (文ハンドルに関連付けられたデータ) は共有できます。

関連項目： OCI 共有モードの詳細は、2-22 ページの「[共有データ・モード](#)」を参照してください。

記述ハンドル

記述ハンドルは、OCI 記述コールである `OCIDescribeAny()` によって使用されます。このコールは、ファンクションやプロシージャなどデータベース内のスキーマ・オブジェクトに関する情報を取得します。このコールでは、記述されているオブジェクトに関する情報とともに、パラメータの 1 つとして記述ハンドルが使用されます。コールが完了すると、記述ハンドルには、そのオブジェクトに関する情報が含まれています。OCI アプリケーションでは、パラメータ記述子の属性を通して記述情報を取得します。

関連項目： `OCIDescribeAny()` 関数の使用方法の詳細は、第 6 章「[スキーマ・メタデータの記述](#)」を参照してください。

複合オブジェクト検索ハンドル

複合オブジェクト検索（COR）ハンドルは、Oracle データベース・サーバー内のオブジェクトを操作する一部の OCI アプリケーションによって使用されます。このハンドルには、他のオブジェクトによって参照されるオブジェクトの検索について指示する COR 記述子が含まれています。

関連項目： 複合オブジェクト検索および複合オブジェクト検索ハンドル
の詳細は、10-20 ページの「[複合オブジェクト検索](#)」を参照してください。

スレッド・ハンドル

マルチスレッド・アプリケーションで使用されているスレッド・ハンドルの詳細は、9-5 ページの「[OCIThread パッケージ](#)」を参照してください。

サブスクリプション・ハンドル

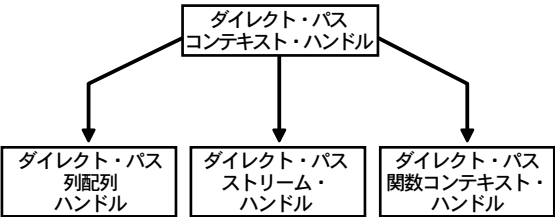
サブスクリプション・ハンドルは、サブスクリプションへの登録を行う OCI クライアント・アプリケーションによって使用され、データベース・イベントまたは AQ 名前空間内のイベントの通知を受け取るために使用されます。サブスクリプション・ハンドルは、クライアントからの登録に関するすべての情報をカプセル化します。

関連項目： パブリッシュ・サブスクライブおよびサブスクリプション・
ハンドルの割当ての詳細は、9-52 ページの「[パブリッシュ・サブスクライ
ブの通知](#)」を参照してください。

ダイレクト・パス・ハンドル

ダイレクト・パス・ハンドルは、Oracle データベース・サーバー内でダイレクト・パス・ロード・エンジンを利用する OCI アプリケーションに必要です。ダイレクト・パス・ロード・インタフェースを使用して、アプリケーションから Oracle サーバーのダイレクト・ブロック・フォーマットにアクセスできます。

図 2-6 ダイレクト・パス・ハンドル



関連項目：

- [ダイレクト・パス・ロード](#)および[ダイレクト・パス・ハンドル](#)の割当ての詳細は、12-2 ページの「[ダイレクト・パス・ロードの概要](#)」を参照してください。
- ハンドル属性の詳細は、A-58 ページの「[ダイレクト・パス・ロード・ハンドル属性](#)」を参照してください。

プロセス・ハンドル

プロセス・ハンドルは、共有データ構造モードを使用してグローバル・パラメータを設定する OCI アプリケーション専用のハンドルです。

関連項目： 2-22 ページ「[共有データ・モード](#)」

接続プール・ハンドル

接続プール・ハンドルは、特定の OCI 関数をコールして物理接続を仮想接続にプールするアプリケーションで使用されます。

関連項目： 9-13 ページ「[接続プーリング](#)」

ハンドル属性

すべての OCI ハンドルには、それに対応する属性があります。属性は、ハンドルに格納されているデータを表します。ハンドル属性は、属性取得コール `OCIAttrGet()` を使用して読み込むことができます。また、属性設定コール `OCIAttrSet()` を使用して変更することもできます。

たとえば、次の文では `OCI_ATTR_USERNAME` 属性に書き込むことで、セッション・ハンドルにユーザー名を設定しています。

```
text username[] = "scott";
err = OCIAttrSet ((dvoid*) mysessp, OCI_HTYPE_SESSION, (dvoid*) username,
                 (ub4) strlen(username), OCI_ATTR_USERNAME,
                 (OCIError *) myerrhp);
```

コールの前に特定のハンドル属性を設定することが必要な OCI 関数もあります。たとえば、ユーザーのログイン・セッションを確立するために `OCISessionBegin()` をコールする場合は、コールの実行前にユーザー名とパスワードをユーザー・セッション・ハンドルに設定する必要があります。

完了後、ハンドル属性に有用なデータを戻す OCI 関数もあります。たとえば、`OCIStmtExecute()` をコールして SQL 問合せを実行した場合は、選択リスト項目に関する記述情報が文ハンドルに戻されます。

```
ub4 parmcnt;
/* get the number of columns in the select list */
err = OCIAttrGet ((dvoid *)stmhp, (ub4)OCI_HTYPE_STMT, (dvoid *)
    &parmcnt, (ub4 *) 0, (ub4)OCI_ATTR_PARAM_COUNT, errhp);
```

関連項目：

- 設定中のユーザー名とパスワードのハンドル属性の例は、15-48 ページの「[OCIAttrGet\(\)](#)」の説明を参照してください。
- すべてのハンドル属性のリストは、付録 A「[ハンドルおよび記述子の属性](#)」を参照してください。

ユーザー・メモリーの割当て

環境ハンドルを初期化する `OCIEnvCreate()` コールと汎用ハンドル割当てコール (`OCIHandleAlloc()`) および記述子割当てコール (`OCIDescriptorAlloc()`) のパラメータ・リストには、`xtramem_sz` パラメータが含まれます。このパラメータは、ユーザーに対して、そのハンドルとともに割り当てられるメモリーのチャック・サイズを指定するために使用されます。このメモリーは OCI では使用されず、アプリケーションのみによって使用されます。

通常、アプリケーションではこのパラメータを使用して、ハンドルと同じ存続期間を持つアプリケーション定義の構造体を割り当てます。この構造体は、アプリケーション・ブックキーピングまたはコンテキスト情報の格納に使用されます。

`xtramem_sz` パラメータを使用すると、アプリケーションは、各ハンドルの割当てや割当て解除にあわせて、メモリーを明示的に割り当てたり解除する必要はありません。メモリーはハンドルとともに割り当てられるため、ハンドルを解放するとユーザーのデータ構造も解放されます。

記述子

OCI の記述子およびロケータは、データ固有の情報をメンテナンスする不透明なデータ構造体です。これらの記述子とロケータ、C データ型、およびその型の記述子を `OCIDescriptorAlloc()` のコール内で割り当てるために使用する OCI 型定数を次の表に示します。`OCIDescriptorFree()` 関数は、記述子とロケータを解放します。

表 2-2 記述子タイプ

説明	C 型	OCI 型定数
スナップショット記述子	<code>OCISnapshot</code>	<code>OCI_DTYPE_SNAP</code>
LOB データ型ロケータ	<code>OCILobLocator</code>	<code>OCI_DTYPE_LOB</code>
FILE データ型ロケータ	<code>OCILobLocator</code>	<code>OCI_DTYPE_FILE</code>
読取り専用パラメータ記述子	<code>OCIParam</code>	<code>OCI_DTYPE_PARAM</code>
ROWID 記述子	<code>OCIRowid</code>	<code>OCI_DTYPE_ROWID</code>
ANSI DATE 記述子	<code>OCIDateTime</code>	<code>OCI_DTYPE_DATE</code>
TIMESTAMP 記述子	<code>OCIDateTime</code>	<code>OCI_DTYPE_TIMESTAMP</code>
TIMESTAMP WITH TIME ZONE 記述子	<code>OCIDateTime</code>	<code>OCI_DTYPE_TIMESTAMP_TZ</code>
TIMESTAMP WITH LOCAL TIME ZONE 記述子	<code>OCIDateTime</code>	<code>OCI_DTYPE_TIMESTAMP_LTZ</code>
INTERVAL YEAR TO MONTH 記述子	<code>OCIInterval</code>	<code>OCI_DTYPE_INTERVAL_YM</code>
INTERVAL DAY TO SECOND 記述子	<code>OCIInterval</code>	<code>OCI_DTYPE_INTERVAL_DS</code>
複合オブジェクト記述子	<code>OCIComplexObjectComp</code>	<code>OCI_DTYPE_COMPLEXOBJECTCOMP</code>
アドバンスト・キューイング・エンキュー・オプション	<code>OCIAQEnqOptions</code>	<code>OCI_DTYPE_AQENQ_OPTIONS</code>
アドバンスト・キューイング・デキュー・オプション	<code>OCIAQDeqOptions</code>	<code>OCI_DTYPE_AQDEQ_OPTIONS</code>
アドバンスト・キューイング・メッセージ・プロパティ	<code>OCIAQMsgProperties</code>	<code>OCI_DTYPE_AQMSG_PROPERTIES</code>
アドバンスト・キューイング・エージェント	<code>OCIAQAgent</code>	<code>OCI_DTYPE_AQAGENT</code>
アドバンスト・キューイング通知	<code>OCIAQNotify</code>	<code>OCI_DTYPE_AQNFY</code>
登録要求でのデータベース・サーバーの識別名	<code>OCIServerDNs</code>	<code>OCI_DTYPE_SRVDN</code>

注意： **OCILobLocator** に対する C 型は 1 つのみですが、このロケータは内部 LOB および外部 LOB に対しては異なる OCI 型定数で割り当てられます。後述の LOB ロケータの項で、この違いについて説明します。

各記述子型の主な用途は次のとおりです。各記述子型については、後述の項で説明します。

- **OCISnapshot** — 文の実行時に使用
- **OCILOBLocator** — LOB コール (**OCI_DTYPE_LOB**) または FILE コール (**OCI_DTYPE_FILE**) で使用
- **OCIParam** — 記述コールで使用
- **OCIRowid** — ROWID 値のバインドまたは定義で使用
- **OCIDateTime** および **OCIInterval** — 日時および時間隔データ型で使用
- **OCIComplexObjectComp** — 複合オブジェクト検索で使用
- **OCIAQEnqOptions**、**OCIAQDeqOptions**、**OCIAQMsgProperties**、**OCIAQAgent** — アドバンスド・キューイングで使用
- **OCIAQNotify** — パブリッシュ・サブスクライブ通知で使用
- **OCIServerDNs** — LDAP ベースのパブリッシュ・サブスクライブ通知で使用

スナップショット記述子

スナップショット記述子は、**OCISstmtExecute()** のコールを実行するためのオプション・パラメータです。この記述子は、特定のデータベース・スナップショットに対する問合せが実行されることを示します。データベース・スナップショットは、ある特定の時点におけるデータベースの状態を表します。

スナップショット記述子は、**OCIDescriptorAlloc()** のコールで、**OCI_DTYPE_SNAP** を *type* パラメータとして渡すことによって割り当てられます。

関連項目： **OCISstmtExecute()** およびデータベース・スナップショットの詳細は、4-7 ページの「[実行スナップショット](#)」を参照してください。

LOB/FILE データ型ロケータ

LOB (ラージ・オブジェクト) は、最大 4GB (ギガバイト) のバイナリ (BLOB) または文字 (CLOB) データを保持できる Oracle データ型です。データベース内では、LOB ロケータと呼ばれる不透明なデータ構造が、データベース行の LOB 列、あるいはオブジェクトの LOB 属性の位置に格納されます。このロケータは、別の位置に格納されている実際の LOB 値を指すポインタとして機能します。

OCILOB ロケータは、LOB (BLOB または CLOB) または FILE (BFILE) に対する OCI 操作の実行に使用されます。OCILOB* 関数では、LOB 値のかわりに LOB ロケータがパラメータとして使用されます。OCILOB 関数では、パラメータとして実際の LOB データは使用されません。この関数では、パラメータとして LOB ロケータを使用し、LOB ロケータによって参照される LOB データを操作します。

したがって、旧式のインタフェースでは、実際の LOB 値を操作できます。この記述子 **OCILOBLocator** は、FILE への操作にも使用されます。

LOB ロケータは、OCIDescriptorAlloc() のコールで、BLOB または CLOB の *type* パラメータとしての OCI_DTYPE_LOB、および BFILE の OCI_DTYPE_FILE を渡すことによって割り当てられます。

注意： これらの 2 つの LOB ロケータ型を互いに交換することはできません。BLOB か CLOB をバインドまたは定義するとき、アプリケーションは、OCI_DTYPE_LOB を使用してロケータが適切に割り当てられているか注意する必要があります。同様に、BFILE をバインドまたは定義するとき、アプリケーションは、OCI_DTYPE_FILE を使用してロケータが割り当てられていることを確認する必要があります。

OCI アプリケーションでは、選択リストの要素として LOB 列または属性を含む SQL 文を発行することによって、サーバーから LOB ロケータを取り出せます。その場合、アプリケーションは最初に LOB ロケータを割り当て、次にそれを使用して出力変数を定義します。同様に、LOB ロケータは、SQL 文で LOB とプレースホルダとの間の関連付けを行うバインド操作の一部として使用できます。

LOB ロケータのデータ型 (**OCILOBLocator**) は、Oracle7 Server に接続している場合にはデータ型として無効です。

関連項目： OCILOB 操作の詳細は、[第 7 章「LOB と FILE の操作」](#) を参照してください。

パラメータ記述子

OCI アプリケーションは、パラメータ記述子を使用して、選択リスト列またはスキーマ・オブジェクトについての情報を取得します。この情報は、記述操作を通して取得されます。

パラメータ記述子の割当てが、`OCIDescriptorAlloc()` を使用して行われることはありません。パラメータ記述子は、`OCIParamGet()` コールを使用してパラメータの位置を指定することによって、記述ハンドルまたは文ハンドル、複合オブジェクト検索ハンドルの属性としてのみ取得できます。

関連項目： パラメータ記述子の取得と使用方法の詳細は、[第 6 章「スキーマ・メタデータの記述」](#) および 4-12 ページの「[選択リスト項目の記述](#)」を参照してください。

ROWID 記述子

ROWID 記述子 (**OCIRowid**) は、Oracle ROWID を取り出して使用する必要があるアプリケーションで使用されます。Oracle7 から Oracle8 へのアップグレードで ROWID のサイズと構造が変更されており、ユーザーにはわかりにくくなっています。リリース 8.x 以上の OCI を使用して ROWID を操作するために、アプリケーションでは SQL 選択リスト内の ROWID の位置に対応する ROWID 記述子を定義し、その記述子内の ROWID を検索できます。この同じ記述子を、後で INSERT 文または WHERE 句の入力変数にバインドできます。

また、文の実行後に文ハンドルで `OCIAttrGet()` を使用し、ROWID を記述子にリダイレクトできます。

日時および時間隔の記述子

これらの記述子は、日時または時間隔データ型 (**OCIDateTime** および **OCIInterval**) を使用するアプリケーションで使用されます。これらの記述子は、バインドおよび定義で使われ、メモリーを割り当てたり解放する関数の `OCIDescAlloc()` および `OCIDescFree()` にパラメータとして渡されます。

関連項目： データ型の詳細は、[第 3 章「データ型」](#) を参照してください。データ型を操作する関数のリストは、[第 18 章「OCI のデータ型マッピング関数および操作関数」](#) を参照してください。

注意： **OCIDateTime** および **OCIInterval** データ型を操作する関数は、**OCIDate** データ型も操作します。

複合オブジェクト記述子

複合オブジェクト記述子およびその使用方法の詳細は、10-20 ページの「[複合オブジェクト検索](#)」を参照してください。

アドバンスト・キューイング記述子

アドバンスト・キューイング記述子および関連する記述子の詳細は、9-48 ページの「[OCI およびアドバンスト・キューイング](#)」を参照してください。

LDAP ベースのパブリッシュ・サブスクリाइブ通知

LDAP ベースのパブリッシュ・サブスクリाइブ通知の詳細は、9-53 ページの「[パブリッシュ・サブスクリाइブの登録関数](#)」を参照してください。

ユーザー・メモリーの割当て

`OCIDescriptorAlloc()` コールには、そのパラメータ・リストに `xtrmem_sz` パラメータが含まれています。このパラメータは、記述子またはロケータとともに割り当てるユーザー・メモリー量を指定するために使用します。

通常、アプリケーションは、このパラメータを使用して、記述子またはロケータと同じ存続期間を持つアプリケーション定義の構造体を割り当てます。この構造体は、アプリケーション・ブックキーピングまたはコンテキスト情報の格納に使用できます。

`xtrmem_sz` パラメータを使用すると、アプリケーションでは、各記述子またはロケータを割り当てたり、割当て解除するたびに、メモリーを明示的に割り当てたり、割当て解除する必要がありません。メモリーは記述子またはロケータとともに割り当てられるため、記述子またはロケータを解放すると (`OCIDescriptorFree()` により)、ユーザーのデータ構造も解放されます。

`OCIHandleAlloc()` コールには、ハンドルと同じ存続期間を持つユーザー・メモリーを割り当てるための同様のパラメータがあります。

`OCIEnvCreate()` コールおよび `OCIEnvInit()` コールには、環境ハンドルと同じ存続期間を持つユーザー・メモリーを割り当てるための同様のパラメータがあります。

OCI プログラミング・ステップ

この後の各項では、OCI アプリケーションで実行する各ステップについて詳しく説明します。いくつかのステップは、オプションです。たとえば、文が問合せでない場合には、選択リスト項目を記述または定義する必要はありません。

注意： SQL 文を処理する OCI コールの使用方法の例は、[付録 B「OCI デモ・プログラム」](#)の最初のサンプル・プログラムを参照してください。

関連項目：

- 実行時に動的にデータを供給する特殊なケースの詳細は、5-44 ページの「[ランタイム・データ割当てとピース単位操作](#)」を参照してください。
- 構造の配列を伴う操作の詳細は、5-25 ページの「[構造体配列のバインドと定義](#)」を参照してください。
- OCI プログラム内での SQL 文の処理に伴うステップのアウトラインは、2-31 ページの「[エラー処理](#)」を参照してください。
- OCI を使用したマルチスレッド・アプリケーションのプログラミングの詳細は、9-2 ページの「[スレッド・セーフティ](#)」を参照してください。
- SQL 文の種類の詳細は、1-7 ページの「[SQL 文](#)」を参照してください。

次の各項では、OCI のアプリケーションに必要なステップについて説明します。

- [OCI 環境の初期化](#)
- [SQL 文の処理](#)
- [コミットまたはロールバック](#)
- [アプリケーションの終了](#)
- [エラー処理](#)

また、OCI 関数のいずれか、またはすべてのステップ間で、アプリケーション固有の処理も発生します。

7.x アップグレードの注意： OCI のプログラマは、明示的な解析ステップが OCI プログラムで不要になったことに注意してください。これは、リリース 8.0 以上のアプリケーションでは、DML 文と DDL 文の両方に対して実行コマンドを発行する必要があることを意味します。

OCI 環境の初期化

この項では、OCI 環境を初期化してサーバーとの接続を確立し、ユーザーがデータベースに対してアクションを実行できるように許可する方法を説明します。

次の3つの項では、OCI 環境を初期化するための3つの主要なステップについて説明します。

1. OCI 環境の作成
2. ハンドルおよび記述子の割当て
3. アプリケーションの初期化、接続およびセッション

OCI 環境の作成

各 OCI ファンクション・コールは、`OCIEnvCreate()` コールで作成された環境のコンテキスト内で実行されます。このコールは、他の OCI コールよりも先に行う必要があります。唯一の例外は、OCI 共有モードのプロセスレベルの属性を設定する場合です。

関連項目： 2-22 ページ「共有データ・モード」

`OCIEnvCreate()` の `mode` パラメータは、OCI ライブラリ関数をコールするアプリケーションが次のことを行うかどうか指定します。

- スレッド化環境での実行 (`mode = OCI_THREADED`)
- オブジェクトの使用 (`mode = OCI_OBJECT`)
- 共有データ構造の使用 (`mode = OCI_SHARED`)
- サブスクリプションの使用 (`mode = OCI_EVENTS`)

これらのモードは、環境ごとに設定できます。

アプリケーションがオブジェクトのバインドおよび定義を行っている場合、または OCI のオブジェクト・ナビゲーション・コールを使用している場合は、オブジェクト・モードでの初期化が必要です。また、アプリケーションでこれらの機能をすべて使用しないようにしたり (`mode = OCI_DEFAULT`)、縦線で区切って組み合わせて使用することもできます。たとえば、`mode = (OCI_THREADED | OCI_OBJECT)` を指定すると、アプリケーションはスレッド化環境で動作し、オブジェクトを使用します。

また、OCI 環境ごとにユーザー定義のメモリー管理関数を指定できます。

注意： 以前のリリースでは、個別のプロセスレベルの初期化を明示的に行う必要がありました。この初期化は自動化されたため、明示的なプロセスレベルの初期化は必要ありません。

関連項目：

- 初期化コールの詳細は、15-9 ページの「[OCIEnvCreate\(\)](#)」の説明および 15-18 ページの「[OCIInitialize\(\)](#)」の説明を参照してください。
- OCI を使用したマルチスレッド・アプリケーションのプログラミングの詳細は、9-2 ページの「[スレッド・セーフティ](#)」を参照してください。
- オブジェクトを使用した OCI プログラミングの詳細は、[第 10 章「OCI オブジェクト・リレーショナル・プログラミング」](#) およびそれ以降の章を参照してください。
- パブリッシュ / サブスクライブ機能の使用方法的詳細は、9-52 ページの「[パブリッシュ・サブスクライブの通知](#)」を参照してください。

共有データ・モード

SQL 文が処理されると、基礎となる特定のデータが文に関連付けられます。このデータには、問合せの定義情報および記述情報とともに、文のテキスト・データおよびバインド・データに関する情報が格納されています。同一の SQL 文の集合が、同一のホスト上にあるアプリケーションの複数のインスタンスで実行される場合は、それらのアプリケーションでこのデータを共有できます。

OCI アプリケーションを共有モードで初期化すると、共通の文データが複数の文ハンドルで共有されるため、アプリケーションのメモリの節約になります。このメモリの節約は、複数の文ハンドルを作成するアプリケーションで特に有効です。このようなアプリケーションでは、同一または複数の接続で、同じ SQL 文が、同一のスキーマ内の複数のユーザー・セッションで実行されるためです。

共有モード機能を使用しない場合、OCI 文ハンドルを使用した問合せの実行において、メタデータを格納するために各実行に専用のメモリが必要になります。その場合、必要なメモリー量の合計は、すべてのプロセスで実行される文の数と、各文ハンドルに必要なメモリー量を乗算した値とほぼ同じになります。

共有モード機能を使用すると、文ハンドルの共通メモリーの大部分が、同じ文を実行するすべてのプロセスで共有されます。このため、すべてのプロセスで使用されるメモリー合計量は、前述のケースとプロセス数が同じ場合でも、大幅に小さくなります。共有モードで実行される文の数が増えるにつれて、文ハンドルごとに必要なメモリー要件は、共有モードを使用しないケースに比べて大幅に小さくなります。

共有データ構造モードは、次の場合に有効です。

- 同一アプリケーションの複数のインスタンスが同一マシン上で実行され、複数のクライアントにサービスする場合。これらの各インスタンスでは、同一 SQL 文を実行し、固有のバインド値で区別されます。
- 1 つまたは複数の接続で、複数のユーザーに対して同一の文を実行するサービス・スレッドが、アプリケーション・プロセスによって分岐される場合。このシナリオでも、前述と同じメモリの節約を実現できます。
- アプリケーションの種類が、SQL ドライバおよび他の中間層アプリケーションである場合。

注意： 複数の単一問合せを同時に実行しない小規模のアプリケーションでは、この機能の利点を生かすことができません。

OCI の共有機能を使用するにはいくつかの方法があります。既存のアプリケーションの場合は、コードを変更することなく、この機能の利点を簡単に確認できます。これらのアプリケーションは、環境変数を設定することにより、OCI 共有モードで初期化できます。新しいアプリケーションでは、OCI API コールを使用して共有モード機能を初期化する必要があります。

OCI 関数の使用方法

OCI 共有モード機能を初期化するには、プロセス・ハンドル・パラメータを設定し、モード・フラグを OCI_SHARED に設定して OCIEnvCreate() をコールする必要があります。次に例を示します。

```
ub4 mode = OCI_SHARED | OCI_THREADED;  
OCIEnvCreate (&envhp, mode, (CONST dvoid *)0, 0, 0, 0, (size_t)0, (dvoid **)0);
```

最初のアプリケーションでは、OCI を共有モードで初期化し、その OCI アプリケーションによって設定されたパラメータを使用して、共有サブシステムを起動します。後続のアプリケーションは共有モードで初期化され、直前に起動された共有サブシステムを使用します。

関連項目： OCI 共有モード・システム用に設定と読み込みが可能なパラメータの詳細は、A-75 ページの「プロセス・ハンドル属性」を参照してください。

OCI アプリケーションが共有モードで初期化されている場合、準備および実行されるすべての文では、デフォルトで共有サブシステムを使用します。特定の SQL 文の実行に共有サブシステムを使用しない場合は、`OCIStmtPrepare()` で `OCI_NO_SHARING` フラグを使用できます。次に例を示します。

```
OCIStmtPrepare(stmthp, (CONST text *)createstmt,
               (ub4)strlen((char *)updstmt), (ub4)OCI_NTV_SYNTAX,
               (ub4)OCI_NO_SHARING);
```

`OCI_NO_SHARING` フラグは、プロセスが共有モードで初期化されていない場合は無効です。

関連項目： 16-15 ページ「[OCIStmtPrepare\(\)](#)」

プロセスを共有メモリー・サブシステムから連結解除するには、`OCITerminate()` コールを使用します。

関連項目： 15-46 ページ「[OCITerminate\(\)](#)」

環境変数の使用方法

環境変数 `OCI_SHARED_MODE` および `OCI_NUM_SHARED_PROCS` は、OCI 共有モード機能の設定に使用できます。ただし、この方法はお薦めしません。この手順は、既存のアプリケーションで共有モード機能を確認するためのものです。

OCI_SHARED_MODE OCI アプリケーションを初期化して共有モードで実行するには、OCI プログラムを実行する前に、環境変数 `OCI_SHARED_MODE` を設定します。たとえば、Solaris オペレーティング環境で C シェルの変数を設定するには、次のコマンドを発行します。

```
setenv OCI_SHARED_MODE number
```

`number` は共有メモリー・アドレス空間のサイズです。次に例を示します。

```
setenv OCI_SHARED_MODE 20000000
```

共有サブシステムが動作していない場合は、この変数を設定すると、指定したサイズで共有メモリー・アドレス空間が作成され、サブシステムが起動します。必要な共有メモリーのサイズは、アプリケーションの性質によって決まり、SQL 文および SQL 文からアクセスされる基礎となる表のサイズと種類によって変わります。

OCI_NUM_SHARED_PROCS

共有サブシステムに接続可能なプロセスの最大数を設定するには、環境変数 `ORA_OCI_NUM_SHARED_PROCS` を設定します。この変数を設定するには、次のコマンドを発行します。

```
setenv OCI_NUM_SHARED_PROCS number
```

`number` は、プロセスの最大数です。次に例を示します。

```
setenv OCI_NUM_SHARED_PROCS 20
```

`ORA_OCI_NUM_SHARED_PROCS` は、共有サブシステムを起動するための初期化パラメータです。この変数は、共有サブシステムがすでに実行中の場合は無効です。

ハンドルおよび記述子の割当て

Oracle は、ハンドルと記述子を割り当てたり割当て解除するための OCI 関数を提供します。ハンドルは、`OCIHandleAlloc()` を使用して割り当てた後、OCI コールに渡す必要があります (`OCIBindByPos()` のように、OCI コールがユーザーにかわってハンドルを割り当てる場合は除きます)。

`OCIHandleAlloc()` を使用して、次の種類のハンドルを割り当てることができます。

- エラー・ハンドル
- サービス・コンテキスト・ハンドル
- 文ハンドル
- 記述ハンドル
- サーバー・ハンドル
- ユーザー・セッション・ハンドル
- トランザクション・ハンドル
- 接続プール・ハンドル
- 複合オブジェクト検索ハンドル
- サブスクリプション・ハンドル
- ダイレクト・パス・コンテキスト・ハンドル
- ダイレクト・パス列配列ハンドル
- ダイレクト・パス・ストリーム・ハンドル

アプリケーションの機能に応じて、これらのハンドルのいくつかまたはすべてを割り当てる必要があります。

関連項目： 15-56 ページの「[OCIHandleAlloc\(\)](#)」の説明を参照してください。

アプリケーションの初期化、接続およびセッション作成

アプリケーションでは、`OCIEnvCreate()` をコールして OCI 環境ハンドルを初期化する必要があります。

このステップに従ってサーバー接続を確立し、ユーザー・セッションを開始する場合、シングル・ユーザーで単一接続である場合と、複数セッションまたは複数接続である場合の、2通りのオプションがあります。

注意： `OCIEnvCreate()` は、`OCIInitialize()` コールおよび `OCIEnvInit()` コールのかわりに使用する必要があります。
`OCIInitialize()` および `OCIEnvInit()` コールは、下位互換性を保つためにサポートされています。

オプション 1: シングル・ユーザー、単一接続

このオプションは、単純化されたログイン方法です。

アプリケーションがデータベース接続ごとに常時シングル・ユーザー・セッションのみをメンテナンスする場合、このアプリケーションは OCI の単純化されたログイン・プロシージャを利用できます。

アプリケーションから `OCILogon()` をコールすると、OCI ライブラリは、渡されたサービス・コンテキスト・ハンドルを初期化し、次に、関数にユーザー名とパスワードを渡したユーザー用に、指定されたサーバーとの接続を作成します。

`OCILogon()` のコール例を次に示します。

```
OCILogon(envhp, errhp, &svchp, "scott", nameLen, "tiger",  
        passwdLen, "oracledb", dbnameLen);
```

このコールのパラメータには、接続を確立するために使用するサービス・コンテキスト・ハンドル（初期化されます）、ユーザー名、ユーザーのパスワードおよびデータベース名を含めることができます。また、この関数によって、サーバー・ハンドルおよびユーザー・セッション・ハンドルも暗黙的に割り当てられます。

アプリケーションがこのログイン方法を使用する場合、サービス・コンテキスト、サーバーおよびユーザー・セッション・ハンドルはすべて読取り専用となります。これは、`OCIAttrSet()` でサービス・コンテキスト・ハンドルの該当する属性を変更することにより、アプリケーションからセッションまたはトランザクションを切り替えられないことを意味します。

アプリケーションで `OCILogon()` を使用してセッションおよび認可を初期化した場合は、`OCILogoff()` を使用してそれらを終了する必要があります。

オプション 2: 複数セッションまたは複数接続

このオプションでは、明示的な連結（アタッチ）コールおよび開始セッション・コールを使用します。

アプリケーションで、データベース接続に複数のユーザー・セッションをメンテナンスする必要がある場合は、別のコールのセットを使用してセッションおよび接続を設定してください。これには、サーバーおよび開始セッションに連結するための特定のコールが含まれます。

- `OCIServerAttach()` では、OCI 操作用のデータ・サーバーにアクセスするためのアクセス・パスを作成します。
- `OCISessionBegin()` では、特定のサーバーに対するユーザーのセッションを確立します。ユーザーがサーバーに対する操作を実行するには、このコールが必須です。

注意： `OCIServerAttach()` コールでブロック化接続または非ブロック化接続を指定する場合の詳細は、2-41 ページの「[非ブロック化モード](#)」を参照してください。

これらのコールにより、データベースに対して SQL 文および PL/SQL 文を実行できる操作環境が設定されます。これらのコールを実行する前に、必ずデータベースを起動して実行状態にしておく必要があります。そうしないと、これらのコールは失敗します。

関連項目： これらのコールの詳細は、15-4 ページの「[接続関数、認証関数および初期化関数](#)」を参照してください。複数セッション、複数トランザクションおよび複数接続のメンテナンスの詳細は、[第 9 章「OCI プログラミングの高度なトピック」](#)を参照してください。

OCI 環境の作成および初期化の例

OCI 環境の作成および初期化の使用例を次に示します。この例では、サーバー・コンテキストを作成し、それをサービス・ハンドルに設定します。次に、ユーザー・セッション・ハンドルを作成し、データベース・ユーザー名とパスワードを使用してそれを初期化します。簡潔にするため、エラー・チェックは含まれていません。

```
#include <s.h>
#include <oci.h>
...
main()
{
    ...
    OCIEnv *myenvhp;      /* the environment handle */
    OCIServer *mysrvhp;    /* the server handle */
    OCIError *myerrhp;     /* the error handle */
    OCISession *myusrhp;   /* user session handle */
    OCISvcCtx *mysvcchp;   /* the service handle */
```

```
...
/* initialize the mode to be the threaded and object environment */
(void) OCIEnvCreate(&myenvhp, OCI_THREADED|OCI_OBJECT, (dvoid *)0,
                  0, 0, 0, (size_t) 0, (dvoid **)0);

/* allocate a server handle */
(void) OCIHandleAlloc ((dvoid *)myenvhp, (dvoid **)&mysrvhp,
                     OCI_HTYPE_SERVER, 0, (dvoid **) 0);

/* allocate an error handle */
(void) OCIHandleAlloc ((dvoid *)myenvhp, (dvoid **)&myerrhp,
                     OCI_HTYPE_ERROR, 0, (dvoid **) 0);

/* create a server context */
(void) OCIServerAttach (mysrvhp, myerrhp, (text *)"inst1_alias",
                      strlen ("inst1_alias"), OCI_DEFAULT);

/* allocate a service handle */
(void) OCIHandleAlloc ((dvoid *)myenvhp, (dvoid **)&mysvchp,
                     OCI_HTYPE_SVCCTX, 0, (dvoid **) 0);

/* set the server attribute in the service context handle*/
(void) OCIAttrSet ((dvoid *)mysvchp, OCI_HTYPE_SVCCTX,
                  (dvoid *)mysrvhp, (ub4) 0, OCI_ATTR_SERVER, myerrhp);

/* allocate a user session handle */
(void) OCIHandleAlloc ((dvoid *)myenvhp, (dvoid **)&myusrhp,
                     OCI_HTYPE_SESSION, 0, (dvoid **) 0);

/* set username attribute in user session handle */
(void) OCIAttrSet ((dvoid *)myusrhp, OCI_HTYPE_SESSION,
                  (dvoid *)"scott", (ub4)strlen("scott"),
                  OCI_ATTR_USERNAME, myerrhp);

/* set password attribute in user session handle */
(void) OCIAttrSet ((dvoid *)myusrhp, OCI_HTYPE_SESSION,
                  (dvoid *)"tiger", (ub4)strlen("tiger"),
                  OCI_ATTR_PASSWORD, myerrhp);

(void) OCISessionBegin ((dvoid *) mysvchp, myerrhp, myusrhp,
                      OCI_CRED_RDBMS, OCI_DEFAULT);

/* set the user session attribute in the service context handle*/
(void) OCIAttrSet ( (dvoid *)mysvchp, OCI_HTYPE_SVCCTX,
                  (dvoid *)myusrhp, (ub4) 0, OCI_ATTR_SESSION, myerrhp);
...
}
```

demo ディレクトリ内のデモ・プログラム `cdemo81.c` は、このプロセスとエラー・チェックを例示しています。

SQL 文の処理

SQL 文の処理の詳細は、[第 4 章「OCI での SQL 文の使用」](#)を参照してください。

コミットまたはロールバック

アプリケーションは、`OCITransCommit()` をコールすることにより、データベースへの変更をコミットします。このコールは、サービス・コンテキストをパラメータの 1 つとして受け取ります。サービス・コンテキストに現在関連付けられているトランザクションは、変更がコミット済みです。このトランザクションは、アプリケーションで明示的に作成されたか、または、アプリケーションがデータベースを変更したときに暗黙的に作成されたトランザクションである場合があります。

注意： `OCIExecute()` コールの `OCI_COMMIT_ON_SUCCESS` モードを使用すると、各文の実行終了時にコミットするトランザクションをアプリケーションで選択できるため、ラウンドトリップを節約できます。

トランザクションをロールバックするには、`OCITransRollback()` コールを使用します。

通常のログオフ以外のなんらかの方法でアプリケーションと Oracle の接続が切断された場合（たとえばネットワークとの接続が切れるなど）、`OCITransCommit()` をコールしていないと、アクティブ・トランザクションはすべて自動的にロールバックされます。

関連項目： 暗黙的なトランザクションおよびトランザクション処理の詳細は、2-9 ページの「[サービス・コンテキスト・ハンドルとそれに対応付けられたハンドル](#)」および 8-2 ページの「[OCI でのトランザクションのサポート](#)」を参照してください。

アプリケーションの終了

OCI アプリケーションは、終了する前に、次の 3 つのステップを実行してください。

1. 各セッションに対して `OCISessionEnd()` をコールし、ユーザー・セッションを終了します。
2. 各データ・ソースに対して `OCIserverDetach()` をコールし、データ・ソースへのアクセスを終了します。
3. 各ハンドルに対して `OCIHandleFree()` をコールし、明示的に全ハンドルの割当てを解除します。
4. 環境ハンドルを削除し、環境ハンドルに関連付けられたその他のハンドルの割当てをすべて解除します。

注意： 親ハンドルが解放されると、それに対応付けられた子ハンドルもすべて自動的に解放されます。

`OCIserverDetach()` および `OCISessionEnd()` のコールは、必須ではありませんが、お勧めします。アプリケーションが `OCITransCommit()` (トランザクション・コミット) をコールしないで終了すると、保留状態のトランザクションはすべて自動的にロールバックされます。

関連項目： アプリケーションの終了時に解放されるハンドルを示した例は、[付録 B「OCI デモ・プログラム」](#) の最初のサンプル・プログラムを参照してください。

注意： アプリケーションで `OCILogon()` の単純化されたログイン方法を使用した場合は、`OCILogout()` をコールするとセッションが終了し、サーバーとの接続が切断され、サービス・コンテキスト・ハンドルとそれに関連付けられたハンドルが解放されます。ただし、アプリケーションにより割り当てた他のハンドルは、そのアプリケーションから解放してください。

エラー処理

OCI ファンクション・コールには、表 2-3「OCI リターン・コード」にリストされた一連のリターン・コードが用意されています。これらのコードにより、コールの成功または失敗（OCI_SUCCESS、OCI_ERROR など）や、アプリケーションが必要とするその他の情報（OCI_NEED_DATA、OCI_STILL_EXECUTING など）が示されます。大部分の OCI コールは、これらのコードの 1 つを戻します。

関連項目： 例外は、2-34 ページの「他の値を戻す関数」を参照してください。

表 2-3 OCI リターン・コード

OCI リターン・コード	説明
OCI_SUCCESS	関数は正常に終了しました。
OCI_SUCCESS_WITH_INFO	関数は正常に終了しました。OCIErrorGet () をコールすると、追加診断情報が戻されます。これには、警告が含まれる場合があります。
OCI_NO_DATA	関数が終了しました。これ以上データはありません。
OCI_ERROR	関数が失敗しました。OCIErrorGet () をコールすると、エラーの追加情報が戻されます。
OCI_INVALID_HANDLE	無効なハンドルがパラメータとして渡されたか、ユーザー・コールバックで無効なハンドルまたは無効なコンテキストが渡されました。追加診断情報ははありません。
OCI_NEED_DATA	アプリケーションで、ランタイム・データを提供する必要があります。
OCI_STILL_EXECUTING	サービス・コンテキストが非ブロック化モードで確立されたため、現行の操作は即時完了できませんでした。この操作を完了するには、これを再度コールする必要があります。OCIErrorGet () がエラー・コードとして ORA-03123 を戻します。
OCI_CONTINUE	このコードはコールバック関数からのみ戻されます。これは、コールバック関数が、OCI ライブラリの標準処理再開を示唆していることを示します。

エラーが発生したことがリターン・コードに示されている場合、アプリケーションでは OCIErrorGet () をコールして、Oracle 固有のエラー・コードおよびメッセージを取り出せます。OCIErrorGet () へのパラメータの 1 つは、エラーが発生したコールに渡されたエラー・ハンドルです。

注意： レコードがなくなる（OCI_NO_DATA が戻される）まで、繰り返し OCIErrorGet () をコールすると、複数の診断レコードを取り出すことができます。OCIErrorGet () は、常に最大 1 個の診断レコードを戻します。

次のコード例では、エラー・ハンドルの渡されるとエラー情報が戻され、OCI ファンクション・コールからはリターン・コードが戻されます。リターン・コードが `OCI_ERROR` の場合は、関数は診断情報を出力します。`OCI_SUCCESS` の場合、出力はありません。その他のリターン・コードの場合は、リターン・コード情報が出力されます。

```
STATICF void checkerr(errhp, status)
OCIError *errhp;
sword status;
{
    text errbuf[512];
    ub4 buflen;
    ub4 errcode;

    switch (status)
    {
        case OCI_SUCCESS:
            break;
        case OCI_SUCCESS_WITH_INFO:
            (void) printf("Error - OCI_SUCCESS_WITH_INFO\n");
            break;
        case OCI_NEED_DATA:
            (void) printf("Error - OCI_NEED_DATA\n");
            break;
        case OCI_NO_DATA:
            (void) printf("Error - OCI_NODATA\n");
            break;
        case OCI_ERROR:
            (void) OCIErrorGet (errhp, (ub4) 1, (text *) NULL, &errcode,
                               errbuf, (ub4) sizeof(errbuf), OCI_HTYPE_ERROR);
            (void) printf("Error - %s\n", errbuf);
            break;
        case OCI_INVALID_HANDLE:
            (void) printf("Error - OCI_INVALID_HANDLE\n");
            break;
        case OCI_STILL_EXECUTING:
            (void) printf("Error - OCI_STILL_EXECUTE\n");
            break;
        default:
            break;
    }
}
```

切捨ておよび NULL データのリターン・コードおよびエラー・コード

表 2-4、表 2-5 および表 2-6 に、フェッチされたデータが NULL の場合または切り捨てられた場合の OCI リターン・コード、Oracle エラー番号、インジケータ変数および列リターン・コードを示します。

関連項目： インジケータ変数の詳細は、2-36 ページの「[インジケータ変数](#)」を参照してください。

表 2-4 通常データ – 非 NULL または切捨てなし

リターン・コード	インジケータ – 戻さない場合	インジケータ – 戻す場合
戻さない場合	OCI_SUCCESS error = 0	OCI_SUCCESS error = 0 indicator = 0
戻す場合	OCI_SUCCESS error = 0 return code = 0	OCI_SUCCESS error = 0 indicator = 0 return code = 0

表 2-5 NULL データ

リターン・コード	インジケータ – 戻さない場合	インジケータ – 戻す場合
戻さない場合	OCI_ERROR error = 1405	OCI_SUCCESS error = 0 indicator = -1
戻す場合	OCI_ERROR error = 1405 return code = 1405	OCI_SUCCESS error = 0 indicator = -1 return code = 1405

表 2-6 切捨てデータ

リターン・コード	インジケータ – 戻さない場合	インジケータ – 戻す場合
戻さない場合	OCI_ERROR error = 1406	OCI_ERROR error = 1406 indicator = data_len
戻す場合	OCI_SUCCESS_WITH_INFO error = 24345 return code = 1405	OCI_SUCCESS_WITH_INFO error = 24345 indicator = data_len return code = 1406

表 2-6 の data_len は、長さが SB2MAXVAL 以下であった場合に切り捨てられたデータの実際の長さです。この値を超えていれば、インジケータが -2 に設定されます。

他の値を戻す関数

一部の関数では、表 2-3 にリストされている OCI エラー・コード以外の値を戻します。これらの関数を使用する場合は必ず、OUT パラメータからではなく、ファンクション・コールから直接値が戻ることを考慮してください。それぞれの関数と戻り値の詳細は、該当する章を参照してください。次のような関数例があります。

- OCICollMax()
- OCIRawPtr()
- OCIRawSize()
- OCIRefHexSize()
- OCIRefIsEqual()
- OCIRefIsNull()
- OCIStrPtr()
- OCIStrSize()

その他のコーディング・ガイドライン

この項では、Oracle Call Interface を使用してアプリケーションをコーディングする際に注意が必要な、その他の要因を説明します。

パラメータの型

OCI 関数には整数、ハンドル、文字列など、多様なデータ型のパラメータを使用することができます。パラメータの型によっては注意事項があり、以降の項で説明します。

関連項目： パラメータのデータ型およびパラメータを渡す際の規則の詳細は、15-4 ページの「[接続関数、認証関数および初期化関数](#)」を参照してください。

アドレス・パラメータ

アドレス・パラメータでは、変数のアドレスを Oracle に渡します。C では、通常スカラー・パラメータを値で渡すので、C で開発する場合は、パラメータがアドレスであることに注意してください。すべてのケースで、ポインタは注意して渡す必要があります。

整数パラメータ

2 進整数パラメータは、サイズがシステムによって異なる数値です。short 型 2 進整数パラメータも、サイズがシステムによって異なる数値ですが、サイズはより短くなります。使用しているシステムでのこれらの整数のサイズについては、システム固有の Oracle マニュアルを参照してください。

文字列パラメータ

文字列は、アドレス・パラメータの特殊な型です。この項では、文字列アドレス・パラメータに適用される追加規則について説明します。

文字列をパラメータとして渡すことができる各 OCI ルーチンには、文字列長のパラメータもあります。このパラメータで文字列の長さを設定する必要があります。

7.x アップグレードの注意： OCI のこれまでのバージョンとは異なり、ヌル文字で終了する文字列の文字列長パラメータに -1 を渡すことはできません。

列への NULL の挿入

データベース列に NULL を挿入する方法はいくつかあります。1 つは、INSERT 文または UPDATE 文のテキストの中でリテラル NULL を使用する方法です。たとえば、次のような SQL 文があるとします。

```
INSERT INTO emp (ename, empno, deptno)
VALUES (NULL, 8010, 20)
```

この文では、ENAME 列が NULL になります。

別の方法として、OCI バインド・コールでインジケータ変数を使用する方法があります。

関連項目： 2-36 ページ「[インジケータ変数](#)」

NULL を挿入するもう 1 つの方法は、バッファ長パラメータと最大長パラメータの両方を、バインド・コールで 0（ゼロ）に設定する方法です。

注意： 対応するインジケータ変数が定義コールで指定されていても、そのインジケータ変数を含まない変数に NULL 選択リスト項目をフェッチすると、Oracle は SQL92 要件に従いエラーを戻します。

インジケータ変数

各バインドと定義 OCI コールには、インジケータ変数（配列を使用する場合はインジケータ変数の配列）を DML 文、PL/SQL 文または問合せに関連付けるパラメータがあります。

C 言語には NULL 値の概念がありません。したがって、インジケータ変数と入力変数を関連付けることによって、関連付けられたプレースホルダが NULL であるかどうかを指定します。データが Oracle に渡されるとき、これらのインジケータ変数の値によって、NULL がデータベース・フィールドに割り当てられているかどうか判断されます。

出力変数では、インジケータ変数によって、Oracle から戻された値が NULL または切り捨てられた値であるかどうか判断されます。NULL フェッチ (OCIStmtFetch()) または切捨て (OCIStmtExecute()) または OCIStmtFetch() の場合は、OCI コールにより OCI_SUCCESS が戻されます。対応するインジケータ変数は、表 2-8「出力インジケータ値」にリストされた該当する値に設定されます。アプリケーションに、対応する OCIDefineByPos() コールのリターン・コード変数がある場合、OCI はそのリターン・コード変数に ORA-01405 (NULL フェッチ用) または ORA-01406 (切捨て用) の値を割り当てます。

インジケータ変数のデータ型は **sb2** です。インジケータ変数の配列の場合、各配列要素の型は **sb2** です。

入力

入力ホスト変数については、OCI アプリケーションは次の値をインジケータ変数に割り当てることができます。

表 2-7 入力インジケータ値

入力インジケータ値	Oracle からのアクション
-1	Oracle は NULL を列に割り当て、入力変数の値は無視します。
>=0	入力変数の値を列に割り当てます。

出力

出力については、Oracle は次の値をインジケータ変数に割り当てます。

表 2-8 出力インジケータ値

出力インジケータ値	意味
-2	項目の長さが出力変数の長さよりも長い場合、項目を切り捨てます。さらに、元の長さが、sb2 インジケータ変数で戻せる最大長さより長くなっています。
-1	選択された値が NULL で、出力変数の値は変更されません。
0	完全な値をホスト変数に割り当てます。
>0	項目の長さが出力変数の長さよりも長い場合、項目を切り捨てます。インジケータ変数に戻された正の値は、切捨て前の実際の長さです。

名前付きデータ型用および REF 用のインジケータ変数

新しい（リリース 8.0 より後の）データ型のインジケータ変数は、前述の説明どおりに機能します。例外は `SQLT_NTY`（名前付きデータ型）です。`SQLT_REF` 型のデータでは、他の変数型と同様に、標準のスカラー・インジケータを使用します。`SQLT_NTY` 型のデータの場
合、インジケータ変数はインジケータ構造体へのポインタである必要があります。

Object Type Translator（OTT）を使用してデータベース型を C 構造体に変換すると、各オブジェクト型について NULL インジケータ構造体が生成されます。この構造体には、アトミック NULL インジケータと各オブジェクト属性のインジケータが含まれます。

関連項目：

- NULL インジケータ構造体の詳細は、このマニュアルの第 14 章「[Object Type Translator \(OTT\)](#)」にある OTT の説明および 10-30 ページの「[NULL かどうか](#)」を参照してください。
- 名前付きデータ型および REF のインジケータ・パラメータ設定の詳細は、15-64 ページの「[バインド関数、定義関数および記述関数](#)」の `OCIBindByName()` と `OCIBindByPos()` の説明、11-37 ページの「[名前付きデータ型および REF バインドの情報](#)」、および 11-39 ページの「[名前付きデータ型、REF 定義および PL/SQL OUT バインドの情報](#)」を参照してください。

コールの取消し

ほとんどのプラットフォームで、長時間実行されたり繰り返されている OCI コールを取り消すことができます。これを実行するには、キーボードからオペレーティング・システムの割込み文字（通常は `[Ctrl] + [C]` キー）を入力します。

注意： OCI コールの取消しをカーソルの取消しと混同しないように注意してください。カーソルの取消しは、`nrows` パラメータを 0（ゼロ）に設定し、`OCIStmtFetch()` をコールして行います。

長時間実行されたり繰り返されているコールをオペレーティング・システムへの割込みにより取り消した場合は、エラー・コード `ORA-01013` 「ユーザーによって現行の操作の取消しが要求されました。」が戻されます。

サービス・コンテキスト・ポインタまたはサーバー・コンテキスト・ポインタを指定すると、`OCIBreak()` 関数は、サーバーに関連付けられている現在実行中の OCI 関数をすべて即時（非同期）に終了します。これは通常、サーバーで長時間実行されている処理中の OCI コールを停止するために使用します。`OCIReset()` 関数は、OCI アプリケーションが `OCIBreak()` で関数を異常終了した後、非ブロック化接続に対しプロトコル同期化を実行するために必要です。

注意： サーバーが NT システムの場合、`OCIBreak()` はサポートされていません。

長時間実行する可能性のあるコールは、非ブロック化コールを使用することによりステータスを監視することができます。詳細は、2-41 ページの「[非ブロック化モード](#)」を参照してください。

位置指定の更新および削除

SELECT...FOR UPDATE OF... 文に関連付けられた ROWID を、後の UPDATE 文または DELETE 文に使用できます。ROWID は、文ハンドルに対して OCIAttrGet() をコールして取り出すことができ、これによりこのハンドルの OCI_ATTR_ROWID 属性を取り出せます。

たとえば、次のような SQL 文があるとします。

```
SELECT ename FROM emp WHERE empno = 7499 FOR UPDATE OF sal
```

この場合、フェッチが実行されると、ハンドルの ROWID 属性には選択された行の行識別子が入ります。次のコードのように OCIAttrGet() をコールして、プログラムのバッファに ROWID を取り出すことができます。

```
OCIRowid *rowid; /* the rowid in opaque format */
/* allocate descriptor with OCIDescriptorAlloc() */
err = OCIDescriptorAlloc ((dvoid *) envhp, (dvoid **) &rowid,
    (ub4) OCI_TYPE_ROWID, (size_t) 0, (dvoid **) 0));
err = OCIAttrGet ((dvoid*) mystmtmp, OCI_HTYPE_STMT,
    (dvoid*) rowid, (ub4 *) 0, OCI_ATTR_ROWID, (OCIError *) myerrhp);
```

その後、保存された ROWID を DELETE 文または UPDATE 文で使用できます。たとえば、rowid が行識別子の保存されているバッファである場合、後で次のような SQL 文を処理できます。

```
UPDATE emp SET sal = :1 WHERE rowid = :2
```

これは、新しい salary (給与) をプレースホルダ :1 にバインドし、rowid をプレースホルダ :2 にバインドして行います。rowid を :2 にバインドする場合は、必ずデータ型コード 104 (ROWID 記述子) を使用してください。

プリフェッチを使用して、ROWID の配列を後続のバッチ更新で使用するために選択できます。

関連項目： ROWID の詳細は、3-6 ページの「[UROWID](#)」および 3-14 ページの「[DATE](#)」を参照してください。

予約語

一部のワードは Oracle によって確保されています。つまり、それらのワードは Oracle にとって特別な意味を持っており、再定義できません。このため、それらのワードを使用して、列、表、索引などのデータベース・オブジェクトに名前を付けることはできません。

関連項目： SQL および PL/SQL 用の Oracle のキーワードまたは予約語のリストについては、『Oracle9i SQL リファレンス』および『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

Oracle 予約済みネームスペース

表 2-9「Oracle 予約済みネームスペース」は、Oracle によって予約されているネームスペースのリストです。Oracle ライブラリ内の関数名の最初の文字列は、このリストの文字列に制限されています。名前が競合する可能性があるため、これらの文字列で始まる関数名を付けないでください。たとえば、Oracle Net Transparent Network Service の関数はすべて、NS という文字で始まります。このため、NS で始まるネーミング関数名を付けないようにする必要があります。

表 2-9 Oracle 予約済みネームスペース

ネームスペース	ライブラリ
XA	XA アプリケーション専用の外部関数
SQ	Oracle のプリコンパイラおよび SQL*Module アプリケーションで使用される外部 SQLLIB 関数
O、OCI	外部 OCI 関数、内部 OCI 関数
UPI、KP	Oracle UPI レイヤーの関数名
NA	Oracle Net ネイティブ・サービス・プロダクト
NC	Oracle Net Rpc プロジェクト
ND	Oracle Net ディレクトリ
NL	Oracle Net ネットワーク・ライブラリ・レイヤー
NM	Oracle Net 管理プロジェクト
NR	Oracle Net インターチェンジ
NS	Oracle Net トランスペアレント・ネットワーク・サービス
NT	Oracle Net ドライバ
NZ	Oracle Net セキュリティ・サービス
OS	SQL*Net バージョン 1
TTC	Oracle Net 2 タスク
GEN、L、ORA	コア・ライブラリ関数
LI、LM、LX	Oracle グローバリゼーション・サポート・レイヤーの関数名
S	システム依存ライブラリの関数名
KO	カーネル・オブジェクト

表 2-9「Oracle 予約済みネームスペース」のリストには、Oracle 予約済みネームスペースのすべての関数は含まれていません。特定のネームスペースのすべての関数リストは、該当する Oracle ライブラリに対応したドキュメントを参照してください。

関数名

OCI プログラム内でユーザー関数を作成する際は、OCI 関数と競合する可能性があるので、OCI で始まる関数名を付けないでください。

非ブロック化モード

OCI では、ブロック化モードまたは非ブロック化モードでサーバー接続を確立できます。接続がブロック化モードで行われた場合、結果が正常かエラーかにかかわらずコールが完了した場合にのみ、OCI コールはコントロールを OCI クライアント・アプリケーションに戻します。非ブロック化モードでは、コールが完了せずコールから OCI_STILL_EXECUTING の値が戻されても、コントロールは即時 OCI プログラムに戻されます。

非ブロック化モードでは、アプリケーションで各 OCI 関数のリターン・コードをテストし、OCI_STILL_EXECUTING が戻されるかどうかを確認する必要があります。この場合、OCI クライアントはこの OCI コールのサーバーでの再試行を待つ間、プログラム・ロジックを継続できます。

非ブロック化モードでは、いったんコールを行うとコントロールが OCI プログラムに戻されるため、サーバーにより OCI コールが処理されている間、他の操作を行うことができます。このモードは、Graphical User Interface (GUI) アプリケーション、リアルタイム・アプリケーションおよび分散環境の場合に特に有用です。

非ブロック化モードは非割込み駆動で、むしろポーリング・パラダイムに基づいています。これは、保留状態のコールがサーバーで終了したかどうかを、クライアント・アプリケーションがチェックする必要があることを意味します。クライアント・アプリケーションは、同一パラメータでそのコールを再び実行することにより、保留状態のコールがサーバーで終了したかどうかをチェックする必要があります。

注意： 非ブロック化 OCI コールの再試行の待機中に、アプリケーションで他の OCI コールを発行することはできません。他の OCI コールを発行すると、ORA-03124 エラーが発生します。この規則の唯一の例外は、OCIBreak() および OCIReset() です。

これらのコールの詳細は、2-42 ページの「非ブロック化コールの取消し」を参照してください。

ブロック化モードの設定

アプリケーションのブロック化状態は、`attrtype` パラメータを `OCI_ATTR_NONBLOCKING_MODE` に設定して、サーバー・コンテキスト・ハンドルで `OCIAttrSet()` をコールして状態を設定したり、`OCIAttrGet()` をコールして状態を読み込むことにより、変更またはチェックすることができます。

関連項目： A-15 ページの「[OCI_ATTR_NONBLOCKING_MODE](#)」を参照してください。

注意： パラメータとしてサーバー・コンテキスト・ハンドルまたはサーバー・コンテキスト・ハンドルを持つ関数のみが、`OCI_STILL_EXECUTING` を戻します。

非ブロック化コールの取消し

`OCIBreak()` 関数を使用することにより、長時間実行している OCI コールを取り消すことができます。OCI コールの実行中に `OCIBreak()` を発行した後で、`OCIReset()` コールを発行して非同期操作およびプロトコルをリセットする必要があります。

非ブロック化の例

次のコードは、非ブロック化モードの例です。

```
int main (int argc, char **argv)
{
    sword retval;

    if (retval = InitOCIHandles()) /* initialize all handles */
    {
        printf ("Unable to allocate handles...\n");
        exit (EXIT_FAILURE);
    }

    if (retval = logon()) /* log on */
    {
        printf ("Unable to log on...\n");
        exit (EXIT_FAILURE);
    }
    if (retval = AllocStmtHandle ()) /* allocate statement handle */
    {
        printf ("Unable to allocate statement handle...\n");
        exit (EXIT_FAILURE);
    }
    /* set nonblocking on */
```



```
if (retval = OCIAttrSet ((dvoid *) srvhp, (ub4) OCI_HTYPE_SERVER,
                        (dvoid *) 0, (ub4) 0,
                        (ub4) OCI_ATTR_NONBLOCKING_MODE, errhp))
{
    printf ("Unable to set nonblocking mode...\n");
    exit (EXIT_FAILURE);
}

while ((retval = OCISstmtExecute (svchp, stmhp, errhp, (ub4)0, (ub4)0,
                                (OCISnapshot *) 0, (OCISnapshot *) 0,
                                OCI_DEFAULT)) == OCI_STILL_EXECUTING)
    printf (".");
printf ("\n");

if (retval != OCI_SUCCESS || retval != OCI_SUCCESS_WITH_INFO)
{
    printf("Error in OCISstmtExecute...\n");
    exit (EXIT_FAILURE);
}

if (retval = logoff ()) /* log out */
{
    printf ("Unable to logout ...\n");
    exit (EXIT_FAILURE);
}

cleanup();
return (int)OCI_SUCCESS;
}
...
```

OCI プログラムでの PL/SQL 使用

PL/SQL は、Oracle が開発した SQL 言語の手続き型拡張機能です。PL/SQL は、単純な問合せや SQL データ操作言語（DML）文よりも複雑なタスクを処理します。PL/SQL を使用すると、複数の構文を単一のブロックにグループ化し、それを 1 単位として実行できます。次のような構文があります。

- 1 つまたは複数の SQL 文
- 変数宣言
- 代入文
- IF...THEN...ELSE 文やループなどのプロシージャ型制御文
- 例外処理

OCI プログラムで PL/SQL ブロックを使用して、次の操作ができます。

- Oracle ストアド・プロシージャおよびストアド・ファンクションのコール
- 複数の SQL 文を指定したプロシージャ型制御文を結合し、1 つの単位として実行
- レコード、表、CURSOR FOR ループ、例外処理など特殊な PL/SQL 機能へのアクセス
- カーソル変数の使用
- Oracle8 Server 内のオブジェクトの操作

注意： OCI で直接処理できるのは無名ブロックのみであり、名前付きパッケージまたは名前付きプロシージャは処理できないのに対し、プログラマは常にパッケージ・コールまたはプロシージャ・コールを無名ブロック内に入れ、そのブロックを処理できます。

OCI で PL/SQL の開始 / 終了ブロックを実行する前に、インジケータを -1 に設定するかまたは実際の長さを 0（ゼロ）に設定して、すべての OUT 変数を NULL に初期化する必要があることに注意してください。

OCI では、PL/SQL RECORD データ型はサポートされていません。

OCI では、PL/SQL VARCHAR2 変数をバインドする場合のバインド変数の最大サイズは、制御構造のオーバーヘッドの理由から、32512 です。

注意： PL/SQL コードを記述する場合は、パーサーにより、「--」で始まるものは改行まですべてコメントとみなされることに注意してください。「--」を使用してコメントを各行に示した場合、C コンパイラでは各行に改行「\n」が挿入されず、すべての行が PL/SQL ブロックで単一行として連結されます。この特定のケースでは、ある行がコメントで終わっている場合、パーサーは次の行の PL/SQL コード抽出に失敗してしまいます。この問題を回避するには、各「--」コメントの後に「\n」を入れ、そこでコメントが必ず終了するように注意してください。

関連項目： PL/SQL ブロックのコーディングの詳細は、『PL/SQL ユーザーズ・ガイドおよびリファレンス』を参照してください。

OCI グローバリゼーション・サポート

OCI では、文字列ベースのファンクション・コール（SQL 文、データ、ユーザーやパスワードなどのメタデータ、オブジェクト・サポート、エラー・メッセージなどでのファンクション・コール）で、UTF-16 Unicode エンコーディング形式をサポートしています。

UTF-16 は、2 バイトの UCS2 で作成される可変幅の Unicode エンコーディングです。任意の言語の文字を UTF-16 で表すことができます。AL16UTF16 は、UTF-16 の Oracle キャラクタ・セット名です。UTF-16 は、UCS2 のスーパーセットです。

ASCII およびシステム固有のキャラクタ・セットも同様にサポートされています。ASCII の文字列は、バイト長セマンティクスと呼ばれる方法に準拠しています。Unicode 文字列は、文字長セマンティクスに準拠しています。

OCI は、ユーザーとサーバー間のインタフェースです。したがって、記述、挿入、更新およびフェッチの各操作では、コードポイント長セマンティクスを認識できます。

関連項目：

- OCI コールでの Unicode サポートの詳細は、5-34 ページの「[バインドおよび定義における文字変換の問題](#)」を参照してください。
- 6-21 ページ「[記述での文字長セマンティクスのサポート](#)」
- OCI コールでの Unicode サポートの詳細は、『Oracle9i Database グローバリゼーション・サポート・ガイド』を参照してください。

UTF-16 環境

UTF-16 を OCI で使用するかどうか、および UTF-16 の OCI での使用方法は、モード・パラメータによる OCI 環境ハンドルの設定およびハンドルの継承方法によって決まります。環境ハンドル以外のすべてのハンドルは、UTF-16 設定に対してステートレスです。つまり、UTF-16 情報は保存されません。したがって、ユーザーは、常に親環境ハンドルから UTF-16 設定を取得する必要があります。

関連項目： UTF-16 環境でプログラミングを行う場合は、`OCIEnvCreate()` よりも 15-14 ページの「[OCIEnvNlsCreate\(\)](#)」を使用することをお勧めします。この関数の使用方法については、2-48 ページの「[OCI からのクライアント・キャラクタ・セットの制御](#)」を参照してください。

`OCIEnvCreate()` を除くすべての関数は、UTF-16 バッファを指す **text *** パラメータを使用します。ただし、OCI を簡素化するため、一部の OCI 関数には、長さのパラメータの前に、**dvoid *** として渡される文字列の基本形のパラメータが必要です。

バイト制限と同様に、コードポイント長制限についてバッファ制約をチェックできます。たとえば、ハンドル属性を取り出す `OCIAttrGet()` には、**dvoid *** 型にキャストするために戻される属性名が必要です。また、既存のスキーマ・オブジェクトを記述する `OCIDescribeAny()` には、**dvoid *** パラメータとして渡されるオブジェクト名が必要です。**dvoid *** パラメータは、名前などの文字列バッファから取得できます。

text * パラメータの場合、対応する文字列の長さは、その値を渡すか取得するかにかかわらず、そして UTF-16 設定に関係なく、常に文字列全体のバイト数になります。つまり、文字列の長さは、常に UTF-16 の文字数の 2 倍になります。

この規則の唯一の例外は、長さのパラメータが文字数を参照する既存の UTF-16 がサポートされた一連の関数です。

トップレベルの環境作成

`OCIEnvCreate()` は、他の OCI ハンドルと構造によってその基本情報が取得される環境ハンドルを初期化します。ユーザーは、このファンクション・コールを使用して、環境全体に対する UTF-16 エンコーディングを設定できます。

テキスト入力を持つリレーショナル OCI 関数

次の関数はサーバーと通信を行いますが、モード・パラメータは不要であるため取得しません。これらの関数では、環境ハンドルまたは文ハンドルに埋め込まれて関数に渡されるモード設定が選択されます。

- `OCIStmtPrepare()`
- `OCIBindByName()`
- `OCIServerAttach()`
- `OCIPasswordChange()`

- OCILogon()
- OCILobFileSetName()
- OCIAttrSet()
- OCIDescribeAny()

アプリケーションのプログラミングで UTF-16 バッファと非 UTF-16 バッファの混在が避けられない場合は、それぞれのエンコーディングを処理するために 2 つの環境ハンドルの作成をお勧めします。これは、ほとんどのリレーショナル OCI コールおよびオブジェクト OCI コールで、一度に 1 つの環境ハンドルしか取得できないためです。テキスト・パラメータ以外のパラメータを取得する関数の場合、文字列バッファは常に同じエンコーディングで設定され処理される必要があります。1 つのファンクション・コールに異なる文字列パラメータが混在するエンコーディングは無効になります。

テキスト出力を持つリレーショナル OCI 関数

テキスト入力を持つ関数についての前述の説明は、次の関数にも適用されます。

- OCISstmtGetBindInfo()
- OCIErrorGet()
- OCIServerVersion()
- OCILobFileName()
- OCIAttrGet()

UTF-16 データを持つ OCI 文字列関数

スカラー型の **OCISString** では、バッファのエンコーディングは、そのバッファが属する環境ハンドルによって異なります。SQLT_VST データ型として処理する場合、**OCISString** は、バインド・ハンドルおよび定義ハンドルに対して通常の文字列型と同様に動作します。

OCI_UTF16 モードで環境ハンドルが作成された場合、**OCISString** のデータは UTF-16 エンコーディングであることが必要です。それ以外の場合、データは NLS_LANG エンコーディングになります。対応するバインド・ハンドルまたは定義ハンドルのデフォルト・キャラクタ・セットは OCI_UTF16ID で、これは UTF-16 を意味します。サイズ・パラメータは、常にバイト単位になります。関連する関数は次のとおりです。

- OCISStringAssignText() – UTF-16 テキストを **OCISString** に割り当てることができます。
- OCISStringResize() – **OCISString** をバイト単位でサイズ変更します。
- OCISStringAllocSize() – **OCISString** をバイト単位で割り当てます。

- `OCIStringPtr()` – テキスト・ポインタを、UTF-16 エンコーディングが可能な **OCIString** に戻します。
- `OCIStringSize()` – サイズをバイト単位で戻します。

文字長セマンティクス

OCI は、サーバーとクライアント間のトランスレータとして機能し、制約チェックを行うための文字情報を渡します。

キャラクタ・セットには可変幅と固定幅の 2 種類があります。シングルバイト・キャラクタ・セットは、各バイトが 1 つの文字を表す固定幅キャラクタ・セットです。

固定幅キャラクタ・セットの場合、バイト数は文字数の倍数であるため、制約チェックが簡単です。したがって、固定幅キャラクタ・セットでは、文字数を調べるために文字列全体をスキャンする必要がありません。これに対して、可変幅キャラクタ・セットでは、文字列全体をスキャンして文字数を調べる必要があります。

キャラクタ・セットのサポート

詳細な説明は、6-21 ページの「[記述での文字長セマンティクスのサポート](#)」および 5-34 ページの「[バインドおよび定義における文字変換の問題](#)」を参照してください。

OCI からのクライアント・キャラクタ・セットの制御

Oracle9i リリース 2 (9.2) で導入された `OCIEnvNlsCreate()` 関数を使用して、`NLS_LANG` および `NLS_NCHAR` の設定値とは別に、キャラクタ・セット情報を実行中のアプリケーションに設定できます。つまり、様々なクライアント側キャラクタ・セット ID および各国語キャラクタ・セット ID を使用して、同じシステム環境内で初期化した複数の環境ハンドルを 1 つのアプリケーションで使用できます。

```
OCIEnvNlsCreate(OCIEnv **envhp, ..., csid, ncsid);
```

`csid` はキャラクタ・セット ID の値で、`ncsid` は各国語キャラクタ・セット ID の値です。いずれの値も 0 (ゼロ) または `OCI_UTF16ID` に設定できます。両方とも 0 (ゼロ) の場合は、`OCIEnvCreate()` を使用するのと同じになります。その他の引数は、`OCIEnvCreate()` コールの引数と同じです。

`OCIEnvNlsCreate()` の導入によって `OCIEnvCreate()` が使用できなくなるわけではありません。これは、`OCIEnvNlsCreate()` で使用する 2 つの追加パラメータが、一部のアプリケーションには不要なためです。`OCIEnvNlsCreate()` は、キャラクタ・セットをプログラムによって制御するための拡張機能です。`OCIEnvNlsCreate()` では、`OCI_UTF16ID` を検証するため、`OCI_UTF16` モードは使用できません。

`OCIEnvNlsCreate()` 関数を使用してキャラクタ・セット ID を設定すると、`NLS_LANG` および `NLS_NCHAR` の設定値が置換されます。`OCIEnvNlsCreate()` 関数では、`NLSRTL` でサポートされるすべてのキャラクタ・セット以外に、`OCI_UTF16ID` をキャラクタ・セッ

ト ID として設定できます (NLS_LANG または NLS_NCHAR では無効となります)。
 NLS_LANG および NLS_NCHAR のキャラクタ・セットは、OCI 関数の
 OCINlsEnvironmentVariableGet () を使用して取得できます。

関連項目： 15-14 ページ「[OCIEnvNlsCreate\(\)](#)」

OCI でキャラクタ・セットを制御するコードの例

次のコード・フラグメントは、前述のコールの使用例を示します。

```
...
OCIEnv *envhp;
ub2 ncsid = 2; /* we8dec */
ub2 hdlcsid, hdlncsid;
raText thename[20];
utext *selstmt = L"SELECT ENAME FROM EMP"; /* UTF16 statement */
OCIStmt *stmthp;
OCIDefine *defhp;
OCIError *errhp;

OCIEnvNlsCreate(OCIEnv **envhp, ..., OCI_UTF16ID, ncsid);
...
OCIStmtPrepare(stmthp, ..., selstmt, ...); /* prepare UTF16 statement */

OCIDefineByPos(stmthp, defnp, ..., 1, thename, sizeof(thename), SQLT_CHR,...);

OCINlsEnvironmentVariableGet(&hdlcsid, (size_t)0, OCI-NLS_CHARSET_ID, (ub2)0,
    (size_t*)NULL);

OCIAttrSet(defnp, ..., &hdlcsid, 0, OCI_ATTR_CHARSET_ID, errhp);
    /* change charset id to NLS_LANG setting*/
...
```

文字制御と OCI インタフェース

`OCIEnvNlsGetInfo()` は、`OCI_UTF16ID` が `OCIEnvNlsCreate()` で使用されている場合に、`OCI_UTF16ID` に関する情報を戻します。

`OCIAttrGet()` は、`OCIEnvNlsCreate()` に渡されたキャラクタ・セット ID および各国語キャラクタ・セット ID を戻します。これは、`OCI_ATTR_ENV_CHARSET_ID` および `OCI_ATTR_ENV_NCHARSET_ID` を取得するために使用します。これには、`OCI_UTF16ID` 値も含まれます。

`OCIEnvNlsCreate()` を使用して *charset* および *ncharset* パラメータに `NULL` が設定されている場合は、`NLS_LANG` および `NLS_NCHAR` のキャラクタ・セット ID が戻されます。

この関数を使用して `OCI_ATTR_CHARSET_FORM` がリセットされている場合、`OCIAttrSet()` は文字 ID をデフォルトとして設定します。`OCI_UTF16ID` は、`OCIEnvNlsCreate()` で *charset* または *ncharset* として渡される場合、有効なキャラクタ・セット ID の 1 つになります。

`OCIBindByName()` および `OCIBindByPos()` は、`OCIEnvNlsCreate()` コールのデフォルトのキャラクタ・セット (`OCI_UTF16ID` を含む) を使用して、変数をバインドします。`OCIEnvNlsCreate()` を使用する場合、実際の長さとは異なる長さは常にバイト単位で表されます。

`OCIDefineByPos()` は、`OCIEnvNlsCreate()` の *charset* の値 (`OCI_UTF16ID` を含む) を持つ変数をデフォルトとして定義します。`OCIEnvNlsCreate()` を使用する場合、実際の長さとは異なる長さは常にバイト単位で表されます。バインド・ハンドルおよび定義ハンドルのこの動作は、`OCIEnvCreate()` を使用する場合とは異なり、`OCI_UTF16ID` はバインド・ハンドルおよび定義ハンドル用のキャラクタ・セット ID であることに注意してください。

OCI データベースのグローバリゼーション・サポート関数

次の表は、データベース・グローバリゼーションをサポートする OCI 関数を示します。これらの関数の詳細および説明は、次を参照してください。

関連項目：

- 『Oracle9i Database グローバリゼーション・サポート・ガイド』で、次の関数のグローバリゼーション・ガイドの説明を参照してください。
- `OCINlsGetInfo()`
- `OCINlsCharSetNameToId()`
- `OCINlsCharsetIdToName()`
- `OCINlsNumericInfoGet()`
- `OCINlsNameMap()`
- `OCINlsCharSetConvert()`
- キャラクタ・セット ID の取得方法については、16-176 ページの「[OCINlsEnvironmentVariableGet\(\)](#)」を参照してください。

OCI 文字列操作関数

表 2-10 OCI 文字列操作関数

関数	説明
<code>OCIMultiByteToWideChar()</code>	ヌル文字で終了する文字列全体を <code>wchar</code> 形式に変換します。
<code>OCIMultiByteInSizeToWideChar()</code>	文字列の一部を <code>wchar</code> 形式に変換します。
<code>OCIWideCharToMultiByte()</code>	ヌル文字で終了するワイド・キャラクタ文字列をマルチバイト文字列に変換します。
<code>OCIWideCharInSizeToMultiByte()</code>	ワイド・キャラクタ文字列の一部をマルチバイト形式に変換します。
<code>OCIWideCharToLower()</code>	指定したロケールに大文字のマッピングがある場合は、ワイド・キャラクタ内の小文字を戻します。それ以外の場合は、ワイド・キャラクタをそのまま戻します。
<code>OCIWideCharToUpper()</code>	指定したロケールに小文字のマッピングがある場合は、ワイド・キャラクタ内の大文字を戻します。それ以外の場合は、ワイド・キャラクタをそのまま戻します。
<code>OCIWideCharStrcmp()</code>	2 つのワイド・キャラクタ文字列を、バイナリ、言語または大 / 小文字を区別しない比較メソッドを使用して比較します。

表 2-10 OCI 文字列操作関数（続き）

関数	説明
<code>OCIWideCharStrncmp()</code>	<code>OCIWideCharStrcmp()</code> と同じですが、2 つのマルチバイト文字列を、バイナリ、言語または大 / 小文字を区別しない比較メソッドを使用して比較します。 <code>str1</code> は最大 <code>len1</code> バイトで形成され、 <code>str2</code> は最大 <code>len2</code> バイトで形成されます。
<code>OCIWideCharStrcat()</code>	<code>wsrcstr</code> が指し示す文字列のコピーを追加します。次に、結果文字列の文字数を戻します。
<code>OCIWideCharStrchr()</code>	<code>wstr</code> が指し示す文字列内にある最初の <code>wc</code> を検索します。検索が正常終了すると、 <code>wchar</code> へのポインタを戻します。
<code>OCIWideCharStrncpy()</code>	<code>wsrcstr</code> が指し示す <code>wchar</code> 文字列を、 <code>wdststr</code> が指し示す配列にコピーします。次に、コピーされた文字数を戻します。
<code>OCIWideCharStrlen()</code>	<code>wstr</code> が指し示す <code>wchar</code> 文字列の文字数を計算し、その文字数を戻します。
<code>OCIWideCharStrncat()</code>	<code>wsrcstr</code> が指し示す文字列のコピーを追加します。ただし、追加されるのは最大 <code>n</code> 文字です。次に、結果文字列の文字数を戻します。
<code>OCIWideCharStrncpy()</code>	<code>wsrcstr</code> が指し示す <code>wchar</code> 文字列を、 <code>wdststr</code> が指し示す配列にコピーします。ただし、配列にコピーされるのは最大 <code>n</code> 文字です。次に、コピーされた文字数を戻します。
<code>OCIWideCharStrrchr()</code>	<code>wstr</code> が指し示す文字列内にある最後の <code>wc</code> を検索します。
<code>OCIWideCharStrCaseConversion()</code>	<code>wsrcstr</code> が指し示すワイド・キャラクタ文字列を大文字または小文字（フラグによって指定）に変換し、その結果を、 <code>wdststr</code> が指し示す配列にコピーします。
<code>OCIWideCharDisplayLength()</code>	表示するときに <code>wc</code> で必要な列位置の番号を判断します。
<code>OCIWideCharMultibyteLength()</code>	マルチバイト・エンコーディングするときに <code>wc</code> で必要なバイト数を判断します。
<code>OCIMultiByteStrcmp()</code>	2 つのマルチバイト文字列を、バイナリ、言語または大 / 小文字を区別しない比較メソッドを使用して比較します。
<code>OCIMultiByteStrncmp()</code>	2 つのマルチバイト文字列を、バイナリ、言語または大 / 小文字を区別しない比較メソッドを使用して比較します。 <code>str1</code> は最大 <code>len1</code> バイトで形成され、 <code>str2</code> は最大 <code>len2</code> バイトで形成されます。
<code>OCIMultiByteStrcat()</code>	<code>srcstr</code> が指し示すマルチバイト文字列のコピーを追加します。
<code>OCIMultiByteStrncpy()</code>	<code>srcstr</code> が指し示すマルチバイト文字列を、 <code>dststr</code> が指し示す配列にコピーします。次に、コピーされたバイト数を戻します。
<code>OCIMultiByteStrlen()</code>	<code>str</code> が指し示すマルチバイト文字列のバイト数を計算し、そのバイト数を戻します。

表 2-10 OCI 文字列操作関数（続き）

関数	説明
OCIMultiByteStrncat()	srcstr が指し示すマルチバイト文字列のコピーを追加します。ただし、srcstr から dststr に追加されるのは最大 <i>n</i> バイトです。
OCIMultiByteStrncpy()	srcstr が指し示すマルチバイト文字列を、dststr が指し示す配列にコピーします。ただし、srcstr が指し示す配列から dststr が指し示す配列にコピーされるのは、最大 <i>n</i> バイトです。次に、コピーされたバイト数を戻します。
OCIMultiByteStrnDisplayLength()	文字列全体が占める表示位置のバイト数を、 <i>n</i> バイト以内で戻します。
OCIMultiByteStrCaseConversion()	文字列の一部を別のキャラクタ・セットに変換します。

OCI 文字分類関数

表 2-11 OCI 文字分類関数

関数	説明
OCIWideCharIsAlnum()	ワイド・キャラクタが文字か 10 進数値かをテストします。
OCIWideCharIsAlpha()	ワイド・キャラクタがアルファベット文字かどうかをテストします。
OCIWideCharIsCntrl()	ワイド・キャラクタが制御文字かどうかをテストします。
OCIWideCharIsDigit()	ワイド・キャラクタが 10 進数値かどうかをテストします。
OCIWideCharIsGraph()	ワイド・キャラクタがグラフ文字かどうかをテストします。
OCIWideCharIsLower()	ワイド・キャラクタが小文字かどうかをテストします。
OCIWideCharIsPrint()	ワイド・キャラクタが印字可能文字かどうかをテストします。
OCIWideCharIsPunct()	ワイド・キャラクタが句読点文字かどうかをテストします。
OCIWideCharIsSpace()	ワイド・キャラクタがスペース文字かどうかをテストします。
OCIWideCharIsUpper()	ワイド・キャラクタが大文字かどうかをテストします。
OCIWideCharIsXdigit()	ワイド・キャラクタが 16 進数値かどうかをテストします。
OCIWideCharIsSingleByte()	マルチバイトへの変換時に、wc がシングルバイト文字かどうかをテストします。

OCI 文字変換関数

表 2-12 OCI キャラクタ・セット変換関数

関数	説明
OCICharsetToUnicode()	src が指し示すマルチバイト文字列を Unicode に変換し、dst が指し示す配列に追加します。
OCIUnicodeToCharset()	src が指し示す Unicode 文字列をマルチバイトに変換し、dst が指し示す配列に追加します。
OCICharSetConversionIsReplacementUsed()	最後に起動された OCICharsetConv() でのキャラクタ・セット変換で、変換不可の文字に対して置換文字が使用されたかどうかを示します。

OCI メッセージ関数

表 2-13 OCI メッセージ関数

関数	説明
OCIMessageOpen()	hndl が指し示す、言語内のメッセージ・ハンドルをオープンします。
OCIMessageGet()	msgno で識別されたメッセージ番号を持つメッセージを取得します。バッファが 0 (ゼロ) でない場合、この関数は、msgbuf が指し示すバッファにメッセージをコピーします。
OCIMessageClose()	msggh が指し示すメッセージ・ハンドルをクローズし、このハンドルに関連するすべてのメモリーを解放します。

この章は、OCI アプリケーションで使用する Oracle 外部データ型についてのリファレンスです。最新リリースから新しく加わった特殊なデータ型を含め、Oracle データ型についても説明しています。この章の情報は、独自に作成したプログラムと Oracle の間でデータを転送する場合に行われる、データの内部表現と外部表現の変換を理解するうえで役に立ちます。この章は、次の項目で構成されています。

- [Oracle データ型](#)
- [内部データ型](#)
- [外部データ型](#)
- [新しい外部データ型](#)
- [データ変換](#)
- [型コード](#)
- [oratypes.h の定義](#)

関連項目： Oracle 内部データ型の詳細は、『Oracle9i SQL リファレンス』を参照してください。

Oracle データ型

OCI プログラムの重要な機能の 1 つは、Oracle サーバーを介してデータベースと通信を行うことです。OCI アプリケーションでは、SQL SELECT を使用した問合せによってデータベースの表からデータを取り出すか、INSERT、UPDATE または DELETE 文を使用して表内の既存データを変更します。

データベース内では、値は表の列に格納されています。内部的には、Oracle は内部データ型という特別な形式でデータを表現します。内部データ型の例として、NUMBER、CHAR、DATE などがあります。

一般的に、OCI アプリケーションではデータの内部データ型表現を処理しません。OCI アプリケーションで操作するホスト言語のデータ型は、そのアプリケーションを作成した言語によって事前定義されています。OCI クライアント・アプリケーションとデータベースの表の間でデータが転送されるとき、OCI ライブラリによって、内部データ型と外部データ型の間でデータ変換が行われます。

外部データ型は、OCI ヘッダー・ファイル内に定義されているホスト言語型です。OCI アプリケーションで入力変数をバインドするとき、バインド・パラメータの 1 つで変数の外部データ型コード（または SQLT コード）を指示します。同様に、定義コールで出力変数を指定するとき、取り出されるデータの外部表現を指定する必要があります。

外部データ型が内部データ型と同じ場合もあります。外部型を使用すると、専用データ形式ではなくホスト言語の型を操作できるので、プログラマにとって便利です。

注意： 一部の外部データ型は内部データ型に類似していますが、OCI アプリケーションが内部データ型にバインドされることはありません。ここでは、内部データ型がどのように外部データ型にマップされるかを理解するために説明します。

OCI では、Oracle と OCI アプリケーション間でデータを転送する際、広範囲のデータ型変換を行うことができます。Oracle 内部データ型より OCI 外部データ型のほうが多くあります。単一の外部型を 1 つの内部型にマップする場合もあり、複数の外部型を単一の内部型にマップする場合もあります。

あるデータ型が多対 1 の関係でマッピングされることによって、OCI プログラムに柔軟性が提供されます。たとえば、次の SQL 文を処理しているとします。

```
SELECT sal FROM emp WHERE empno = :employee_number
```

このとき、sal（給与）を 2 進数浮動小数点形式ではなく文字データ型として戻すには、sal 列に対する OCIDefineByPos() コールで dty パラメータに、VARCHAR2（コード = 1）や CHAR（コード = 96）などの Oracle 外部文字列データ型を指定します。また、プログラムで文字列変数を宣言し、そのアドレスを valuep パラメータで指定する必要があります。

ただし、給与情報を 2 進数浮動小数点値として戻す場合は、FLOAT (コード = 4) 外部データ型を指定します。また、`valuep` パラメータに適切な型の変数を定義する必要があります。

Oracle では、ほとんどのデータ変換が透過的に実行されます。また、ほとんどすべての外部データ型を指定できるため、特殊なタスクを実行できます。たとえば、DATE 外部データ型 (コード = 12) を使用すると、文字変換を行わずに純粋なバイナリ形式で DATE 値を入出力できます。詳細は、3-14 ページの「[DATE](#)」の外部データ型の説明を参照してください。

データ変換を制御するには、適切な外部データ型コードをバインド・ルーチンおよび定義ルーチンで使用する必要があります。OCI プログラム内の入力変数または出力変数の位置、および変数のデータ型と長さを Oracle に対して指定する必要があります。

また、OCI では、オブジェクト型属性のデータ型を表すために Oracle の型管理システムで使用する一連の OCI 型コードも用意されています。これらの型コードを表すために使用する一連の事前定義済みの定数があります。各定数には接頭辞 `OCI_TYPECODE` が含まれます。

要約すると、OCI プログラマは、次の各種のデータ型またはデータ表現について認識する必要があります。

- 内部 Oracle データ型。Oracle データベースの表の列で使用されます。これは、PL/SQL で使用され、Oracle 列では使用されないデータ型も含まれます (たとえば、索引付き表、ブール、レコード)。

関連項目： 3-4 ページの「[内部データ型](#)」

- 外部 OCI データ型。Oracle データのホスト言語表現を指定するために使用されます。

関連項目： 3-7 ページの「[外部データ型](#)」および 3-4 ページの「[外部データ型コードの使用](#)」

- `OCI_TYPECODE` 値。Oracle でオブジェクト型属性の型情報を表現するために使用されます。

関連項目： 3-29 ページの「[型コード](#)」および 3-31 ページの「[SQLT 値および OCI_TYPECODE 値の関係](#)」

列の内部データ型についての情報は、内部データ型コードの形式でアプリケーションに送られます。アプリケーション側では、どの型のデータが戻されるのかわかると、出力データの変換方法と書式設定の方法について適切な判断ができます。Oracle 内部データ型コードのリストは、3-4 ページの「[内部データ型](#)」を参照してください。

関連項目： Oracle 内部データ型の詳細は、『Oracle9i SQL リファレンス』を参照してください。問合せでの選択リスト項目記述の詳細は、4-12 ページの「[選択リスト項目の記述](#)」を参照してください。

外部データ型コードの使用

Oracle では、外部データ型コードによって、プログラムでのホスト変数のデータ表現方法が指定されます。これによって、プログラムの出力変数にデータが戻されるときの変換方法、または入力（バインド）変数から Oracle の列値へのデータ変換方法が決まります。たとえば、Oracle 列の NUMBER を可変長文字配列に変換する場合は、出力変数を定義する OCIDefineByPos () コールで VARCHAR2 外部データ型コードを指定します。

バインド変数を Oracle 列の値に変換するには、バインド変数の型に対応する外部データ型コードを指定します。たとえば、02-FEB-65 などの文字列を DATE 列に入力する場合は、データ型を文字列として指定し、文字列長パラメータを 9 に設定します。

使用する値を変換可能にするのは、常にプログラマの責任です。文字列 MY BIRTHDAY を DATE 列に挿入する文を実行すると、エラーが発生します。

関連項目： 外部データ型およびデータ型コードの完全なリストは、[表 3-2「外部データ型とそのコード」](#)を参照してください。

内部データ型

次の表は、Oracle 内部（組込み）データ型を各型の最大内部長およびデータ型コードとともに示しています。

表 3-1 Oracle 内部データ型

Oracle 内部データ型	最大内部長	データ型コード
VARCHAR2、NVARCHAR2	4000 バイト	1
NUMBER	21 バイト	2
LONG	2^31-1 バイト (2GB)	8
ROWID	10 バイト	11
DATE	7 バイト	12
RAW	2000 バイト	23
LONG RAW	2^31-1 バイト	24
CHAR、NCHAR	2000 バイト	96
ユーザー定義型（オブジェクト型、VARRAY、NESTED TABLE）	該当なし	108
REF	該当なし	111
CLOB、NCLOB	4GB	112
BLOB	4GB	113

表 3-1 Oracle 内部データ型（続き）

Oracle 内部データ型	最大内部長	データ型コード
BFILE	4GB	114
TIMESTAMP	11 バイト	180
TIMESTAMP WITH TIME ZONE	13 バイト	181
INTERVAL YEAR TO MONTH	5 バイト	182
INTERVAL DAY TO SECOND	11 バイト	183
UROWID	3950 バイト	208
TIMESTAMP WITH LOCAL TIME ZONE	11 バイト	231

関連項目： これらの組み込みデータ型の詳細は、『Oracle9i SQL リファレンス』を参照してください。次の各項では、これらのデータ型に関する OCI 固有の情報について詳しく説明します。

LONG、RAW、LONG RAW、VARCHAR2

OCIBindByName()、OCIBindByPos()、OCIDefineByPos()、OCIStmtGetPieceInfo() および OCIStmtSetPieceInfo() によって提供されるピース単位機能を使用すると、これらの型の列データの挿入、更新またはフェッチを実行できます。

文字列およびバイト配列

文字またはバイト配列を含む列を指定する場合は、CHAR、VARCHAR2、RAW、LONG および LONG RAW の 5 つの Oracle 内部データ型を使用できます。

注意： LOB には文字を、FILE にはバイナリ・データを含めることができます。これらは、他の型とは異なる処理がされるので、ここでは説明しません。これらのデータ型の詳細は、第 7 章「LOB と FILE の操作」を参照してください。

通常、CHAR 列、VARCHAR2 列および LONG 列には文字データが入ります。RAW および LONG RAW には、文字として解釈されないバイト、たとえばビットマップ化された画像のピクセル値などが入ります。ネットワーク間のゲートウェイを介して文字データを渡すと、文字データが変形する可能性があります。たとえば、1 文字を表現するバイト数が異なる別々の言語を使用しているマシン間で文字データを渡すと、長さが大きく変わる可能性があります。RAW データがこのように変換されることはありません。

表の列ごとに適切な Oracle 内部データ型を選択するのは、データベースの設計者の責任です。OCI プログラムは、文字データおよびバイト配列データの表現方法、および OCI プログラムの変数と Oracle 表の間でのデータの変換方法を多数知っておく必要があります。

配列に文字が入っている場合、OCI コール内の配列の長さを示すパラメータは、常に文字単位ではなくバイト単位で渡され、バイト単位で戻されます。

UROWID

ユニバーサル ROWID (UROWID) は、Oracle 表の論理 ROWID と物理 ROWID、およびゲートウェイ経由でアクセスする DB2 表などの外部表の ROWID を格納できるデータ型です。論理 ROWID は、索引構成表 (IOT) の行に対する主キー・ベースの論理識別子です。

UROWID データ型の列を使用するには、COMPATIBLE 初期化パラメータの値をリリース 8.1 以上に設定する必要があります。

次のホスト変数を、ユニバーサル ROWID にバインドできます。

- SQLT_CHR (VARCHAR2)
- SQLT_VCS (VARCHAR)
- SQLT_STR (ヌル文字で終了する文字列)
- SQLT_LVC (LONG VARCHAR)
- SLQT_AFC (CHAR)
- SQLT_AVC (CHARZ)
- SQLT_VST (OCI 文字列)
- SQLT_RDD (ROWID 記述子)

外部データ型

表 3-2 は、外部データ型のデータ型コードを示しています。この表ではデータ型ごとに、Oracle 内部データの変換元または変換先となる、C のプログラム変数の型を示しています。

表 3-2 外部データ型とそのコード

外部データ型	コード	プログラム変数	OCI 定義の定数
VARCHAR2	1	char[n]	SQLT_CHR
NUMBER	2	unsigned char[21]	SQLT_NUM
8 ビット符号付き INTEGER	3	signed char	SQLT_INT
16 ビット符号付き INTEGER	3	signed short、signed int	SQLT_INT
32 ビット符号付き INTEGER	3	signed int、signed long	SQLT_INT
FLOAT	4	float、double	SQLT_FLT
ヌル文字で終了する STRING	5	char[n+1]	SQLT_STR
VARNUM	6	char[22]	SQLT_VNU
LONG	8	char[n]	SQLT_LNG
VARCHAR	9	char[n+sizeof(short integer)]	SQLT_VCS
DATE	12	char[7]	SQLT_DAT
VARRAW	15	unsigned char[n+sizeof(short integer)]	SQLT_VBI
RAW	23	unsigned char[n]	SQLT_BIN
LONG RAW	24	unsigned char[n]	SQLT_LBI
UNSIGNED INT	68	unsigned	SQLT_UIN
LONG VARCHAR	94	char[n+sizeof(integer)]	SQLT_LVC
LONG VARRAW	95	unsigned char[n+sizeof(integer)]	SQLT_LVB
CHAR	96	char[n]	SQLT_AFC
CHARZ	97	char[n+1]	SQLT_AVC
ROWID 記述子	104	OCIRowid *	SQLT_RDD
NAMED DATA TYPE	108	struct	SQLT_NTY
REF	110	OCIRef	SQLT_REF

表 3-2 外部データ型とそのコード（続き）

外部データ型	コード	プログラム変数	OCI 定義の定数
キャラクタ LOB 記述子	112	OCILobLocator (注意 2 を参照)	SQLT_CLOB
バイナリ LOB 記述子	113	OCILobLocator (注意 2 を参照)	SQLT_BLOB
バイナリ FILE 記述子	114	OCILobLocator	SQLT_FILE
OCI 文字列型	155	OCIString	SQLT_VST (注意 1 を参照)
OCI 日付型	156	OCIDate*	SQLT_ODT (注意 1 を参照)
ANSI DATE 記述子	184	OCIDateTime *	SQLT_DATE
TIMESTAMP 記述子	187	OCIDateTime *	SQLT_TIMESTAMP
TIMESTAMP WITH TIME ZONE 記述子	188	OCIDateTime *	SQLT_TIMESTAMP_TZ
INTERVAL YEAR TO MONTH 記述子	189	OCIInterval *	SQLT_INTERVAL_YM
INTERVAL DAY TO SECOND 記述子	190	OCIInterval *	SQLT_INTERVAL_DS
TIMESTAMP WITH LOCAL TIME ZONE 記述子	232	OCIDateTime *	SQLT_TIMESTAMP_LTZ

注意：

(1) これらのデータ型の使用方法の詳細は、[第 11 章「オブジェクト・リレーショナル・データ型」](#)を参照してください。

(2) OTT で生成したデータ型マッピングを使用しているアプリケーションでは、CLOB は OCIClobLocator としてマッピングされ、BLOB は OCIBlobLocator としてマッピングされます。詳細は、[第 14 章「Object Type Translator \(OTT\)」](#)を参照してください。

注意： データ長が n となっている場合、 n は可変でプログラム（ROWID の場合はオペレーティング・システム）の要件に依存します。

次に、それぞれの外部データ型について説明します。リリース 8.0 から追加された新しいデータ型の詳細は、3-18 ページの「[新しい外部データ型](#)」を参照してください。

次の 3 つの型は PL/SQL に対する内部型で、OCI では値として戻すことができません。

- ブール、SQLT_BOL
- 索引付き表、SQLT_TAB
- レコード、SQLT_REC

VARCHAR2

VARCHAR2 データ型は、最大長 4000 バイトの可変長文字列です。

注意： Oracle オブジェクトを使用する場合は、一連の事前定義済み OCI 関数を使用することにより、特殊な **OCIString** 外部データ型を扱うことができます。このデータ型の詳細は、[第 11 章「オブジェクト・リレーションアル・データ型」](#)を参照してください。

入力

OCIBindByName() または OCIBindByPos() コールでは、`value_sz` パラメータによって長さが決定されます。

`value_sz` パラメータが 0 (ゼロ) より大きい場合は、指定のバイト数のみがプログラムのバッファ・アドレスの先頭から読み取られて、バインド変数値が取得されます。後続の空白は削除され、結果の値は SQL 文または PL/SQL ブロックで使用されます。INSERT 文の場合、結果の値がデータベース列で定義された長さより長いと、INSERT は失敗しエラーが戻されます。

注意： 後続の NULL は除去されません。変数はヌル文字で終了させずに、空白埋めをする必要があります。

`value_sz` パラメータが 0 (ゼロ) の場合、バインド変数は、実際の内容とは無関係に NULL として処理されます。NULL は、SQL 文のバインド変数値として使用できます。NOT NULL 整合性制約付きの列に NULL を挿入しようとする、Oracle ではエラーとなり、行は挿入されません。

Oracle 内部 (列) データ型が NUMBER の場合、数値の文字表現が含まれている文字列からの入力は有効です。入力文字列は内部数値形式に変換されます。VARCHAR2 文字列に無効な変換文字が含まれている場合、Oracle ではエラーを戻され、値はデータベースに挿入されません。

出力

`OCIDefineByPos()` コールの `value_sz` パラメータで、または、PL/SQL ブロックの場合は `OCIBindByName()` か `OCIBindByPos()` の `value_sz` パラメータで、戻り値の希望の長さを指定します。長さとしてゼロを指定した場合、データは戻されません。

`OCIDefineByPos()` の `rlenp` パラメータを省略すると、戻される値はバッファ長に達するまで空白が埋め込まれ、空白文字の文字列として `NULL` が戻されます。`rlenp` を指定した場合、戻り値は空白埋めされません。かわりに、実際の長さが `rlenp` パラメータに戻されます。

`NULL` が戻されたかどうか、または文字の切捨てが発生したかどうかをチェックするには、`OCIDefineByPos()` コールにインジケータ・パラメータを指定します。Oracle では、`NULL` がフェッチされた場合にインジケータ・パラメータが -1 に設定され、戻り値が切り捨てられている場合はインジケータ・パラメータが元の列長に設定されます。それ以外の場合は、0 (ゼロ) が設定されます。インジケータ・パラメータを指定していない場合、`NULL` が選択されると、フェッチ・コールでエラー・コード `OCI_SUCCESS_WITH_INFO` が戻されます。エラーについての診断情報の取出しでは、`ORA-01405` が戻されます。

関連項目： 2-36 ページ「インジケータ変数」

内部 `NUMBER` データ型から文字列への出力を要求することもできます。数値変換は、使用しているシステムのグローバリゼーション・サポートによって規定される規則に従って行われます。たとえば、システムが、小数点としてピリオドではなくカンマを認識するように構成されている場合があります。

NUMBER

通常は、`NUMBER` を外部データ型として使用する必要はありません。`NUMBER` を使用すると、Oracle では 21 バイトの内部バイナリ形式で数値を戻し、入力時にこの形式が必要になります。完全な情報が必要な場合は、次の説明を参照してください。

注意： Oracle データベース・サーバー内でオブジェクトを使用する場合は、一連の事前定義済み OCI 関数を使用して、特殊な `OCINumber` データ型を扱うことができます。このデータ型の詳細は、第 11 章「オブジェクト・リレーショナル・データ型」を参照してください。

Oracle では、`NUMBER` データ型の値を可変長形式で格納します。最初のバイトは指数であり、その後に 1 ～ 20 個の仮数バイトが続きます。指数バイトの上位ビットは符号ビットです。正数の場合はそのビットが設定され、負数の場合は消去されます。下位の 7 ビットは指数を表します。この指数はオフセット 65 で基本 100 の数字です。

10 進数の指数を計算するには、基本 100 の指数に 65 を加算し、数値が正数の場合はさらに 128 を加算します。数値が負数の場合も同様に計算しますが、後でビットが反転されます。

たとえば、-5 の場合は、基本 100 の指数 = 62 (0x3e) になります。したがって、10 進数の指数は、 $(\sim 0x3e) - 128 - 65 = 0xc1 - 128 - 65 = 193 - 128 - 65 = 0$ になります。

各仮数バイトは基本 100 で範囲 1 ～ 100 の数字です。正数の場合は、それに 1 を加算した数字となります。したがって、値 5 に対する仮数は 6 となります。負数の場合、1 を加算するかわりに、その数字が 101 から減算されます。つまり、値 -5 に対する仮数は 96 (101-5) となります。負数には、102 が含まれているバイトがデータ・バイトの後に追加されます。ただし、20 個の仮数バイトを持つ負数には、102 のバイトは後続しません。仮数バイトは基本 100 であるため、各バイトは 2 桁の 10 進数値を表すことができます。仮数は正規化され、先行する 0 (ゼロ) は格納されません。

仮数を表すために最大 20 個のデータ・バイトを使用できます。ただし、精度が保証されているのは 19 個のみです。19 個のデータ・バイトは、それぞれ基本 100 を表し、Oracle NUMBER に対して 38 桁の最大精度を提供します。

OCIDefineByPos() コールの `dtype` パラメータにデータ型コード 2 を指定すると、プログラムではこの Oracle 内部形式で数値データを受け取ります。出力変数は、可能な最大の数値を格納できる 21 バイトの配列である必要があります。数値を表すバイトのみが戻されることに注意してください。空白埋めまたはヌル文字終了はありません。戻されたバイト数を調べる場合は、NUMBER ではなく VARNUM 外部型を使用します。Oracle 内部数値書式の例は、3-13 ページの「VARNUM」の説明を参照してください。

INTEGER

INTEGER データ型では数値を変換します。外部 INTEGER は符号付きの 2 進数です。バイト単位のサイズはシステムによって異なります。変数内のバイトの順序は、ホスト・システム・アーキテクチャによって決まります。長さ指定は、入力でも出力でも必須です。Oracle から戻される数値が整数でない場合、小数部分は廃棄され、エラーやその他の情報は戻されません。戻される数値がシステムの符号付き INTEGER で表現できない場合は、「変換時にオーバーフローが発生しました (overflow on conversion)」エラーが戻されます。

FLOAT

FLOAT データ型は、小数部分のある数値、または INTEGER では表現できない数値を処理します。数値はホスト・システムの浮動小数点書式で表されます。通常、長さは 4 バイトか 8 バイトのいずれかです。長さ指定は、入力および出力の両方について必要です。

Oracle の数値の内部形式は 10 進数で、大部分の浮動小数点は 2 進数です。したがって、Oracle では、数値を浮動小数点で表現するよりもより高い精度で表現できます。

注意： FLOAT および NUMBER 間の変換時には四捨五入のエラーが発生する場合もあります。たとえば、問合せでバインド変数として FLOAT を使用すると、ORA-01403 エラーが戻されることがあります。この状況を回避するには、FLOAT を STRING に変換し、データ型コード 1 または 5 を設定します。

STRING

ヌル文字で終了する `STRING` 形式は、文字列にヌル終端文字が必要なことを除けば、`VARCHAR2` 形式（データ型コード 1）と同じように動作します。このデータ型は、特に C 言語のプログラムで有用です。

入力

`OCIBindByName()` または `OCIBindByPos()` コールで供給される文字列の長さによって、ヌル終端文字があるかどうかのスキップの範囲が制限されます。ヌル終端文字が指定された中で見つからなかった場合は、エラーとなります。

ORA-01480: STR バインド値に終了の NULL がありません。

バインド・コールで長さが指定されていない場合、OCI は、文字列の暗黙の最大長の 4000 を使用します。

文字列の最小の長さは 2 バイトです。最初の文字がヌル終端文字であり、長さが 2 と指定されている場合、許可されていれば NULL が列に挿入されます。コード 1 および 96 の型とは異なり、空白のみを含む文字列は入力時に NULL として扱われるのではなく、そのまま挿入されます。

注意： OCI のこれまでのバージョンとは異なり、リリース 8.0 以上の OCI では、ヌル文字で終了する文字列の文字列長パラメータに -1 を渡せません。

出力

ヌル終端文字は、最後の文字が戻された後に配置されます。文字列が指定のフィールド長より長い場合、文字列は切り捨てられ、出力変数の最後の文字の位置にヌル終端文字が入ります。

NULL 選択リスト項目は、最初の文字の位置にヌル終端文字を戻します。ORA-01405 エラーが戻される可能性があります。

VARNUM

VARNUM データ型は、最初のバイトに数値表現の長さが記述されることを除けば、外部 NUMBER データ型と同じです。この長さには、長さを記述するバイト自体は含まれません。最大長の VARNUM に備えて、22 バイトを確保してください。VARNUM 値を Oracle に送信するときは、長さを記述するバイトを設定します。

次の表に、Oracle 表の数値に戻される VARNUM 値の例をいくつか示します。

表 3-3 VARNUM の例

10 進数	長さバイト	指数バイト	仮数バイト	終了文字バイト
0	1	128	該当なし	該当なし
5	2	193	6	該当なし
-5	3	62	96	102
2767	3	194	28、68	該当なし
-2767	4	61	74、34	102
100000	2	195	11	該当なし
1234567	5	196	2、24、46、68	該当なし

LONG

LONG データ型では、4001 バイト以上の文字列を格納します。LONG 列には、最大 2GB (2³¹-1 バイト) を格納できます。この列の型は、長い文字列の格納およびフェッチにのみ使用します。関数、式または WHERE 句では使用できません。一般的に、LONG 列値の変換は、文字列との間で行われます。

VARCHAR

VARCHAR データ型では、可変長の文字列が格納されます。先頭の 2 バイトには文字列の長さが記述され、残りのバイトには文字列が含まれます。バインド・コールまたは定義コールで指定する文字列の長さには、この 2 バイトを含める必要があるため、送受信可能な VARCHAR 文字列の最大長は、65535 バイトではなく、65533 バイトになります。これより長い文字列を変換する場合は、LONG VARCHAR 外部データ型を使用します。

DATE

DATE データ型では、Oracle 内部日付バイナリ形式を使用して、日付の値の更新、挿入または検索を行うことができます。バイナリ形式の日付は、表 3-4 に示すとおり 7 バイトです。

表 3-4 DATE データ型の書式

バイト	1	2	3	4	5	6	7
意味	世紀	年	月	日	時	分	秒
例 (1992 年 11 月 30 日 午後 3 時 17 分)	119	192	11	30	16	18	1

世紀および年を表すバイト（バイト 1 および 2）の値は、100 を加算した表記です。最初のバイトには世紀が格納されます。この例の 1992 年の表記は、1992 を 100 で除算し、100 を加算した整数の 119 になります。2 番目のバイトには、100 を加算した年が格納され、この例では 192 になります。西暦紀元前（BCE）は 100 未満です。西暦紀元は BCE 4712 年 1 月 1 日、つまりユリウス日の第 1 日です。この日付の場合、世紀バイトは 53、年バイトは 88 です。時間バイト、分バイトおよび秒バイトは 1 を加算する表記法で表されます。時間バイトの範囲は 1 ～ 24、分および秒バイトは 1 ～ 60 です。データ作成時に時刻を指定しないと、時刻はデフォルトで深夜（1,1,1）となります。

DATE 外部データ型を使用してバイナリ形式で日付を入力する場合、データベースでは一貫性または範囲のチェックは行いません。この形式になっているすべてのデータは、入力前に注意して検証する必要があります。

注意： 日常のデータベース操作で、Oracle 外部 DATE データ型が必要になることはほとんどありません。一般にプログラムでは DD-MON-YY などのように文字形式でデータを扱うため、DATE は文字形式に変換した方がはるかに便利です。

DATE 列はプログラム内で文字列に変換されるとき、使用中セッションのデフォルトの書式マスク、または INIT.ORA ファイルでの指定に従って戻されます。

注意： Oracle データベースでオブジェクトを使用している場合は、一連の事前定義済み OCI 関数を使用することにより、特殊な **OCIDate** データ型を扱うことができます。

- このデータ型の詳細は、第 11 章「オブジェクト・リレーショナル・データ型」を参照してください。
- DATETIME データ型および INTERVAL データ型の詳細は、3-22 ページの「日時および時間隔のデータ型記述子」を参照してください。

RAW

RAW データ型は、たとえば図形文字列の格納など、Oracle で解釈されないバイナリ・データまたはバイト列で使用されます。RAW 列の最大長は 2000 バイトです。

関連項目：『Oracle9i SQL リファレンス』

Oracle 表内の RAW データ（ロー・データ）がプログラムで文字列に変換される場合、データは 16 進文字コードで表されます。RAW データの各バイトは、それぞれの値を示す '00' ～ 'FF' の 2 文字で戻されます。プログラムの文字列を Oracle 表内の RAW 列に入力する場合は、この 16 進コードを使用して文字列のデータをコード化する必要があります。

OCIDefineByPos()、OCIBindByName()、OCIBindByPos()、OCIStmtGetPieceInfo() および OCIStmtSetPieceInfo() によって提供されるピース単位機能を使用すると、RAW 列（または LONG RAW 列）を扱う挿入、更新またはフェッチを実行できます。

注意： Oracle データベースでオブジェクトを使用する場合は、一連の事前定義済み OCI 関数を使用して、特殊な **OCIRaw** データ型を扱うことができます。このデータ型の詳細は、[第 11 章「オブジェクト・リレーショナル・データ型」](#)を参照してください。

VARRAW

VARRAW データ型は、RAW データ型に類似しています。ただし、先頭の 2 バイトにはデータの長さが記述されます。バインド・コールまたは定義コールの文字列に指定する長さには、この 2 バイトが含まれている必要があります。したがって、送受信可能な VARRAW 文字列の最大長は、65535 バイトではなく、65533 バイトになります。これより長い文字列を変換する場合は、LONG VARRAW 外部データ型を使用します。

LONG RAW

LONG RAW データ型は、最大 2GB ($2^{31}-1$ バイト) の RAW データを格納できるということを除けば、RAW データ型と同じです。

UNSIGNED

UNSIGNED データ型は、符号なし 2 進整数に対して使用されます。バイト単位のサイズは、システムによって異なります。ワード内のバイトの順序は、ホスト・システムのアーキテクチャによって決まります。長さ指定は、入力でも出力でも必須です。Oracle から出力される数値が整数でない場合は、小数部分が廃棄され、エラーやその他の情報は戻されません。戻される数値がシステムの符号なし整数より大きくて表現できない場合、「変換時にオーバーフローが発生しました (overflow on conversion)」というエラーが戻されます。

LONG VARCHAR

LONG VARCHAR データ型は、Oracle の LONG 列からデータを格納する場合、またはこの列にデータを格納する場合に使用します。LONG VARCHAR の先頭の 4 バイトには項目の長さが記述されます。したがって、格納される項目の最大長は、 $2^{31}-5$ バイトです。

LONG VARRAW

LONG VARRAW データ型は、Oracle LONG RAW 列からデータを格納する場合、またはこの列にデータを格納する場合に使用します。長さは、先頭の 4 バイトに記述されます。最大長は、 $2^{31}-5$ バイトです。

CHAR

CHAR データ型は最長 2000 の文字列です。CHAR 文字列は空白埋め比較方法で比較されます。

関連項目：『Oracle9i SQL リファレンス』

入力

長さは、OCIBindByName() または OCIBindByPos() コールの `value_sz` パラメータによって決まります。

注意： バッファの全内容 (`value_sz` の文字数) は、後続の空白または NULL もすべて含めてデータベースに渡されます。

`value_sz` パラメータが 0 (ゼロ) である場合、バインド変数は、実際の内容とは無関係に NULL として処理されます。NULL は、SQL 文のバインド変数値として使用できます。NOT NULL 整合性制約付きの列に NULL を挿入しようとすると、Oracle ではエラーとなり、行は挿入されません。

CHAR の場合、`value_sz` パラメータには負の値を使用できません。

Oracle 内部 (列) データ型が NUMBER の場合、数値の文字表現が含まれている文字列からの入力は有効です。入力文字列は内部数値形式に変換されます。CHAR 文字列に無効な変換文字が含まれている場合、Oracle ではエラーが戻され、値は挿入されません。数値変換は、使用しているシステムのグローバリゼーション・サポートの設定値によって規定される規則に従って行われます。たとえば、システムが、ピリオドではなくカンマを小数点として認識するように構成されている場合もあります。

出力

`OCIDefineByPos()` コールの `value_sz` パラメータで、戻り値の希望の長さを指定します。長さとしてゼロを指定した場合、データは戻されません。

`OCIDefineByPos()` の `rlenp` パラメータを省略すると、戻される値はバッファの長さ達するまで空白が埋め込まれ、空白文字の文字列として `NULL` が戻されます。`rlenp` を指定した場合、戻り値は空白埋めされません。かわりに、実際の長さが `rlenp` パラメータに戻されます。

`NULL` が戻されたかどうか、または文字の切捨てが発生したかどうかをチェックするには、`OCIDefineByPos()` コールにインジケータ・パラメータまたはインジケータ・パラメータの配列を指定します。インジケータ・パラメータには、`NULL` がフェッチされた場合は -1 が設定され、値が切り捨てられて戻された場合は元の列の長さが設定されます。それ以外の場合は、0 (ゼロ) が設定されます。インジケータ・パラメータを指定していない場合、`NULL` が選択されると、フェッチ・コールで `ORA-01405` エラーが戻されます。

関連項目： 2-36 ページ「[インジケータ変数](#)」

内部 `NUMBER` データ型から文字列への出力を要求することもできます。数値変換は、使用しているシステムのグローバリゼーション・サポートの設定値によって規定される規則に従って行われます。たとえば、システムが、ピリオド (.) ではなくカンマ (,) を小数点として認識するように構成されている場合もあります。

CHARZ

`CHARZ` 外部データ型は、入力時に文字列がヌル文字で終了していること、および出力時に `Oracle` によって文字列の最後にヌル終端文字が配置されることを除けば、`CHAR` データ型と同じです。ヌル終端文字は入出力時に文字列を区切るためのもので、表のデータの一部ではありません。

入力時に、長さのパラメータは、ヌル終端文字も含めた正確な長さを示す必要があります。たとえば、`C` の配列が次のように宣言されたとします。

```
char my_num[] = "123.45";
```

この場合、`my_num` のバインド時の長さのパラメータは7になっている必要があります。この例で他の値を指定すると、エラーが戻されます。

新しい外部データ型

次の新しい外部データ型は、リリース 8.0 から導入されました。これらのデータ型は、Oracle7 Server へ接続した場合にはサポートされません。

注意： 内部データ型および外部データ型は、いずれもそれぞれのデータ型コードに対応する Oracle 定義の定数値 (SQLT_NTY、SQLT_REF など) を含んでいます。この章に出てくるすべての型についての定数はリストしていませんが、この項では新しい Oracle データ型を説明する際に、それらの定数を使用します。また、データ型定数は、これらの新しい型を参照するときに、このマニュアルの他の章でも使用されます。

名前付きデータ型（オブジェクト、VARRAY、NESTED TABLE）

名前付きデータ型は、SQL の CREATE TYPE コマンドで指定するユーザー定義の型です。例として、オブジェクト型、VARRAY、NESTED TABLE などがあります。OCI では、名前付きデータ型 (named data type) は型のホスト言語による表現を意味します。SQLT_NTY データ型コードは、名前付きデータ型をバインドまたは定義する際に使用します。

C アプリケーションでは、名前付きデータ型は C 構造体として表されます。このような構造体は、データベース内に格納された型から Object Type Translator (OTT) を使用して生成されます。これらの型は、OCI_TYPECODE_OBJECT に対応します。

関連項目：

- OCI で名前付きデータ型を使用する場合の詳細は、このマニュアルの第 II 部を参照してください。
- 名前付きデータ型の C 構造体としての表現方法の詳細は、[第 14 章「Object Type Translator \(OTT\)」](#)を参照してください。

REF

これは名前付きデータ型への参照です。REF の C 言語表現は、**OCIRef *** 型として宣言された変数です。SQLT_REF データ型コードは、REF をバインドまたは定義する際に使用します。

REF にアクセスできるのは、OCI アプリケーションをオブジェクト・モードで初期化した場合のみです。サーバーから REF が取り出されると、REF はクライアント側のオブジェクト・キャッシュに格納されます。

アプリケーションで使用する REF を割り当てるには、変数を REF へのポインタとして宣言し、OCI_TYPECODE_REF を *typecode* パラメータとして渡す OCIObjectNew() をコールします。

関連項目： OCI で REF を使用する場合は、このマニュアルの第 II 部を参照してください。

ROWID 記述子

ROWID データ型は、データベースの表の特定の行を識別するために使用されます。ROWID は、次のように、問合せの選択リスト項目になっている場合があります。

```
SELECT ROWID, ename, empno FROM emp
```

このケースでは、戻された ROWID を後続の DELETE 文で使用できます。

SELECT FOR UPDATE を実行する場合、ROWID は暗黙的に戻されます。文ハンドルで OCIAttrGet() を使用して、この ROWID をユーザーが割り当てた ROWID 記述子に読み込み、後続の UPDATE 文で使用できます。プリフェッチ操作では SELECT FOR UPDATE のすべての ROWID がフェッチされるため、プリフェッチを使用してから単一行フェッチを行います。

ROWID へのアクセスは ROWID 記述子を使用して行います。この記述子はバインド変数または定義変数として使用できます。

関連項目： ROWID 記述子の使用方法の詳細は、2-15 ページの「[記述子](#)」および 2-39 ページの「[位置指定の更新および削除](#)」を参照してください。

LOB 記述子

LOB (ラージ・オブジェクト) には、長さが最大 4GB のバイナリ・データまたは文字データを格納できます。バイナリ・データは BLOB (バイナリ LOB) に格納され、文字データは、CLOB (キャラクタ LOB) または NCLOB (各国語キャラクタ LOB) に格納されます。

LOB 値は、データベースの行データとともに、インラインに格納される場合とされない場合があります。いずれの場合も、LOB は、データベース・サーバーのトランザクションを完全にサポートしています。データベースの表には、別の記憶領域にある LOB 値を指す LOB ロケータが格納されます。

OCI アプリケーションで選択リストに LOB 列または属性を含む SQL 問合せを発行した場合、問合せ結果のフェッチでは、実際の LOB 値ではなくロケータが戻されます。OCI では、LOB ロケータは **OCILobLocator** 型の変数にマッピングされます。

関連項目：

- LOB ロケータを含む記述子の詳細は、2-15 ページの「[記述子](#)」を参照してください。
- LOB の詳細は、『Oracle9i SQL リファレンス』および『Oracle9i アプリケーション開発者ガイド - ラージ・オブジェクト』を参照してください。

LOB に対する OCI 関数は、LOB ロケータを引数の 1 つとして受け取ります。LOB にデータが含まれているかどうかに関係なく、OCI 関数はその LOB を指すロケータがすでに作成済みであるとみなします。

バインド操作および定義操作は、OCIDescriptorAlloc() 関数で割り当てられる LOB ロケータ上で実行されます。

SQL または OCIObjectPin() を使用して、常に最初にロケータをフェッチしてから、ロケータを使用して操作を実行します。OCI 関数が実際の LOB 値をパラメータとして受け取ることはありません。

関連項目： OCI LOB 関数の詳細は、[第 7 章「LOB と FILE の操作」](#)を参照してください。

LOB をバインドまたは定義するために使用できるデータ型コードは、次のとおりです。

- SQLT_BLOB – バイナリ LOB データ型
- SQLT_CLOB – キャラクタ LOB データ型

NCLOB は、次の要件を伴う CLOB の特殊型です。

- NCLOB に対する書込みおよび読込みを行うには、キャラクタ・セット・フォーム (*csfrm*) パラメータを SQLCS_NCHAR に設定する必要があります。
- CLOB および NCLOB を含むコールの **amount** (*amtp*) パラメータは常に、バイトではなく固定幅キャラクタ・セットの文字として解釈されます。

関連項目： [7-5 ページ「LOB および FILE の関数」](#)

BFILE

BFILE データ型を使用すると、Oracle データベース以外のファイル・システムに格納されているファイル LOB へのアクセスが可能になります。Oracle では現在、バイナリ・ファイル、つまり BFILE へのアクセスのみをサポートしています。

BFILE の列または属性には、サーバーのファイル・システムにあるバイナリ・ファイルへのポインタとして機能するファイル LOB ロケータが格納されます。このロケータにディレクトリの別名とファイル名が維持されています。

バイナリ・ファイル LOB は、トランザクションでは使用できません。むしろ、ベースとなるオペレーティング・システムにより、ファイルの整合性と恒久性が提供されます。サポートされるファイルの最大サイズは、4GB です。

データベース管理者は、ファイルが存在していること、および Oracle プロセスがそのファイルに対してオペレーティング・システムの読取り権限を持っていることを確認する必要があります。

BFILE データ型では、大きなバイナリ・ファイルの読取り専用サポートが可能です。Oracle からファイルを変更することはできません。Oracle では、ファイル・データにアクセスするための API を提供しています。

FILE をバインドまたは定義するために使用できるデータ型コードは、次のとおりです。

- SQLT_BFILE – バイナリ FILE LOB データ型

関連項目： ディレクトリ別名の詳細は、『Oracle9i アプリケーション開発者ガイド - ラージ・オブジェクト』を参照してください。

BLOB

BLOB データ型には、非構造化バイナリ・ラージ・オブジェクトが格納されます。BLOB は、キャラクタ・セットとしての意味を持たないビットストリームであると考えられます。BLOB には、最大 4GB のバイナリ・データを格納できます。

BLOB は完全なトランザクション・サポートを備えています。OCI を通じて行った変更は、完全にトランザクションで使用できます。BLOB 値に対して行った操作は、コミットまたはロールバックできます。1 つのトランザクション内で変数の BLOB ロケータを保存し、次にそれを別のトランザクションまたはセッションで使用することはできません。

CLOB

CLOB データ型には固定幅または可変幅の文字データが格納されます。CLOB には、最大 4GB の文字データを格納できます。

CLOB は完全なトランザクション・サポートを備えています。OCI を通じて行った変更は、完全にトランザクションで使用できます。CLOB 値に対して行った操作は、コミットまたはロールバックできます。1 つのトランザクション内で変数の CLOB ロケータを保存し、次にそれを別のトランザクションまたはセッションで使用することはできません。

NCLOB

NCLOB は、CLOB の各国語キャラクタ・バージョンです。NCLOB は、固定幅、シングルのバイト、マルチバイトの各国語キャラクタ・セット (NCHAR)、または可変幅のキャラクタ・セット・データを格納します。NCLOB には、最大 4GB の文字データを格納できます。

NCLOB は完全なトランザクション・サポートを備えています。OCI を通じて行った変更は、完全にトランザクションで使用できます。NCLOB 値に対して行った操作は、コミットまたはロールバックできます。1 つのトランザクション内で変数の NCLOB ロケータを保存し、次にそれを別のトランザクションまたはセッションで使用することはできません。

日時および時間隔のデータ型記述子

ここでは、日時および時間隔のデータ型記述子について簡単に説明します。

関連項目： 詳細は、『Oracle9i SQL リファレンス』を参照してください。

ANSI DATE

ANSI DATE データ型は DATE データ型が基本になっていますが、時間部分は含まれていません（したがって、タイム・ゾーンはありません）。ANSI DATE は、DATE データ型の ANSI 仕様に従います。ANSI DATE を DATE データ型または TIMESTAMP データ型に割り当てると、Oracle の DATE や TIMESTAMP の時間部分は 0（ゼロ）に設定されます。DATE データ型または TIMESTAMP データ型を ANSI DATE に割り当てると、時間部分は無視されます。

かわりに、日付と時間の両方を含む TIMESTAMP データ型の使用をお勧めします。

TIMESTAMP

TIMESTAMP は、DATE データ型を拡張したデータ型です。このデータ型には、DATE データ型の年、月、日に加えて、時間、分および秒の各値が格納されます。タイム・ゾーンはありません。TIMESTAMP データ型の書式は次のとおりです。

`TIMESTAMP(fractional_seconds_precision)`

fractional_seconds_precision（オプション）では、SECOND 日時フィールドの小数部分の桁数を指定します。桁数は 0～9 を指定でき、デフォルト値は 6 です。

TIMESTAMP WITH TIME ZONE

TIMESTAMP WITH TIME ZONE (TSTZ) データ型は TIMESTAMP の改良型で、その値に明示的なタイム・ゾーン置換が含まれています。タイム・ゾーン置換は、現地時間と UTC（協定世界時、以前はグリニッジ平均時）との差異（時分による）です。TIMESTAMP WITH TIME ZONE データ型の書式は次のとおりです。

`TIMESTAMP(fractional_seconds_precision) WITH TIME ZONE`

fractional_seconds_precision（オプション）では、SECOND 日時フィールドの小数部分の桁数を指定します。桁数は 0～9 を指定でき、デフォルト値は 6 です。

2 つの TIMESTAMP WITH TIME ZONE の値が UTC で同じ時刻を表している場合は、データに格納されている TIME ZONE オフセットに関係なく同一とみなされます。

TIMESTAMP WITH LOCAL TIME ZONE

TIMESTAMP WITH LOCAL TIME ZONE (TSLTZ) データ型は TIMESTAMP の別の改良型で、その値にタイム・ゾーン置換が含まれています。格納される値の形式は TIMESTAMP と同じです。このデータ型と TIMESTAMP WITH TIME ZONE との違いは、データベースに格納されるデータがデータベース・タイム・ゾーンに正規化されること、およびタイム・ゾーン置換が列データの一部として格納されないことです。ユーザーがデータを取り出すと、データは、ユーザーのローカル・セッションのタイム・ゾーンで戻されます。

タイム・ゾーン置換は、現地時間と UTC（協定世界時、以前はグリニッジ平均時）との差異（時分による）です。TIMESTAMP WITH LOCAL TIME ZONE データ型の書式は次のとおりです。

TIMESTAMP(*fractional_seconds_precision*) WITH LOCAL TIME ZONE

fractional_seconds_precision (オプション) では、SECOND 日時フィールドの小数部分の桁数を指定します。桁数は 0～9 を指定でき、デフォルト値は 6 です。

INTERVAL YEAR TO MONTH

INTERVAL YEAR TO MONTH データ型には、YEAR および MONTH 日時フィールドを使用した期間が格納されます。INTERVAL YEAR TO MONTH データ型の書式は次のとおりです。

INTERVAL YEAR(*year_precision*) TO MONTH

year_precision (オプション) では、YEAR 日時フィールドの桁数を指定します。*year_precision* のデフォルト値は 2 です。

INTERVAL DAY TO SECOND

INTERVAL DAY TO SECOND データ型には、日、時間、分および秒による期間が格納されます。INTERVAL DAY TO SECOND データ型の書式は次のとおりです。

INTERVAL DAY (*day_precision*) TO SECOND(*fractional_seconds_precision*)

次のように指定します。

- *day_precision* では、DAY 日時フィールドの桁数を指定します。これはオプションです。指定できる値は 0～9 で、デフォルト値は 2 です。

fractional_seconds_precision では、SECOND 日時フィールドの小数部分の桁数を指定します。これはオプションです。指定できる値は 0～9 で、デフォルト値は 6 です。

日時を使用する際の予期しない結果の回避

注意： 日時データの DML 操作での予期しない結果を回避するために、組込み SQL 関数の `DBTIMEZONE` と `SESSIONTIMEZONE` を問い合せて、データベースとセッションのタイム・ゾーンを確認できます。タイム・ゾーンを手動で設定していない場合、Oracle では、デフォルトでオペレーティング・システムのタイム・ゾーンが使用されます。オペレーティング・システムのタイム・ゾーンが有効な Oracle タイム・ゾーンでない場合、Oracle では、UTC がデフォルト値として使用されます。

C オブジェクト・リレーショナル・データ型マッピング

OCI では、ユーザー定義のデータ型を C 表現にマッピングするために使用される、Oracle 定義の C データ型がサポートされています (`OCINumber`、`OCIArray` など)。OCI では、これらのデータ型を操作するための一連の OCI コールが用意されています。これらのデータ型は、OCI 外部データ型コードとともに、バインド操作および定義操作で使用できます。

関連項目： Oracle 定義の C データ型の使用方法の詳細は、[第 11 章「オブジェクト・リレーショナル・データ型」](#)を参照してください。

データ変換

[表 3-5](#) および [表 3-6](#) は、リリース 7.3 で使用可能な全データ型について、内部データ型から外部データ型、および外部データ型から内部列表現へのサポートされている変換を示しています。リリース 7.3 より後に追加された新しいデータ型のデータ変換の詳細は、次のとおりです。

- データベースに格納されている REF は、出力時に `SQLT_REF` に変換されます。
- `SQLT_REF` は、入力時に REF の内部表現に変換されます。
- データベースに格納されている名前付きデータ型は、出力時に `SQLT_NTY` に変換されます (アプリケーション内では C 構造体で表現されます)。
- `SQLT_NTY` (アプリケーション内では C 構造体で表現されます) は、入力時に、対応する型の内部表現に変換されます。

表の幅が限られているため、LOB については後の表にリストされています。

関連項目： `OCIString`、`OCINumber` およびその他の新しいデータ型の詳細は、[第 11 章「オブジェクト・リレーショナル・データ型」](#)を参照してください。

表 3-5 データ変換

外部 データ型	内部データ型								
	VARCHAR2	NUMBER	LONG	ROWID	UROWID	DATE	RAW	LONG RAW	CHAR
VARCHAR	I/O	I/O	I/O	I/O(1)	I/O(1)	I/O(2)	I/O(3)	I/O(3)	
NUMBER	I/O(4)	I/O	I	—	—	—	—	—	I/O(4)
INTEGER	I/O(4)	I/O	I	—	—	—	—	—	I/O(4)
FLOAT	I/O(4)	I/O	I	—	—	—	—	—	I/O(4)
STRING	I/O	I/O	I/O	I/O(1)	I/O(1)	I/O(2)	I/O(3)	I/O(3、5)	I/O
VARNUM	I/O(4)	I/O	I	—	—	—	—	—	I/O(4)
DECIMAL	I/O(4)	I/O	I	—	—	—	—	—	I/O(4)
LONG	I/O	I/O	I/O	I/O(1)	I/O(1)	I/O(2)	I/O(3)	I/O(3、5)	I/O
VARCHAR	I/O	I/O	I/O	I/O(1)	I/O(1)	I/O(2)	I/O(3)	I/O(3、5)	I/O
DATE	I/O	—	I	—	—	I/O	—	—	I/O
VARRAW	I/O(6)	—	I(5、6)	—	—	—	I/O	I/O	I/O(6)
RAW	I/O(6)	—	I(5、6)	—	—	—	I/O	I/O	I/O(6)
LONG RAW	O(6)	—	I(5、6)	—	—	—	I/O	I/O	O(6)
UNSIGNED	I/O(4)	I/O	I	—	—	—	—	—	I/O(4)
LONG VARCHAR	I/O	I/O	I/O	I/O(1)	I/O(1)	I/O(2)	I/O(3)	I/O(3、5)	I/O
LONG VARRAW	I/O(6)	—	I(5、6)	—	—	—	I/O	I/O	I/O(6)
CHAR	I/O	I/O	I/O	I/O(1)	I/O(1)	I/O(2)	I/O(3)	I(3)	I/O

表 3-5 データ変換（続き）

外部		内部データ型							
データ型		VARCHAR2	NUMBER	LONG	ROWID	UROWID	DATE	RAW	LONG RAW CHAR
CHARZ		I/O	I/O	I/O	I/O(1)	I/O(1)	I/O(2)	I/O(3)	I(3) I/O
ROWID 記述子	I(1)	—	—	I/O	I/O	—	—	—	I(1)
注意：								凡例：	
(1) 入力時、ホスト文字列は Oracle ROWID/UROWID 形式である必要があります。								I = 入力時のみ有効な変換	
出力時、列値は Oracle ROWID/UROWID 形式で戻されます。								O = 出力時のみ有効な変換	
(2) 入力時、ホスト文字列は Oracle DATE 文字形式である必要があります。								I/O = 入力時または出力時に有効な変換	
出力時、列値は Oracle DATE 形式で戻されます。									
(3) 入力時、ホスト文字列は 16 進フォーマットである必要があります。									
出力時、列値は 16 進フォーマットで戻されます。									
(4) 出力時、列値が有効な数値を表している必要があります。									
(5) 長さは 2000 以下である必要があります。									
(6) 入力時、列値は 16 進フォーマットで格納されます。									
出力時、列値は 16 進フォーマットである必要があります。									

LOB データ型記述子のデータ変換

表 3-6 LOB のデータ変換

外部データ型	内部 CLOB	内部 BLOB
VARCHAR	I/O	—
CHAR	I/O	—
LONG	I/O	—
LONG VARCHAR	I/O	—
RAW	—	I/O
VARRAW	—	I/O
LONG RAW	—	I/O
LONG VARRAW	—	I/O

日時および時間隔データ型のデータ変換

文字データ型のいずれかを、日時列または時間隔列のフェッチ操作または挿入操作で使用するホスト変数に対して使用することもできます。Oracle では、文字データ型と日時 / 時間隔データ型との間の変換が行われます。

表 3-7 日時および時間隔データ型のデータ変換

外部データ型 / 内部データ型	VARCHAR、 CHAR	DATE	TS	TSTZ	TSLTZ	INTERVAL YEAR TO MONTH	INTERVAL DAY TO SECOND
VARCHAR2、CHAR	I/O	I/O	I/O	I/O	I/O	I/O	I/O
DATE	I/O	I/O	I/O	I/O	I/O	—	—
OCI DATE	I/O	I/O	I/O	I/O	I/O	—	—
ANSI DATE	I/O	I/O	I/O	I/O	I/O	—	—
TIMESTAMP (TS)	I/O	I/O	I/O	I/O	I/O	—	—
TIMESTAMP WITH TIME ZONE (TSTZ)	I/O	I/O	I/O	I/O	I/O	—	—
TIMESTAMP WITH LOCAL TIME ZONE (TSLTZ)	I/O	I/O	I/O	I/O	I/O	—	—
INTERVAL YEAR TO MONTH	I/O	—	—	—	—	I/O	—
INTERVAL DAY TO SECOND	I/O	—	—	—	—	—	I/O

注意：タイム・ゾーンがあるソースを、タイム・ゾーンのないターゲットに割り当てると、ソースのタイム・ゾーン部分は無視されます。タイム・ゾーンのないソースを、タイム・ゾーンがあるターゲットに割り当てると、ターゲットのタイム・ゾーンには、セッションのデフォルト・タイム・ゾーンが設定されます。

(0) Oracle DATE を TIMESTAMP に割り当てると、DATE の TIME 部分が TIMESTAMP にコピーされます。TIMESTAMP を Oracle DATE に割り当てると、割当て後の DATE の TIME 部分には 0 (ゼロ) が設定されます。この 0 (ゼロ) 設定は、Oracle DATE から ANSI 準拠の DATETIME データ型へのアップグレードを容易にするために行われます。

(1) ANSI DATE を Oracle DATE または TIMESTAMP に割り当てると、Oracle DATE の TIME 部分と TIMESTAMP には 0 (ゼロ) が設定されます。Oracle DATE または TIMESTAMP を ANSI DATE に割り当てると、TIME 部分は無視されます。

(2) DATETIME を文字列に割り当てると、DATETIME は、セッションのデフォルトの DATETIME 形式によって変換されます。文字列を DATETIME に割り当てる場合は、その文字列にセッションのデフォルトの DATETIME 形式に基づく有効な DATETIME 値が含まれている必要があります。

(3) 文字列を INTERVAL に割り当てる場合、その文字列が有効な INTERVAL 文字形式である必要があります。

日時および時間隔データ型のデータ変換に関する注意

(1) TSLTZ を CHAR、DATE、TIMESTAMP および TSTZ に変換すると、値はセッションのタイム・ゾーンに調整されます。

(2) CHAR、DATE および TIMESTAMP を TSLTZ に変換すると、セッションのタイム・ゾーンはメモリー内に格納されます。

(3) TSLTZ を ANSI DATE に変換すると、TIME 部分には 0 (ゼロ) が設定されます。

(4) TSTZ を変換すると、タイム・スタンプを含むタイム・ゾーンはメモリー内に格納されます。

(5) 文字列を INTERVAL に割り当てる場合、文字列は有効な INTERVAL 文字形式である必要があります。

日時および日付のアップグレード規則

OCI では、日時列および日付列について、クライアント・アプリケーションとデータベース・サーバー間の完全な上位互換性および下位互換性を備えています。

リリース 1 (9.0.1) より前のクライアントとリリース 1 (9.0.1) 以上のサーバー

リリース 1 (9.0.1) より前のアプリケーションで使用できる日時データ型は、DATE データ型の `SQLT_DAT` のみです。バッファを `SQLT_DAT` に定義したリリース 1 (9.0.1) より前のクライアントで `TSLTZ` 列からデータを取得しようとする、値の日付部分のみがクライアントに戻されます。

リリース 1 (9.0.1) より前のサーバーとリリース 1 (9.0.1) 以上のクライアント

このケースでは、新しいクライアントに `SQLT_TIMESTAMP_LTZ` 型のバインド・バッファまたは定義バッファがある場合があります。この場合は、次の互換性が維持されます。

クライアント・アプリケーションで、`SQLT_TIMESTAMP_LTZ` (または新しい日時データ型) を `DATE` 列に挿入しようとする、データが失われる可能性があるため、エラーが発生します。

クライアントに `SQLT_TIMESTAMP_LTZ` データ型の `OUT` バインド・バッファまたは定義バッファがあり、サーバー側の基礎となる `SQL` バッファまたは列が `DATE` 型の場合は、セッションのタイム・ゾーンが割り当てられます。

型コード

Oracle の型はそれぞれ、スカラー、コレクション、参照、オブジェクト型のいずれであっても、他と重複しない型コードと関連付けられています。この型コードで型を識別します。また、この型コードはオブジェクト型属性についての情報を管理するために Oracle で使用されます。この型コード・システムは、汎用的で拡張可能に設計されており、Oracle データ型への 1 対 1 の直接マップには連結されていません。次の `SQL` 文で考えてみます。

```
CREATE TYPE my_type AS OBJECT
( attr1    NUMBER,
  attr2    INTEGER,
  attr3    SMALLINT );

CREATE TABLE my_table AS TABLE OF my_type;
```

これらの文では、オブジェクト型とオブジェクト表を作成します。作成された際に、`my_table` には 3 つの列がありますが、これらはすべて Oracle `NUMBER` 型となります。これは、`SMALLINT` と `INTEGER` が内部的に `NUMBER` にマップするためです。ただし、`my_type` の属性の内部表現は、これら 3 つの属性のデータ型の区別を次のように維持します。`attr1`

は OCI_TYPECODE_NUMBER、attr2 は OCI_TYPECODE_INTEGER、そして attr3 は OCI_TYPECODE_SMALLINT です。アプリケーションで my_type を記述する場合、これらの型コードが戻されます。

OCITypeCode は、型コードの C データ型です。型コードは、OCIObjectNew()（どの型のオブジェクトが作成されたかを判断するのに役立つ）などの一部の OCI 関数で使用されます。また、オブジェクトが記述されるとき、いくつかの属性の値としても戻されます。たとえば、OCI_ATTR_TYPECODE 属性の型の問合せを行うと、**OCITypeCode** の値が戻ります。

表 3-8 は、**OCITypeCode** に可能な値を示しています。各 Oracle データ型に対応する値があります。

表 3-8 OCITypeCode 値

値	データ型
OCI_TYPECODE_REF	REF
OCI_TYPECODE_DATE	DATE
OCI_TYPECODE_TIMESTAMP	TIMESTAMP
OCI_TYPECODE_TIMESTAMP_TZ	TIMESTAMP WITH TIME ZONE
OCI_TYPECODE_TIMESTAMP_LTZ	TIMESTAMP WITH LOCAL TIME ZONE
OCI_TYPECODE_INTERVAL_YM	INTERVAL YEAR TO MONTH
OCI_TYPECODE_INTERVAL_DS	INTERVAL DAY TO SECOND
OCI_TYPECODE_REAL	単精度実数
OCI_TYPECODE_DOUBLE	倍精度実数
OCI_TYPECODE_FLOAT	浮動小数点
OCI_TYPECODE_NUMBER	Oracle 数値
OCI_TYPECODE_DECIMAL	10 進数
OCI_TYPECODE_OCTET	8 進数
OCI_TYPECODE_INTEGER	INTEGER
OCI_TYPECODE_SMALLINT	SMALLINT
OCI_TYPECODE_RAW	RAW
OCI_TYPECODE_VARCHAR2	ANSI SQL の可変文字列、すなわち VARCHAR2
OCI_TYPECODE_VARCHAR	Oracle SQL の可変文字列、すなわち VARCHAR
OCI_TYPECODE_CHAR	SQL 内部の固定長文字列、すなわち SQL CHAR

表 3-8 OCITypeCode 値（続き）

値	データ型
OCI_TYPECODE_VARRAY	可変長配列（VARRAY）
OCI_TYPECODE_TABLE	多重集合
OCI_TYPECODE_CLOB	キャラクタ・ラージ・オブジェクト（CLOB）
OCI_TYPECODE_BLOB	バイナリ・ラージ・オブジェクト（BLOB）
OCI_TYPECODE_BFILE	バイナリ・ラージ・オブジェクト・ファイル（BFILE）
OCI_TYPECODE_OBJECT	名前付きオブジェクト型、または SYS.XMLType
OCI_TYPECODE_NAMEDCOLLECTION	ドメイン（名前付き基本型）

SQLT 値および OCI_TYPECODE 値の関係

Oracle では、2 種類のデータ型コード値のセットを認識します。これらのセットの 1 つは SQLT_ 接頭辞、もう 1 つは OCI_TYPECODE_ 接頭辞によって区別されます。

SQLT 型コードは、バインドまたは定義操作でデータ型を指定するために、OCI で使用されます。このように、SQL 型コードは、Oracle と OCI クライアント・アプリケーション間のデータ変換を制御するのに役立ちます。OCI_TYPECODE 型は、ユーザー定義型を操作または作成する際に、Oracle の型システムで事前定義済みの型を参照または記述するために使用します。

多くの場合、SQLT 値と OCI_TYPECODE 値の間で、ダイレクト・マッピングがあります。ただし、1 対 1 のダイレクト・マッピングがない場合もあります。たとえば、OCI_TYPECODE_SIGNED16、OCI_TYPECODE_SIGNED32、OCI_TYPECODE_INTEGER、OCI_TYPECODE_OCTET および OCI_TYPECODE_SMALLINT はすべて、SQLT_INT 型にマップされます。

次の表では、SQLT 型と OCI_TYPECODE 型間のマッピングを示します。

表 3-9 OCI_TYPECODE から SQLT へのマップ

Oracle 型システムの型名	Oracle 型システムの型	等価 SQLT 型
BFILE	OCI_TYPECODE_BFILE	SQLT_BFILE
BLOB	OCI_TYPECODE_BLOB	SQLT_BLOB
CHAR	OCI_TYPECODE_CHAR (n)	SQLT_AFC(n) [注意 1]
CLOB	OCI_TYPECODE_CLOB	SQLT_CLOB
COLLECTION	OCI_TYPECODE_NAMEDCOLLECTION	SQLT_NCO
DATE	OCI_TYPECODE_DATE	SQLT_DAT

表 3-9 OCI_TYPECODE から SQLT へのマップ (続き)

Oracle 型システムの型名	Oracle 型システムの型	等価 SQLT 型
TIMESTAMP	OCI_TYPECODE_TIMESTAMP	SQLT_TIMESTAMP
TIMESTAMP WITH TIME ZONE	OCI_TYPECODE_TIMESTAMP_TZ	SQLT_TIMESTAMP_TZ
TIMESTAMP WITH LOCAL TIME ZONE	OCI_TYPECODE_TIMESTAMP_LTZ	SQLT_TIMESTAMP_LTZ
INTERVAL YEAR TO MONTH	OCI_TYPECODE_INTERVAL_YM	SQLT_INTERVAL_YM
INTERVAL DAY TO SECOND	OCI_TYPECODE_INTERVAL_DS	SQLT_INTERVAL_DS
FLOAT	OCI_TYPECODE_FLOAT (b)	SQLT_FLT (8) [注意 2]
DECIMAL	OCI_TYPECODE_DECIMAL (p)	SQLT_NUM (p, 0) [注意 3]
DOUBLE	OCI_TYPECODE_DOUBLE	SQLT_FLT (8)
INTEGER	OCI_TYPECODE_INTEGER	SQLT_INT (i) [注意 4]
NUMBER	OCI_TYPECODE_NUMBER (p, s)	SQLT_NUM (p, s) [注意 5]
OCTET	OCI_TYPECODE_OCTET	SQLT_INT (1)
POINTER	OCI_TYPECODE_PTR	< なし >
RAW	OCI_TYPECODE_RAW	SQLT_LVB
REAL	OCI_TYPECODE_REAL	SQLT_FLT (4)
REF	OCI_TYPECODE_REF	SQLT_REF
OBJECT または SYS.XMLType	OCI_TYPECODE_OBJECT	SQLT_NTY
SIGNED(8)	OCI_TYPECODE_SIGNED8	SQLT_INT (1)
SIGNED(16)	OCI_TYPECODE_SIGNED16	SQLT_INT (2)
SIGNED(32)	OCI_TYPECODE_SIGNED32	SQLT_INT (4)
SMALLINT	OCI_TYPECODE_SMALLINT	SQLT_INT (i) [注意 4]
TABLE [注意 6]	OCI_TYPECODE_TABLE	< なし >
TABLE (索引付き表)	OCI_TYPECODE_ITABLE	SQLT_TAB
UNSIGNED(8)	OCI_TYPECODE_UNSIGNED8	SQLT_UIN (1)
UNSIGNED(16)	OCI_TYPECODE_UNSIGNED16	SQLT_UIN (2)
UNSIGNED(32)	OCI_TYPECODE_UNSIGNED32	SQLT_UIN (4)
VARRAY [注意 6]	OCI_TYPECODE_VARRAY	< なし >

表 3-9 OCI_TYPECODE から SQLT へのマップ (続き)

Oracle 型システムの型名	Oracle 型システムの型	等価 SQLT 型
VARCHAR	OCI_TYPECODE_VARCHAR (n)	SQLT_CHR (n) [注意 1]
VARCHAR2	OCI_TYPECODE_VARCHAR2 (n)	SQLT_VCS (n) [注意 1]

注意：

1. n はバイト単位の文字列サイズです。
2. これらは浮動小数点数で、精度は 2 進数で与えられます。b は、数値の精度を 2 進数の数字で表します。
3. これは小数点以下の数値を伴わない NUMBER と等価です。
4. i はバイト単位の数値サイズで、OCI コールの一部として設定されます。
5. p は 10 進数での数値の精度です。s は 10 進数での数値のスケールです。
6. 名前付きコレクション型の一部としてのみ可能です。

oratypes.h の定義

このマニュアルでは、全体を通して **ub2**、**sb4** などのデータ型、または **UB4MAXVAL** などの定数を参照することができます。これらの型は、**public** ディレクトリにある **oratypes.h** ヘッダー・ファイルで定義されます。正確な内容は、使用しているプラットフォームによって異なります。

注意： oratypes.h でのデータ型の使用は、OCI にパラメータを渡す手段としてサポートされています。

OCI での SQL 文の使用

この章では、Oracle Call Interface を使用した SQL 文の処理に関する概念およびステップを説明します。この章は、次の項目で構成されています。

- [SQL 文の処理の概要](#)
- [SQL 文の処理](#)
- [文の準備](#)
- [バインド](#)
- [文の実行](#)
- [選択リスト項目の記述](#)
- [定義](#)
- [結果のフェッチ](#)
- [スクロール・カーソル](#)

SQL 文の処理の概要

第2章「OCI プログラミングの基本」では、OCI アプリケーションに含まれる基本ステップについて説明しました。この章では、OCI プログラムで SQL 文を処理するときの特定の作業について、さらに詳細に説明します。

SQL 文の処理

OCI プログラムの最も一般的なタスクの 1 つは、SQL 文の受入れと処理です。この項では、SQL 処理に含まれる特定のステップの概要を説明します。

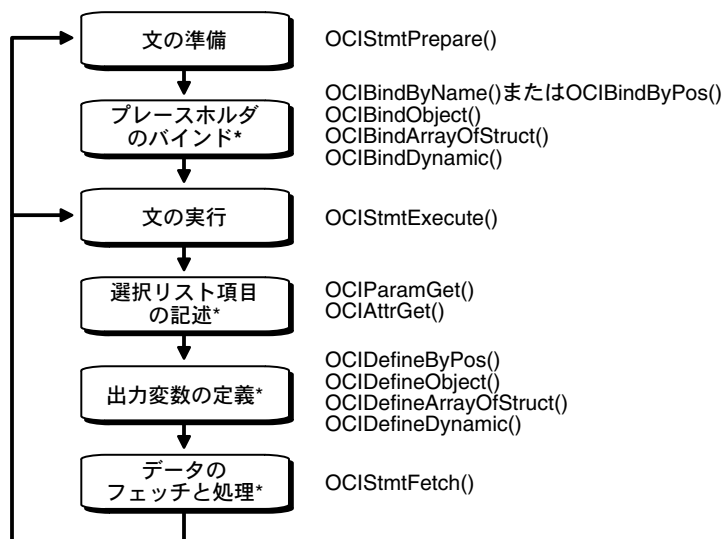
必要なハンドルを割り当ててサーバーへ接続した後は、[図 4-1「SQL 文の処理におけるステップ」](#)に示すように、次の基本ステップで SQL 文の処理を行います。

1. **準備** — `OCIStmtPrepare()` を使用してアプリケーション要求を定義します。
2. **バインド** — 入力変数を伴う DML 文および問合せの場合は、次の関数のうちいずれかを 사용하여バインド・コールを 1 つ以上実行します。
 - `OCIBindByPos()`
 - `OCIBindByName()`
 - `OCIBindObject()`
 - `OCIBindDynamic()`
 - `OCIBindArrayOfStruct()`各入力変数（または PL/SQL 出力変数）または各配列のアドレスは、文中の各プレースホルダにバインドします。
3. **実行** — `OCIStmtExecute()` をコールして文を実行します。DDL 文の場合は、これ以降のステップは必要ありません。
4. **記述** — 必要な場合は、`OCIParamGet()` および `OCIAttrGet()` を使用して、選択リスト項目を記述します。このステップはオプションです。選択リスト項目の数および各項目の属性（長さやデータ型など）がコンパイル時にわかっている場合には、このステップは必要ありません。
5. **定義** — 問合せの場合は、`OCIDefineByPos()`、`OCIDefineObject()`、`OCIDefineDynamic()` または `OCIDefineArrayOfStruct()` の定義コールを 1 つ以上実行して、SQL 文の各選択リスト項目に対して出力変数を定義します。無名 PL/SQL ブロックの出力変数の定義には定義コールを使用しないように注意してください。その定義は、データをバインドする時点ですすでに行われています。
6. **フェッチ** — 問合せの場合は、`OCIStmtFetch()` をコールして問合せの結果をフェッチします。

これらのステップの後、アプリケーションでは割り当てたハンドルを解放し、次にサーバーから接続解除するか、追加の文を処理できます。

7.x アップグレードの注意： OCI プログラムでは明示的な解析ステップが不要になりました。文の解析が必要な場合、解析のステップは実行時に行われます。これは、リリース 8.0 以上のアプリケーションでは、DML 文と DDL 文の両方に対して実行コマンドを発行する必要があることを意味します。

図 4-1 SQL 文の処理におけるステップ



* 必要に応じて実行されるステップです

この図では、各ステップについて、対応する OCI ファンクション・コールを示しています。複数のコールが必要な場合もあります。

前述のステップの詳細は、次の項を参照してください。

注意： ステップの順序には、いくつかのバリエーションがあります。たとえば、コンパイル時にデータ型と戻される値の長さがわかっている場合には、実行前に定義ステップを処理できます。また、アスタリスク (*) で示しているように、アプリケーションによってはいくつかのステップは不要です。

アプリケーションで次のことを行う場合には、図に示したステップの他に追加ステップが必要です。

- 複数トランザクションの開始および管理
- 実行の複数スレッドの管理
- ピース単位の挿入、更新またはフェッチの実行

関連項目：

- これらの項目の詳細は、[第9章「OCI プログラミングの高度なトピック」](#)を参照してください。
- OCI 共有モード機能の使用方法的詳細は、2-22 ページの「[共有データ・モード](#)」を参照してください。

文の準備

SQL 文および PL/SQL 文は、文の準備コールおよびバインド・コール（必要な場合）を使用して、実行の準備をする必要があります。このフェーズでは、アプリケーションは SQL 文または PL/SQL 文を指定し、文中の関連付けられたプレースホルダをデータにバインドして実行できるようにします。クライアント側ライブラリによって、実行準備の完了した文をメンテナンスするための記憶域が割り当てられます。

アプリケーションでは、`OCIStmtPrepare()` コールを使用し、事前に割り当てられている文ハンドルに SQL 文または PL/SQL 文を渡すことによって、文の実行準備を要求します。これは、完全なローカル・コールであるため、サーバーとのラウンドトリップは必要ありません。この時点では、文と特定のサーバーとの関連付けは行われません。

要求コールの後、アプリケーションから文ハンドルで `OCIAttrGet()` をコールし、`attrtype` パラメータへ `OCI_ATTR_STMT_TYPE` を渡すことにより、準備されている SQL 文の種類を調べることができます。可能な属性値とそれに対応する文の種類は、[表 4-1](#) にリストされています。

表 4-1 OCI_ATTR_STMT_TYPE 値および文の種類

属性値	文の種類
OCI_STMT_SELECT	SELECT 文
OCI_STMT_UPDATE	UPDATE 文
OCI_STMT_DELETE	DELETE 文
OCI_STMT_INSERT	INSERT 文
OCI_STMT_CREATE	CREATE 文
OCI_STMT_DROP	DROP 文
OCI_STMT_ALTER	ALTER 文
OCI_STMT_BEGIN	BEGIN... (PL/SQL)
OCI_STMT_DECLARE	DECLARE... (PL/SQL)

関連項目：

- OCI アプリケーションでの PL/SQL 使用方法の詳細は、2-44 ページの「[OCI プログラムでの PL/SQL 使用](#)」を参照してください。
- [OCIStmtPrepare\(\)](#) コールを参照してください。

プリコンパイルされた SQL 文を複数のサーバーで使用方法

文ハンドルとサーバーの個々のサービス・コンテキスト・ハンドルの再関連付けによって、準備完了アプリケーション要求を実行時に複数のサーバーに対して実行できます。新たな関連付けが行われると、現行のサービス・コンテキスト・ハンドルと文ハンドル間の関連付けについてキャッシュされている情報はすべて失われます。

たとえば、複数のサーバーを管理するネットワーク・マネージャのようなアプリケーションを考えてみます。多くの場合、情報を取り出して表示するには、同一の SELECT 文を複数のサーバーに対して実行する必要があります。OCI によって、ネットワーク・マネージャ・アプリケーションでは、一度準備した SELECT 文を複数のサーバーに対して実行できます。プリコンパイルされた SQL 文を次のサーバーに再度関連付けする前に、各サーバーから必要なすべての行をフェッチしておく必要があります。

注意： プリコンパイルされた SQL 文を同じサーバーで頻繁に再実行する必要がある場合は、別のサービス・コンテキスト用に新しい文を準備すると効率的です。

バインド

大部分の DML 文と一部の問合せ（WHERE 句を使用した問合せなど）では、プログラムでデータを SQL または PL/SQL 文の一部として Oracle に渡す必要があります。このようなデータは、プログラムのコンパイル時にわかっている定数またはリテラル・データです。たとえば、次の SQL 文は従業員をデータベースへ追加するものですが、これには 'BESTRY' および 2365 などのリテラルがいくつか含まれています。

```
INSERT INTO emp VALUES
    (2365, 'BESTRY', 'PROGRAMMER', 2000, 20)
```

このように文をアプリケーション内にコーディングすると、その使用範囲が非常に限定されます。データベースに新しい従業員を追加するたびに、文を変更してプログラムを再コンパイルする必要があります。ユーザーが実行時に入力データを指定できるようにすれば、プログラムにもっと柔軟性を持たせることができます。

入力データが実行時に指定される SQL 文または PL/SQL ブロックを準備する場合は、SQL 文または PL/SQL ブロックのプレースホルダによって、どの位置にデータが入力されるのかをマークします。たとえば、次の SQL 文には、5 つのプレースホルダが含まれています。プレースホルダは先頭にコロンが付き（:ename など）、プログラムにより入力データが供給される必要があることを示しています。

```
INSERT INTO emp VALUES
    (:empno, :ename, :job, :sal, :deptno)
```

入力変数のプレースホルダは、DELETE 文、INSERT 文、SELECT 文、UPDATE 文または PL/SQL ブロックで、式またはリテラル値を使用できる位置であれば文中の任意の位置で使用できます。PL/SQL では、出力変数にもプレースホルダを使用できます。

プレースホルダを、表などの他の Oracle オブジェクトを表すために使用することはできません。たとえば、emp プレースホルダを次のように使用すると無効になります。

```
INSERT INTO :emp VALUES
    (12345, 'OERTEL', 'WRITER', 50000, 30)
```

SQL 文または PL/SQL ブロックの各プレースホルダに対して、プログラム内の変数のアドレスをプレースホルダにバインドする OCI ルーチンをコールする必要があります。Oracle では、文を実行すると、プログラムで入力変数またはバインド変数に入れたデータが読み取られ、それが SQL 文でサーバーに渡されます。

関連項目： バインド操作の実装の詳細は、[第 5 章「バインドと定義」](#)を参照してください。

文の実行

OCI アプリケーションは、`OCIStmtExecute()` を使用して、プリコンパイルされた SQL 文を個別に実行します。

関連項目： 構文の詳細は、[OCIStmtExecute\(\)](#) を参照してください。

OCI アプリケーションで問合せを実行すると、問合せ指定に一致するデータを Oracle から受け取ります。データベース内では、データは Oracle 独自の形式で格納されています。戻された結果に対して、OCI アプリケーションでは、データを特定のホスト言語形式に変換し、それを特定の出力変数またはバッファ内に格納することを要求できます。

問合せの各選択リスト項目に対し、問合せの結果を受け取るため、OCI アプリケーションで出力変数を定義してください。定義ステップでは、バッファのアドレスおよび取り出すデータの型を指示します。

注意： `OCIStmtExecute()` コール前に出力変数が `SELECT` 文に対して定義されている場合は、`iters` パラメータで指定した行数が定義済みの出力バッファに直接フェッチされ、プリフェッチ・カウントと同じ数の追加行がプリフェッチされます。追加行がない場合、フェッチは `OCIStmtFetch()` をコールしないで完了します。

問合せ以外の場合、配列操作中の文の実行回数は `iters - rowoff` と等しくなります。`rowoff` はバインド済み配列のオフセットであり、`OCIStmtExecute()` コールのパラメータでもあります。たとえば、10 項目の配列が `INSERT` 文の 1 個のプレースホルダにバインドされていて、`iters` が 10 と設定されている場合は、`rowoff` が 0 (ゼロ) であれば、10 項目すべてが 1 回の実行コールで挿入されます。`rowoff` に 2 が設定されていれば、挿入されるのは 8 項目のみです。

関連項目： 出力変数の定義の詳細は、4-15 ページの「[定義](#)」を参照してください。

実行スナップショット

`OCIStmtExecute()` コールを使用すると、データベースのコミット済みデータの一貫した同スナップショット上で、複数のサービス・コンテキストが確実に動作します。これは、特定の `OCIStmtExecute()` コールの `snap_out` パラメータの内容を読み取り、その値を次の `OCIStmtExecute()` コールの `snap_in` パラメータに渡すことによって実現されます。

注意： 同じスナップショットを使用している場合でも、1つのサービス・コンテキスト内でコミットされていないデータは、他のコンテキストでは認識されません。

snap_out パラメータおよび *snap_in* パラメータのデータ型は、どちらも OCI スナップショット記述子のデータ型 **OCISnapshot** です。この記述子は `OCIDescAlloc()` 関数で割り当てられます。

関連項目： 記述子の詳細は、2-15 ページの「[記述子](#)」を参照してください。

`OCIStmtExecute()` をコールする場合は、スナップショットの指定は不要です。次のサンプル・コードは、スナップショット・パラメータが `NULL` として渡される基本的な実行例です。

```
checkerr(errhp, OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,  
                               (OCISnapshot *)NULL, (OCISnapshot *) NULL, OCI_DEFAULT))
```

注意： `checkerr()` 関数によって、OCI アプリケーションからのリターン・コードが評価されます。この関数用のコードは、2-31 ページの「[エラー処理](#)」にリストされています。

実行モード

`OCIStmtExecute()` コールのために、次のようないくつかのモードを指定できます。

- **OCI_DEFAULT** — このモードで `OCIStmtExecute()` をコールすると、文が実行されます。また、選択リストに関する記述情報が暗黙的に戻されます。
- **OCI_DESCRIBE_ONLY** — このモードは、実行の前に問合せを記述するユーザー用です。`OCIStmtExecute()` をこのモードでコールすると、文は実行されませんが、選択リスト記述は戻されます。
- **OCI_COMMIT_ON_SUCCESS** — このモードで文を実行すると、実行が正常終了した場合は、実行後にカレント・トランザクションがコミットされます。
- **OCI_EXACT_FETCH** — アプリケーションでフェッチされる行数があらかじめ正確にわかっているときに使用します。
- **OCI_BATCH_ERRORS** — このモードの詳細は、4-9 ページの「[OCIStmtExecute\(\) 用のバッチ・エラー・モード](#)」を参照してください。

OCIStmtExecute() 用のバッチ・エラー・モード

OCI では、配列 DML 操作を実行できます。たとえば、アプリケーションでは、INSERT 文、UPDATE 文または DELETE 文の配列を 1 回の文実行で処理できます。一意性制約違反などのサーバーからのエラーのために操作のいずれかが失敗すると、配列操作は中止され、OCI はエラーを戻します。その場合、配列内の残りの行はすべて無視されます。このため、アプリケーションでは残りの配列を再実行する必要があります。さらにエラーが発生した場合は、この処理全体を再度繰り返す必要があります。これによって追加のラウンドトリップが発生します。

配列 DML 操作を容易に処理するために、OCI ではバッチ・エラー・モード（拡張 DML 配列機能とも呼ばれます）が用意されています。このモードは OCIStmtExecute() コールに指定するもので、1 つ以上のエラーが発生する場合の DML 配列処理を単純化します。このモードでは、OCI はすべての行の INSERT 文、UPDATE 文または DELETE 文を試行し、発生したエラーに関する情報を収集（バッチ処理）します。次に、アプリケーションでこのエラー情報を取り出し、最初のコールで失敗した任意の DML 操作を再実行できます。

注意： この機能は、リリース 8.1 以上のサーバーに対して実行中のリリース 8.1 以上の OCI ライブラリにリンクされているアプリケーションでのみ使用可能です。また、この項で説明している新しいプログラム・ロジックを反映するために、アプリケーションをコーディングしなおす必要があります。

この方法では、配列内のすべての DML 操作が最初のコールで試行され、失敗した操作を 2 番目のコールで再発行できます。

このモードは次のように使用します。

1. OCIStmtExecute() コールの mode パラメータとして OCI_BATCH_ERRORS を指定します。
2. OCIStmtExecute() で配列 DML 操作を実行した後、アプリケーションでは文ハンドルで OCIAttrGet() をコールして OCI_ATTR_NUM_DML_ERRORS 属性を取り出すことにより、操作中に発生したエラーの番号を取り出すことができます。次に例を示します。

```
ub4    num_errs;  
OCIAttrGet(stmt, OCI_HTYPE_STMT, &num_err, 0, OCI_ATTR_NUM_DML_ERRORS, errhp);
```

3. エラーのリストが取り出されると、エラー・ハンドルが解放されます。

アプリケーションでは、OCIParamGet() を使用して、OCISstmtExecute() コールに渡されたエラー・ハンドルから各エラーを行情報とともに抽出します。この情報を取り出すには、アプリケーションで OCIParamGet() コール用の追加の新しいエラー・ハンドルを割り当てる必要があります。この新しいエラー・ハンドルには、バッチとして記録されたエラー情報が含まれています。アプリケーションでは、OCIErrorGet() で各エラーの構文を取得し、新しいエラー・ハンドルで OCIAttrGet() をコールすることにより、エラーが発生した行のオフセットを（DML 配列内に）取得します。

たとえば、num_errs の結果が取り出された後で、アプリケーションから次のコールを発行する可能性があります。

```
...
OCIError errhndl;
for (i=0; i<num_errs; i++)
{
    OCIParamGet(errhp, OCI_HTYPE_ERROR, errhp, &errhndl, i);
    OCIAttrGet(errhndl, OCI_HTYPE_ERROR, &row_offset, 0,
        OCI_ATTR_DML_ROW_OFFSET, errhp);
    OCIErrorGet(..., errhndl, ...);
}
...
```

次に、アプリケーションではバッチとして記録されたエラーから取り出された診断情報を使用し、配列内の該当するエントリのバインド情報を修正できます。該当するバインド・バッファが修正または更新されると、アプリケーションでそれに対応する DML 文を再実行できます。

アプリケーションでは、最初の実行でエラーの原因となる行をコンパイル時に検出できないため、後続の DML のバインドは、実行時に該当するバッファを渡すことにより、動的に実行する必要があります。ユーザーは、最初の DML 操作での配列バインドで使われたバインド・バッファを再利用できます。

バッチ・エラー・モードの例

次のコードは、この実行モードの使用法の例を示しています。この例では、表に 5 つの行を（NUMBER および CHAR 型の 2 つの列とともに）挿入するアプリケーションがあるものとします。さらに、2 つの行（1 および 3）のみが最初の DML 操作で正常に挿入されるものとします。ユーザーは次に、データ（最初の操作で挿入された誤ったデータ）を修正し、修正したデータの UPDATE を発行します。ユーザーは文ハンドル stmt1 および stmt2 を使用して、INSERT および UPDATE をそれぞれ発行します。


```

...
OCIBind *bindp1[2], *bindp2[2];
ub4 num_errs, row_off[MAXROWS], number[MAXROWS] = {1,2,3,4,5};
char grade[MAXROWS] = {'A','B','C','D','E'};
...
/* Array bind all the positions */
OCIBindByPos (stmtpl,&bindp1[0],errhp,1,(dvoid *)&number[0],
              sizeof(number[0]),SQLT_NUM,(dvoid *)0, (ub2 *)0,(ub2 *)0,
              0,(ub4 *)0,OCI_DEFAULT);
OCIBindByPos (stmtpl,&bindp1[1],errhp,2,(dvoid *)&grade[0],
              sizeof(grade[0]),SQLT_CHR,(dvoid *)0, (ub2 *)0,(ub2 *)0,0,0,
              (ub4 *)0,OCI_DEFAULT);
/* execute the array INSERT */
OCIStmtExecute (svchp,stmtpl,errhp,5,0,0,0,OCI_BATCH_ERRORS);
/* get the number of errors */
OCIAttrGet (stmtpl, OCI_HTYPE_STMT, &num_errs, 0,
            OCI_ATTR_NUM_DML_ERRORS, errhp);
if (num_errs) {
    /* The user can do one of two things: 1) Allocate as many */
    /* error handles as number of errors and free all handles */
    /* at a later time; or 2) Allocate one err handle and reuse */
    /* the same handle for all the errors */
    OCIError *errhndl[num_errs];
    for (i = 0; i < num_errs; i++) {
        OCIParamGet(errhp, OCI_HTYPE_ERROR, errhp, &errhndl[i], i);
        OCIAttrGet (errhndl[i], OCI_HTYPE_ERROR, &row_off[i], 0,
                    OCI_ATTR_DML_ROW_OFFSET, errhp);
        /* get server diagnostics */
        OCIErrorGet (... , errhndl[i], ...);
    }
}
/* make corrections to bind data */
OCIBindByPos (stmtpl2,&bindp2[0],errhp,1,(dvoid *)0,0,SQLT_NUM,
              (dvoid *)0, (ub2 *)0,(ub2 *)0,0,(ub4 *)0,OCI_DATA_AT_EXEC);
OCIBindByPos (stmtpl2,&bindp2[1],errhp,2,(dvoid *)0,0,SQLT_DAT,
              (dvoid *)0, (ub2 *)0,(ub2 *)0,0,(ub4 *)0,OCI_DATA_AT_EXEC);
/* register the callback for each bind handle */
OCIBindDynamic (bindp2[0],errhp,row_off,my_callback,0,0);
OCIBindDynamic (bindp2[1],errhp,row_off,my_callback,0,0);
/* execute the UPDATE statement */
OCIStmtExecute (svchp,stmtpl2,errhp,2,0,0,0,OCI_BATCH_ERRORS);
...

```

この例では、どの行がエラーとともに戻されるかコンパイル時にはわからないため、`OCIBindDynamic()` をコールバックとともに使用しています。コールバックでは、`row_off` に格納されたエラーを含む行の番号をコールバック・コンテキストを通じて渡し、更新または修正の必要がある行のみを送信できます。`INSERT` の実行と `UPDATE` の実行では、同じバインド・バッファを共有できます。

選択リスト項目の記述

OCI アプリケーションで問合せを処理する場合には、選択リスト項目に関して詳細な情報を取得する必要があります。実行時まで問合せの内容がわからない動的問合せの場合は、特にそうです。そのような場合、プログラムでは、選択リスト項目のデータ型および列の長さに関する情報を取得する必要があります。この情報は、問合せの結果を受け取る出力変数を定義するために必要です。

たとえば、ユーザーが次の問合せを入力したとします。

```
SELECT * FROM employees
```

プログラムには、`employees` 表の列についての事前情報はありません。

使用可能な記述の種類には、暗黙的および明示的の 2 つがあります。暗黙的な記述は、サーバーから記述情報を取得するのに特別なコールを必要としない記述です。ただし、その情報にアクセスするには複数の特別なコールが必要です。明示的な記述は、サーバーから記述情報を取得するために、アプリケーションから特定の関数をコールする必要がある記述です。

アプリケーションでは、選択リスト（問合せ）を暗黙的にも明示的にも記述できます。その他のスキーマ要素は、明示的に記述する必要があります。

暗黙的な記述の場合、アプリケーションでは、文の実行が終了した後文ハンドルの属性として選択リストの情報を取得できます。特定の記述コールは必要ありません。記述コールが不要なため、これは暗黙的と呼ばれます。記述情報は、実行時に自動的に供給されます。

問合せは、実行前に明示的に記述できます。そのためには、`OCIStmtExecute()` のモードとして `OCI_DESCRIBE_ONLY` を指定します。`OCIStmtExecute()` をこのモードでコールすると、文は実行されませんが、選択リスト記述は戻されます。ただし、パフォーマンス向上のためには、標準的な文の実行で自動的に得られる暗黙的記述をアプリケーションで活用することをお勧めします。

`OCIDescribeAny()` コールを使用した明示的な記述では、選択リストではなく、スキーマ・オブジェクトに関する情報を取得します。

すべての場合において、列およびデータ型に関する特定の情報は、ハンドル属性を読み込むことによって検索されます。

関連項目： `OCIDescribeAny()` を使用して、スキーマ・オブジェクトに関するメタデータを取得する方法の詳細は、[第 6 章「スキーマ・メタデータの記述」](#) を参照してください。

暗黙的記述

SQL 文を実行した後、文ハンドルの属性として選択リストに関する情報を取得できます。明示的な記述コールは必要ありません。

アプリケーションで文ハンドルから選択リストに関する情報を取り出すには、選択リストの各位置に対して `OCIParamGet()` を 1 回ずつコールし、その位置のパラメータ記述子を割り当てる必要があります。選択リスト位置は 1 が基準となります。これは選択リストの最初の項目が位置番号 1 であることを意味しています。

複数の選択リストに関する情報を取り出すには、最初に `pos` パラメータを 1 と設定した `OCIParamGet()` をコールし、次に `pos` の値を反復して、ORA-24334 の `OCI_ERROR` が戻されるまで `OCIParamGet()` コールを繰り返します。また、アプリケーションでは、列をランダムに取得するために、位置 `n` を指定できます。

アプリケーションでは、選択リストの位置のパラメータ記述子を割り当てた後、パラメータ記述子について `OCIAttrGet()` をコールすることによって、特定の情報を取り出せます。パラメータ記述子から取得できる情報として、データ型とパラメータの最大サイズがあります。

次のコード例では、問合せ実行に続く問合せに対応した列名やデータ型を取り出すループを示しています。この問合せは、事前に `OCIStmtPrepare()` のコールによって文ハンドルと関連付けられています。

```
OCIParam      *mypard;
ub4           counter;
ub2           dtype;
text          *col_name;
ub4           col_name_len;
sb4           parm_status;
...
/* Request a parameter descriptor for position 1 in the select-list */
counter = 1;
parm_status = OCIParamGet(stmthp, OCI_HTYPE_STMT, errhp, &mypard,
                          (ub4) counter);

/* Loop only if a descriptor was successfully retrieved for
   current position, starting at 1 */
while (parm_status==OCI_SUCCESS) {
/* Retrieve the data type attribute */
checkerr(errhp, OCIAttrGet((dvoid*) mypard, (ub4) OCI_DTYPE_PARAM,
                          (dvoid*) &dtype, (ub4 *) 0, (ub4) OCI_ATTR_DATA_TYPE,
                          (OCIError *) errhp ));
/* Retrieve the column name attribute */
checkerr(errhp, OCIAttrGet((dvoid*) mypard, (ub4) OCI_DTYPE_PARAM,
                          (dvoid**) &col_name, (ub4 *) &col_name_len, (ub4) OCI_ATTR_NAME,
                          (OCIError *) errhp ));
printf("column=%s datatype=%d\n\n", col_name, dtype);
fflush(stdout);
/* increment counter and get next descriptor, if there is one */
```

```
counter++;
parm_status = OCIParmGet(stmthp, OCI_HTYPE_STMT, errhp, &mypard,
                          (ub4) counter);
}
```

注意： 最初の OCIParmGet() コールに関するエラー処理は、この例に含まれていません。省略記号 (...) は、この例で部分的にコードを省略した箇所を示しています。

checkerr() 関数は、エラー処理に使用します。完全なリストは、[付録 B「OCI デモ・プログラム」](#)の最初のサンプル・アプリケーションを参照してください。

OCIAttrGet() および OCIParmGet() のコールは、ネットワーク・ラウンドトリップを必要としないローカル・コールです。これは、文の実行後すべての選択リスト情報がクライアント側にキャッシュされるためです。

関連項目：

- [OCIParmGet\(\)](#) および [OCIAttrGet\(\)](#) の説明を参照してください。
- [OCIAttrGet\(\)](#) で読み込むことができるパラメータ記述子の特定の属性のリストは、6-5 ページの「[パラメータ属性](#)」を参照してください。

問合せの明示的記述

問合せは、実行前に明示的に記述できます。そのためには、OCIStmtExecute() のモードとして OCI_DESCRIBE_ONLY を指定します。OCIStmtExecute() をこのモードでコールすると、文は実行されませんが、選択リスト記述は戻されます。

注意： パフォーマンスを最大にするために、アプリケーションではデフォルト・モードで文を実行し、実行に伴う暗黙的な記述を使用することをお勧めします。

次の短い例では、このメカニズムを使用して、選択リストの明示的な記述を実行し、選択リストの列に関する情報を戻しています。この疑似コードでは、列情報（たとえばデータ型）を取り出す方法を示します。

```

/* initialize svchp, stmhp, errhp, rowoff, iters, snap_in, snap_out */
/* set the execution mode to OCI_DESCRIBE_ONLY. Note that setting the mode to
OCI_DEFAULT does an implicit describe of the statement in addition to executing the
statement */

OCIParam *colhd; /* column handle */
checkerr(errhp, OCISStmtExecute(svchp, stmhp, errhp, iters, rowoff,
                               snap_in, snap_out, OCI_DESCRIBE_ONLY);

/* Get the number of columns in the query */
checkerr(errhp, OCIAttrGet(stmhp, OCI_HTYPE_STMT, &numcols,
                           0, OCI_ATTR_PARAM_COUNT, errh));

/* go through the column list and retrieve the data type of each column. We start
from pos = 1 */
for (i = 1; i <= numcols; i++)
{
    /* get parameter for column i */
    checkerr(errhp, OCIParamGet(stmhp, OCI_HTYPE_STMT, errh, &colhd, i));

    /* get data-type of column i */
    checkerr(errhp, OCIAttrGet(colhd, OCI_DTYPE_PARAM,
                               &type[i-1], 0, OCI_ATTR_DATA_TYPE, errh));
}

```

定義

問合せ文は、データベースのデータをアプリケーションに戻します。問合せを処理するときには、データを検索する選択リストの各項目について、出力変数または出力変数の配列を定義する必要があります。定義ステップでは、戻される結果の格納場所と格納形式を決定する関連付けを行います。

たとえば、OCI 文で次の文を処理するとします。

```

SELECT name, ssn FROM employees
      WHERE empno = :empnum

```

この場合、通常は 2 つの出力変数を定義する必要があります。1 つは、name 列から戻される値を受け取るための変数、もう 1 つは ssn 列から戻される値を受け取るための変数です。

関連項目： 定義操作の実装の詳細は、[第 5 章「バインドと定義」](#)を参照してください。

結果のフェッチ

OCI アプリケーションで問合せを処理した場合は通常、文の実行後に `OCIStmtFetch()` または `OCIStmtFetch2()` を使用して、結果をフェッチする必要があります。**スクロール・カーソル**をサポートし、かつ機能拡張される予定の `OCIStmtFetch2()` の使用をお勧めします。

関連項目： スクロール・カーソルの詳細は、4-17 ページの「[スクロール・カーソル](#)」を参照してください。

フェッチされたデータは、定義操作で指定した出力変数に格納されます。

注意： `OCIStmtExecute()` のコール前に `SELECT` 文に出力変数を定義した場合は、`iters` パラメータで指定した行数が、定義されている出力バッファに直接フェッチされます。

関連項目：

- これらの文により、5-19 ページの「[定義に使用するステップ](#)」のコード例に関連したデータがフェッチされます。詳細は、そのコード例を参照してください。
- 出力変数の定義の詳細は、5-18 ページの「[定義](#)」を参照してください。

LOB データのフェッチ

LOB 列または属性が選択リストの一部である場合、その LOB 列または属性は、ユーザーの定義方法に従って、LOB ロケータまたは実際の LOB 値として戻すことができます。LOB ロケータがフェッチされると、アプリケーションでは、`OCILOB*` インタフェースを介してそのロケータに対してさらに操作を実行できます。

関連項目： OCI における LOB ロケータの処理の詳細は、[第 7 章「LOB と FILE の操作」](#)を参照してください。

プリフェッチ・カウントの設定

サーバー・ラウンドトリップを最小限にし、アプリケーションのパフォーマンスを最大化するために、OCI では問合せの実行時に結果セット行をプリフェッチできます。OCI のプログラマは、OCIAttrSet() 関数を使用して、文ハンドルの OCI_ATTR_PREFETCH_ROWS 属性または OCI_ATTR_PREFETCH_MEMORY 属性を設定することにより、このプリフェッチをカスタマイズできます。属性は次のように使用します。

- OCI_ATTR_PREFETCH_ROWS は、プリフェッチする行数を設定します。
- OCI_ATTR_PREFETCH_MEMORY は、プリフェッチする行に割り当てられたメモリを設定します。その後、アプリケーションではメモリの容量が許すかぎり、行数をフェッチします。

これらの属性を設定すると、OCI では、OCI_ATTR_PREFETCH_MEMORY で設定した最大メモリを超えないかぎり、OCI_ATTR_PREFETCH_ROWS で設定した行数まで行をプリフェッチします。この場合 OCI は、OCI_ATTR_PREFETCH_MEMORY のバッファ・サイズに適応した行数を戻します。

デフォルトでは、プリフェッチはオンになっており、OCI では、常に 1 行余分にフェッチします。プリフェッチをオフにするには、OCI_ATTR_PREFETCH_ROWS 属性および OCI_ATTR_PREFETCH_MEMORY 属性の両方に 0 (ゼロ) を設定します。

注意： LONG 列が問合せに含まれている場合、プリフェッチは無効になります。LOB 列を含む問合せはプリフェッチできます。これは、問合せによってデータではなく LOB ロケータが戻されるためです。

関連項目： これらのハンドル属性の詳細は、A-27 ページの「[文ハンドル属性](#)」を参照してください。

スクロール・カーソル

カーソルは、データベース問合せとその結果セットです。カーソルを実行すると、問合せの結果は結果セットと呼ばれる行セットに格納されます。結果セットは、連続的または非連続的にフェッチできます。非連続のフェッチは、スクロール・カーソルと呼ばれます。

スクロール・カーソルは、指定した位置から結果セットへの前後方向へのアクセスをサポートします。このアクセスには、結果セットに対する絶対的または相対的な行番号のオフセットが使用されます。

行番号は 1 から始まります。スクロール・カーソルの場合は、以前にフェッチした行、結果セットの n 番目の行、あるいは現在の位置から n 番目の行をフェッチできます。クライアント側で結果セットの一部または全部をキャッシュすると、サーバーへのコールが減るためパフォーマンスが向上します。

Oracle では、スクロール・カーソルでの DML 操作はサポートしていません。選択リストに LONG データ型が含まれている場合、カーソルはスクロール可能にできません。

さらに、スクロール可能な文ハンドルからのフェッチは、実行時のスナップショットに基づいています。クライアント・キャッシュのサイズは、既存の OCI 属性である OCI_ATTR_PREFETCH_ROWS と OCI_ATTR_PREFETCH_MEMORY によって制御できます。

注意： スクロール・カーソルは、多くのサーバー・リソースを使用し、スクロール不可カーソルより応答時間が長くなる場合があるため、この機能を必要としないかぎり使用しないでください。

OCI でのスクロール・カーソルのサポート

OCIStmtExecute() コールには、スクロール・カーソル用の実行モードである OCI_STMT_SCROLLABLE_READONLY が用意されています。文ハンドルのデフォルトはスクロール不可です。つまり、前方への順次アクセスのみ（モードは OCI_FETCH_NEXT）です。この実行モードは、文ハンドルを実行するたびに設定する必要があります。

関連項目： 詳細は、OCIStmtExecute() を参照してください。

文ハンドル属性の OCI_ATTR_CURRENT_POSITION は、OCIAttrGet() を使用してのみ取得できます。この属性をアプリケーションで設定することはできません。この属性は、結果セット内の現在の位置を示します。

スクロール不可カーソルの場合、OCI_ATTR_ROW_COUNT は、この文ハンドルの実行後に発行された OCIStmtFetch2() コールでユーザー・バッファにフェッチされた行の合計数です。スクロール不可カーソルは前方への順次アクセスのみであるため、この属性は、アプリケーションで参照できる最大行数を表します。

スクロール・カーソルの場合、OCI_ATTR_ROW_COUNT は、ユーザー・バッファにフェッチされた行の最大（絶対）数を表します。アプリケーションでは、任意の位置でフェッチを行うことができるため、この属性は、（スクロール可能な）文の実行後にユーザーのバッファにフェッチされた行の合計数である必要はありません。

文ハンドルの OCI_ATTR_ROWS_FETCHED 属性は、前回のフェッチ・コールまたは実行で、ユーザーのバッファに正常にフェッチされた行数を表します。この属性は、スクロール・カーソルとスクロール不可カーソルの両方で使用されます。

OCIStmtFetch2() コールを使用してください。このコールは、下位互換性を保持するための OCIStmtFetch() コールを置き換えるコールです。スクロール・カーソルを使用していない場合でも、すべての新しいアプリケーションでは、OCIStmtFetch2() の使用をお勧めします。このコールはスクロール不可カーソルでも機能しますが、OCI_DEFAULT または OCI_FETCH_NEXT 以外の方向が渡されるとエラーが発生します。

スクロール・カーソルのパフォーマンスの向上

OCI のクライアント側プリフェッチ・バッファを使用すると、応答時間が短縮します。

スクロール・カーソルに対して `OCIStmtExecute()` をコールした後、`OCI_FETCH_LAST` を使用して `OCIStmtFetch2()` をコールし、結果セットのサイズを取得します。次に、`OCI_ATTR_PREFETCH_ROWS` をそのサイズの約 20% に設定し、結果セットで大量のメモリを使用している場合は `OCI_PREFETCH_MEMORY` を設定します。

スクロール・カーソル使用の制限

フェイルオーバーは、スクロール・カーソルでは機能しません。

スクロール・カーソルでは、リモートでマップされた問合せを使用できません。

関連項目：

- スクロール・カーソルの使用方法については、[OCIStmtFetch2\(\)](#) を参照してください。
- 4-17 ページ「[プリフェッチ・カウントの設定](#)」

スクロール・カーソルでのアクセスの例

次の SQL 問合せによって結果セットが戻されるとします。

```
SELECT empno, ename FROM emp
```

表 EMP の行数は 14 行です。次に、スクロール・カーソルの使用方法の 1 例を示します。

```
...
/* execute the scrollable cursor in the scrollable mode */
OCIStmtExecute(svchp, stmthp, errhp, (ub4) 0, (ub4)
               0, (CONST OCISnapshot *)NULL, (OCISnapshot *) NULL,
               OCI_STMT_SCROLLABLE_READONLY );

/* Fetches rows with absolute row numbers 6, 7, 8. After this call,
   OCI_ATTR_CURRENT_POSITION = 8, OCI_ATTR_ROW_COUNT = 8 */
checkprint(errhp, OCIStmtFetch2(stmthp, errhp, (ub4) 3,
                               OCI_FETCH_ABSOLUTE, (sb4) 6, OCI_DEFAULT);

/* Fetches rows with absolute row numbers 6, 7, 8. After this call,
   OCI_ATTR_CURRENT_POSITION = 8, OCI_ATTR_ROW_COUNT = 8 */
checkprint(errhp, OCIStmtFetch2(stmthp, errhp, (ub4) 3,
                               OCI_FETCH_RELATIVE, (sb4) -2, OCI_DEFAULT);
```

```
/* Fetches rows with absolute row numbers 14. After this call,
   OCI_ATTR_CURRENT_POSITION = 14, OCI_ATTR_ROW_COUNT = 14 */
checkprint(errhp, OCISstmtFetch2(stmthp, errhp, (ub4) 1,
                                OCI_FETCH_LAST, (sb4) 0, OCI_DEFAULT));

/* Fetches rows with absolute row number 1. After this call,
   OCI_ATTR_CURRENT_POSITION = 1, OCI_ATTR_ROW_COUNT = 14 */
checkprint(errhp, OCISstmtFetch2(stmthp, errhp, (ub4) 1,
                                OCI_FETCH_FIRST, (sb4) 0, OCI_DEFAULT));

/* Fetches rows with absolute row numbers 2, 3, 4. After this call,
   OCI_ATTR_CURRENT_POSITION = 4, OCI_ATTR_ROW_COUNT = 14 */
checkprint(errhp, OCISstmtFetch2(stmthp, errhp, (ub4) 3,
                                OCI_FETCH_NEXT, (sb4) 0, OCI_DEFAULT));

/* Fetches rows with absolute row numbers 3,4,5,6,7. After this call,
   OCI_ATTR_CURRENT_POSITION = 7, OCI_ATTR_ROW_COUNT = 14. It is assumed
   the user's define memory is allocated. */
checkprint(errhp, OCISstmtFetch2(stmthp, errhp, (ub4) 5,
                                OCI_FETCH_PRIOR, (sb4) 0, OCI_DEFAULT));

...
}
checkprint (errhp, status)
{
    ub4 rows_fetched;
    checkerr (errhp, status);
    checkerr(errhp, OCIAttrGet((CONST void *) stmthp, OCI_HTYPE_STMT,
                             (void *) &rows_fetched, (uint *) 0, OCI_ATTR_ROWS_FETCHED, errhp));
}
...
```

バインドと定義

この章では、第2章「OCIプログラミングの基本」で紹介したバインドと定義の基本概念をもう一度取り上げ、OCIアプリケーションで使える様々な種類のバインドと定義を詳しく説明します。さらに、構造体配列の使用法、およびバインド、定義、文字変換における諸問題についても説明します。

この章は、次の項目で構成されています。

- バインド
- 拡張バインド操作
- 定義
- 拡張定義操作
- 構造体配列のバインドと定義
- RETURNING 句を使用した DML
- バインドおよび定義における文字変換の問題
- PL/SQL REF CURSOR および NESTED TABLE
- ランタイム・データ割当てとピース単位操作

バインド

大部分の DML 文と一部の問合せ（WHERE 句を使用した問合せなど）では、プログラムでデータを SQL または PL/SQL 文の一部として Oracle に渡す必要があります。このようなデータは、プログラムのコンパイル時にわかっている定数またはリテラル・データです。たとえば、次の SQL 文は従業員をデータベースへ追加するものですが、これには 'BESTRY' および 2365 などのリテラルがいくつか含まれています。

```
INSERT INTO emp VALUES
    (2365, 'BESTRY', 'PROGRAMMER', 2000, 20)
```

このように文をアプリケーション内にコーディングすると、その使用範囲が非常に限定されます。データベースに新しい従業員を追加するたびに、文を変更してプログラムを再コンパイルする必要があります。ユーザーが実行時に入力データを指定できるようにすれば、プログラムにもっと柔軟性を持たせることができます。

入力データが実行時に指定される SQL 文または PL/SQL ブロックを準備する場合は、SQL 文または PL/SQL ブロックのプレースホルダによって、どの位置にデータが入力されるのかをマークします。たとえば、次の SQL 文には、5 つのプレースホルダが含まれています。プレースホルダは先頭にコロンが付き（:ename など）、プログラムにより入力データが供給される必要があることを示しています。

```
INSERT INTO emp VALUES
    (:empno, :ename, :job, :sal, :deptno)
```

入力変数のプレースホルダは、DELETE 文、INSERT 文、SELECT 文、UPDATE 文または PL/SQL ブロックで、式またはリテラル値を使用できる位置であれば文中の任意の位置で使用できます。PL/SQL では、出力変数にもプレースホルダを使用できます。

注意： プレースホルダを使用して表や列などの他の Oracle オブジェクトの名前は指定できません。

SQL 文または PL/SQL ブロックの各プレースホルダに対して、プログラム内の変数のアドレスをプレースホルダにバインドする OCI ルーチンをコールする必要があります。Oracle では、文を実行すると、プログラムで入力変数またはバインド変数に入れたデータが読み取られ、それが SQL 文でサーバーに渡されます。バインド・ステップを実行する際、必ずしもバインド変数にデータが入っている必要はありません。バインド・ステップでは、変数のアドレス、データ型および長さを指定するのみです。

注意： バインド時にプログラム変数にデータが含まれていない場合は、OCIStmtExecute() を使用して SQL 文または PL/SQL ブロックを実行するときに、必ず有効なデータが含まれるようにする必要があります。

たとえば、次のような INSERT 文があるとしてします。

```
INSERT INTO emp VALUES
    (:empno, :ename, :job, :sal, :deptno)
```

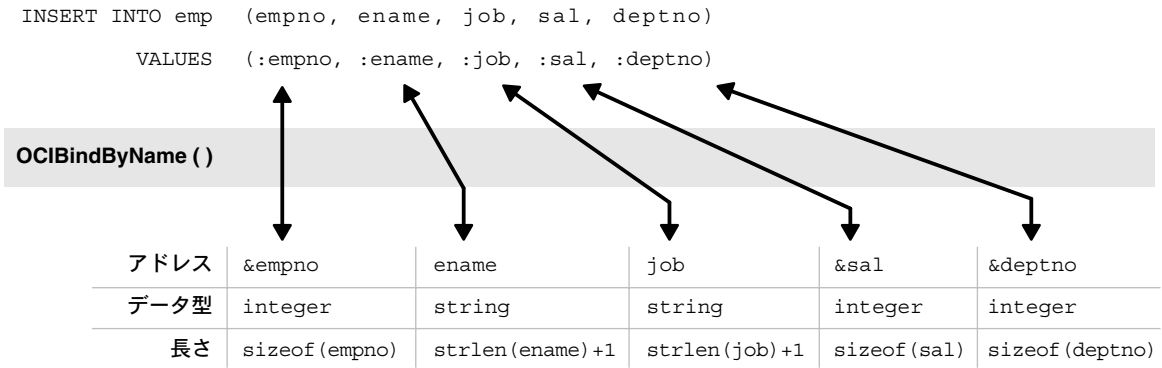
次のように変数を宣言したとしてします。

```
text      *ename, *job;
sword     empno, sal, deptno;
```

バインド・ステップでは、プレースホルダ名とプログラム変数のアドレス間の関連付けを行います。また、[図 5-1](#) に示すように、バインドでは、プログラム変数のデータ型と長さを指定します。

関連項目： この例が実装されているコードについては、5-6 ページの「[バインドで使用するステップ](#)」を参照してください。

図 5-1 OCIBindByName() を使用したプレースホルダとプログラム変数の関連付け



バインド変数の値のみを変更した場合は、文を再実行するために再バインドする必要はありません。バインドは参照による結合であるため、バインド変数のアドレスおよびバインド・ハンドルが有効であるかぎり、再バインドせずに、変数を参照する文を再実行できます。

注意： インタフェース・レベルの場合は、バインド変数はすべて IN であるとみなされるため、適切に初期化する必要があります。変数が純 OUT バインド変数の場合は、この変数を 0（ゼロ）に設定できます。また、NULL インジケータを使用して、そのインジケータを -1（NULL）に設定することもできます。

Oracle サーバーでは、名前付きデータ型、REF および LOB 用の新しいデータ型が実装されています。新しいデータ型は、SQL 文中でプレースホルダとしてバインドできます。

注意： 記述子やロケータなど、サイズが不明で不透明なデータ型の場合は、記述子またはロケータ・ポインタのアドレスを渡してください。サイズ・パラメータに、適切なデータ構造のサイズ (`sizeof(structure)`) を設定します。

名前付きバインドおよび定位置バインド

前の項の SQL 文は、名前付きバインドの例です。文の各プレースホルダには、'ename' や 'sal' などの名前が関連付けられています。この文を準備し、プレースホルダをアプリケーションの値に関連付けるときは、`OCIBindByName()` コールの *placeholder* パラメータでプレースホルダの名前を渡し、プレースホルダの名前による関連付けを行います。

もう 1 つの型のバインドは、定位置バインドです。定位置バインドでは、プレースホルダは名前ではなく文中の位置によって参照されます。このバインドでは、`OCIBindByPos()` コールを使用して、入力値とプレースホルダの位置との間の関連付けを行います。

前の項に示した例は、定位置バインドにも使用できます。

```
INSERT INTO emp VALUES
    (:empno, :ename, :job, :sal, :deptno)
```

5 つのプレースホルダは、`OCIBindByPos()` をコールし、*position* パラメータでプレースホルダの位置番号を渡すことによって、それぞれバインドされます。たとえば、`OCIBindByPos()` をコールすることによって、`:empno` プレースホルダは位置 1 に、`:ename` は位置 2 にというようにバインドされます。

重複バインドの場合、1 つのバインド・コールのみが必要です。次の SQL 文について考えてみます。この SQL 文は、コミッションと給料の両方が指定の金額よりも大きい従業員をデータベースで問い合わせます。

```
SELECT empno FROM emp
    WHERE sal > :some_value
    AND comm > :some_value
```

OCI アプリケーションでは、`OCIBindByName()` を 1 回コールするのみで、`:some_value` プレースホルダを名前によってバインドし、この文のバインドを完了できます。この場合、2 番目のプレースホルダは、最初のプレースホルダからのバインド情報を継承します。

OCI 配列インタフェース

Oracle にデータを渡すには、様々な方法があります。OCIStmtExecute() ルーチンを使用して SQL 文を繰り返し実行し、反復のたびに異なる入力値を指定することもできます。別の方法として、Oracle 配列インタフェースを使用すると、単一の文と OCIStmtExecute() のコール 1 回のみで、多数の値を入力できます。その場合、配列を入力プレースホルダにバインドし、*iters* パラメータで制御して配列全体を同時に渡すことができます。

配列インタフェースを使用すると、大量のデータを更新または挿入する必要がある場合に、Oracle とのラウンドトリップの回数を大幅に削減できます。この削減により、通信量の多いクライアント / サーバー環境では、大きなパフォーマンスの向上につながります。たとえば、データベースに 10 行挿入するアプリケーションを考えてみます。OCIStmtExecute() を 10 回、それぞれ 1 個の値を指定してコールすると、すべてのデータの挿入にネットワーク・ラウンドトリップが 10 回必要です。入力配列を使用すると、OCIStmtExecute() を 1 回コールするのみで同じ結果が得られ、ネットワーク・ラウンドトリップは 1 回で済みます。

注意： OCI 配列インタフェースを使用して挿入を行う場合は、各挿入行が挿入されるたびに、データベース内の行トリガーが起動されます。

PL/SQL のプレースホルダのバインド

PL/SQL ブロックの処理は、単一の SQL 文の場合と同じようにそのブロックを文字列変数に入れて、任意の変数をバインドし、ブロックを含む文を実行して行います。

PL/SQL ブロックのプレースホルダをプログラム変数にバインドする場合は、OCIBindByName() または OCIBindByPos() を使用して基本バインドを行います。OCIBindByName() または OCIBindByPos() を使用すると、スカラーまたは配列のいずれかのホスト変数をバインドできます。

次の短い PL/SQL ブロックには 2 つのプレースホルダが含まれ、従業員番号と新規給与額に基づき、従業員の給与を更新するプロシージャへの IN パラメータを表します。

```
char pls_sql_statement[] = "BEGIN\
    RAISE_SALARY(:emp_number, :new_sal);\
END;" ;
```

これらのプレースホルダは、SQL 文のプレースホルダと同じ方法で入力変数にバインドできます。

PL/SQL 文を処理するとき、出力変数もまたバインド・コールを使用してプログラム変数に関連付けられます。

たとえば、次のような PL/SQL ブロックがあるとします。

```
BEGIN
    SELECT ename,sal,comm INTO :emp_name, :salary, :commission
    FROM emp
    WHERE ename = :emp_number;
END;
```

OCIBindByName() を使用して、変数を `:emp_name`、`:salary` および `:commission` の各出力プレースホルダにバインドし、入力プレースホルダ `:emp_number` にも変数をバインドします。

7.x アップグレードの注意： Oracle7 OCI では、アプリケーションで初期化する必要があるのは IN バインド・バッファのみでした。これより後のリリースでは、バインド・コールでバッファ長を 0 (ゼロ) に設定するか、あるいは対応するインジケータを -1 に設定して、純 OUT バッファの場合も含め、すべてのバッファを初期化する必要があります。

関連項目： PL/SQL プレースホルダのバインドに関する詳細は、11-37 ページの「[名前付きデータ型および REF バインドの情報](#)」を参照してください。

バインドで使用するステップ

プレースホルダをバインドするには、1 つ以上のステップがあります。単純なスカラー・バインドまたは配列バインドの場合は、プレースホルダとデータとの関連付けを指定するのみです。これは、名前 (OCIBindByName()) による OCI バインド、または定位置 (OCIBindByPos()) コールによる OCI バインドを使用して行います。

関連項目： バインド型の違いについては、5-4 ページの「[名前付きバインドおよび定位置バインド](#)」を参照してください。

バインドが完了すると、SQL 文の実行時には、入力データの位置や PL/SQL 出力データを入れる位置が OCI ライブラリに対して明らかになります。5-2 ページの「[バインド](#)」で説明したとおり、プログラム入力データは、プログラム変数をプレースホルダにバインドする際にプログラム変数内に存在している必要はありませんが、文を実行する際には存在している必要があります。

次のコード例は、SQL 文の 5 つのプレースホルダそれぞれのハンドル割当てとバインドを示しています。

注意： checkerr() 関数によって、OCI アプリケーションからのリターン・コードが評価されます。この関数に対するコードのリストは、2-31 ページの「[エラー処理](#)」を参照してください。

```
...
/* The SQL statement, associated with stmthp (the statement handle)
by calling OCISstmtPrepare() */
text *insert = (text *) "INSERT INTO emp(empno, ename, job, sal, deptno)\
VALUES (:empno, :ename, :job, :sal, :deptno)";
...

/* Bind the placeholders in the SQL statement, one for each bind handle. */
checkerr(errhp, OCIBindByName(stmthp, &bnd1p, errhp, (text *) ":ENAME",
    strlen(":ENAME"), (ub1 *) ename, enamelen+1, STRING_TYPE, (dvoid *) 0,
    (ub2 *) 0, (ub2 *) 0, (ub4 *) 0, (ub4 *) 0, OCI_DEFAULT))
checkerr(errhp, OCIBindByName(stmthp, &bnd2p, errhp, (text *) ":JOB",
    strlen(":JOB"), (ub1 *) job, joblen+1, STRING_TYPE, (dvoid *)
    &job_ind, (ub2 *) 0, (ub2 *) 0, (ub4 *) 0, (ub4 *) 0, OCI_DEFAULT))
checkerr(errhp, OCIBindByName(stmthp, &bnd3p, errhp, (text *) ":SAL",
    strlen(":SAL"), (ub1 *) &sal, (sword) sizeof(sal), INT_TYPE,
    (dvoid *) &sal_ind, (ub2 *) 0, (ub2 *) 0, (ub4 *) 0, (ub4 *) 0,
    OCI_DEFAULT))
checkerr(errhp, OCIBindByName(stmthp, &bnd4p, errhp, (text *) ":DEPTNO",
    strlen(":DEPTNO"), (ub1 *) &deptno, (sword) sizeof(deptno), INT_TYPE,
    (dvoid *) 0, (ub2 *) 0, (ub2 *) 0, (ub4 *) 0, (ub4 *) 0, OCI_DEFAULT))
checkerr(errhp, OCIBindByName(stmthp, &bnd5p, errhp, (text *) ":EMPNO",
    strlen(":EMPNO"), (ub1 *) &empno, (sword) sizeof(empno), INT_TYPE,
    (dvoid *) 0, (ub2 *) 0, (ub2 *) 0, (ub4 *) 0, (ub4 *) 0, OCI_DEFAULT))
```

PL/SQL の例

OCI プログラム中の PL/SQL ブロックで最もよく使用されるのは、おそらくストアード・プロシージャまたはストアード・ファンクションのコールです。たとえば、データベース中に RAISE_SALARY というプロシージャが格納されており、これを OCI プログラムからコールするとします。そのためには、そのプロシージャに対するコールを無名の PL/SQL ブロックに埋め込み、OCI プログラムの PL/SQL ブロックを処理します。

次に示すプログラムの一部分は、ストアード・プロシージャ・コールを OCI アプリケーションに埋め込む方法の例です。簡潔にするために、ここではプログラムに関連する部分のみを示しています。

このプログラムでは、これらのパラメータを受け取る raise_salary というストアード・プロシージャへの入力として、従業員番号と新しい給与が渡されます。

```
raise_salary (employee_num IN, sal_increase IN, new_salary OUT);
```

このプロシージャでは、指定された従業員給与に指定された額が上乗せされます。増額された給与はストアド・プロシージャの OUT 変数 `new_salary` に戻され、この値がプログラムで表示されます。

```
/* Define PL/SQL statement to be used in program. */
text *give_raise = (text *) "BEGIN\
        RAISE_SALARY(:emp_number,:sal_increase, :new_salary);\
        END;";

OCIBind *bnd1p = NULL;           /* the first bind handle */
OCIBind *bnd2p = NULL;           /* the second bind handle */
OCIBind *bnd3p = NULL;           /* the third bind handle */

static void checkerr();
sb4 status;

main()
{
    sword empno, raise, new_sal;
    dvoid *tmp;
    OCISession *usrhp = (OCISession *)NULL;
    ...
    /* attach to database server, and perform necessary initializations
    and authorizations */
    ...
    /* allocate a statement handle */
    checkerr(errhp, OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &stmthp,
        OCI_HTYPE_STMT, 100, (dvoid **) &tmp));

    /* prepare the statement request, passing the PL/SQL text
    block as the statement to be prepared */
    checkerr(errhp, OCISTmtPrepare(stmthp, errhp, (text *) give_raise, (ub4)
        strlen(give_raise), OCI_NTV_SYNTAX, OCI_DEFAULT));

    /* bind each of the placeholders to a program variable */
    checkerr( errhp, OCIBindByName(stmthp, &bnd1p, errhp, (text *) ":emp_number",
        -1, (ub1 *) &empno,
        (sword) sizeof(empno), SQLT_INT, (dvoid *) 0,
        (ub2 *) 0, (ub2) 0, (ub4) 0, (ub4 *) 0, OCI_DEFAULT));

    checkerr( errhp, OCIBindByName(stmthp, &bnd2p, errhp, (text *) ":sal_increase",
        -1, (ub1 *) &raise,
        (sword) sizeof(raise), SQLT_INT, (dvoid *) 0,
        (ub2 *) 0, (ub2) 0, (ub4) 0, (ub4 *) 0, OCI_DEFAULT));

    /* remember that PL/SQL OUT variable are bound, not defined */
}
```

```

checkerr( OCIBindByName(stmthp, &bnd3p, errhp, (text *) ":new_salary",
                    -1, (ub1 *) &new_sal,
                    (sword) sizeof(new_sal), SQLT_INT, (dvoid *) 0,
                    (ub2 *) 0, (ub2) 0, (ub4) 0, (ub4 *) 0, OCI_DEFAULT));

    /* prompt the user for input values */
printf("Enter the employee number: ");
scanf("%d", &empno);
    /* flush the input buffer */
myfflush();

printf("Enter employee's raise: ");
scanf("%d", &raise);
    /* flush the input buffer */
myfflush();

    /* execute PL/SQL block*/
checkerr(errhp, OCISmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                    (OCISnapshot *) NULL, (OCISnapshot *) NULL, OCI_DEFAULT));

    /* display the new salary, following the raise */
printf("The new salary is %d\n", new_sal);
}

```

このプログラムの出力例を次に示します。実行前の従業員 7954 番の給与は 2000 です。

```

Enter the employee number: 7954
Enter employee's raise: 1000
The new salary is 3000.

```

前の項では、単純なスカラー・バインドを実行する方法を例で示しました。その例では、必要なバインド・コールは 1 回のみでした。場合によっては、特定のバインド・データ型または実行モードについて特定の属性を定義するために、追加のバインド・コールが必要です。このような高度なバインド操作については、次の項で説明しています。

Oracle には、オブジェクト属性をマップする事前定義済みの C データ型もあります。

関連項目： `OCIDate` や `OCINumber` など、これらのデータ型のバインドの詳細は、[第 12 章「ダイレクト・パス・ロード」](#) を参照してください。

拡張バインド操作

4-6 ページの「[バインド](#)」では、`OCIBindByName()` または `OCIBindByPos()` を使用して、SQL 文のプレースホルダとプログラム変数間の関連付けを行うための基本バインド操作の方法について説明しました。

この項では、マルチステップ・バインド、名前付きデータ型のバインド、REF のバインドなどの拡張バインド操作について説明します。

場合によっては、特定のバインド・データ型または特定の実行モードについて特定の属性を定義するために、追加のバインド・コールが必要です。

以降の項ではこの特別なケースについて説明します。バインドの詳細は、[表 5-1 「バインド型の違いについて」](#) にまとめてあります。

名前付きデータ型のバインド

名前付きデータ型（オブジェクト）のバインドの詳細は、次を参照してください。

関連項目： 11-36 ページ「[名前付きデータ型のバインド](#)」

REF のバインド

この項目の詳細は、次を参照してください。

関連項目： 11-37 ページ「[REF のバインド](#)」

LOB のバインド

LOB をバインドするには、次の 2 つの方法があります。

- 実際の LOB 値ではなく、LOB ロケータをバインドします。この場合、LOB 値の書込みや読み込みは、LOB ロケータを `OCILOB` 関数に渡すことによって行われます。
- LOB ロケータを使用せず、LOB 値を直接バインドします。

これらの方法について、次に説明します。

LOB ロケータのバインド

単一のバインド・コール内で、単一のロケータをバインドすることも、ロケータの配列をバインドすることもできます。いずれの場合も、アプリケーションは、ロケータ自体ではなく、LOB ロケータのアドレスを渡す必要があります。たとえば、アプリケーションで、次のような SQL 文を用意したとします。

```
INSERT INTO some_table VALUES (:one_lob)
```

この SQL 文では、one_lob が、1 つの LOB 列に対応するバインド変数です。このとき、次のような宣言をしたとします。

```
OCILobLocator * one_lob;
```

次のようなステップを使用して、プレースホルダをバインドし、文を実行します。

```
/* initialize single locator */
one_lob = OCIDescriptorAlloc(...OCI_DTYPE_LOB...);
...
/* pass the address of the locator */
OCIBindByName(..., (dvoid *) &one_lob, ... SOLT_CLOB, ...);
OCISstmtExecute(..., 1, ...) /* 1 is the iters parameter */
```

注意： ここでは例を簡潔にするために、大部分のパラメータを省略しています。

また、同一の SQL INSERT 文を使用して、配列の挿入ができます。このケースでは、アプリケーション内に次のコードが組み込まれている必要があります。

```
OCILobLocator * lob_array[10];
...
for (i=0; i<10, i++)
    lob_array[i] = OCIDescriptorAlloc(...OCI_DTYPE_LOB...);
                                /* initialize array of locators */
...
OCIBindByName(..., (dvoid *) lob_array, ...);
OCIBindArrayOfStruct(...);
OCISstmtExecute(..., 10, ...) /* 10 is the iters parameter */
```

記述子は、使用する前に OCIDescriptorAlloc() ルーチンによって割り当てる必要があることに注意してください。ロケータの配列の場合は、OCIDescriptorAlloc() を使用して、各配列要素を初期化する必要があります。BLOB、CLOB および NCLOB を割り当てる場合は、OCI_DTYPE_LOB を type パラメータとして使用します。BFILE を割り当てる場合は、OCI_DTYPE_FILE を使用します。

LOB データのバインド

Oracle では、0 (ゼロ) 以外のあらゆるサイズの LOB で INSERT と UPDATE に対するバインドが可能です。OCIBindByPos()、OCIBindByName() および PL/SQL のバインドを使用すると、最大 4GB のデータを LOB 列にバインドできます。1 行に複数の LOB を含めることができるため、同一の INSERT 文または UPDATE 文中の各 LOB に対して最大 4GB のデータをバインドできます。

4KB を超えるデータを LOB 列にバインドする場合は、一時表領域の一部を使用します。この機能を使用するときは、一時表領域が少なくとも LOB バインドの長さの合計値に等しい量のデータを格納できる大きさであることを確認してください。一時表領域が拡張可能であれば、既存の領域がいっぱいになると自動的に拡張されます。次のコマンドを使用して、

```
"CREATE TABLESPACE ... AUTOEXTENT ON ... TEMPORARY ...;"
```

拡張できる一時表領域を作成します。

LOB バインドに対する制限

- 表に LONG 列と LOB 列の両方がある場合は、LONG 列と LOB 列のどちらに対しても 4KB を超えるデータのバインドが可能ですが、同一の文で両方を実行することはできません。
- オブジェクト・リレーショナル・データ型の LOB 属性には、どのようなサイズのデータもバインドできません。LOB 属性に対しては、空の LOB ロケータを挿入し、OCILOB*() 関数を使用して LOB の内容を変更する必要があります。
- Oracle では、INSERT AS SELECT 操作で、LOB 列に対するデータのバインドはできません。
- 4000 バイトを超えるデータに対しては、HEX から RAW、RAW から HEX などの暗黙的な変換は行われません。次の PL/SQL コードはその例です。

```
create table t (c1 clob, c2 blob);
declare
  text   varchar(32767);
  binbuf raw(32767);
begin
  text := lpad ('a', 12000, 'a');
  binbuf := utl_raw.cast_to_raw(text);

  -- The following works ...
  insert into t values (text, binbuf);

  -- The following won't work because Oracle won't do implicit
  -- hex to raw conversion.
  insert into t (c2) values (text);

  -- The following won't work because Oracle won't do implicit
  -- raw to hex conversion.
  insert into t (c1) values (binbuf);

  -- The following won't work because we can't combine the
  -- utl_raw.cast_to_raw() operator with the >4k bind.
  insert into t (c2) values (utl_raw.cast_to_raw(text));

end;
/
```

- 4000 バイトを超えるデータを BLOB または CLOB にバインドし、データが SQL 演算子によってフィルタされる場合は、結果のサイズが制限されて 4000 バイト以下になります。

次に例を示します。

```
create table t (c1 clob, c2 blob);
-- The following command inserts only 4000 bytes because the result of
-- LPAD is limited to 4000 bytes
insert into t(c1) values (lpad('a', 5000, 'a'));

-- The following command inserts only 2000 bytes because the result of
-- LPAD is limited to 4000 bytes, and the implicit hex to raw conversion
-- converts it to 2000 bytes of RAW data.
insert into t(c2) values (lpad('a', 5000, 'a'));
```

LOB のバインド例

次の例で使用する SQL 文について考えます。

```
CREATE TABLE foo( a INTEGER );
CREATE TYPE lob_typ( A1 CLOB );
CREATE TABLE lob_long_tab (C1 CLOB, C2 CLOB, CT3 lob_typ, L LONG);
```

例 1: LOB のバインド

```
void insert()                      /* A function in an OCI program */
{
    /* The following is allowed */
    ub1 buffer[8000];
    text *insert_sql = "INSERT INTO lob_long_tab (C1, C2, L)
                        VALUES (:1, :2, :3)";
    OCISTmtPrepare(stmthp, errhp, insert_sql, strlen((char*)insert_sql),
                   (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
    OCIBindByPos(stmthp, &bindhp[0], errhp, 1, (dvoid *)buffer, 8000,
                 SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
    OCIBindByPos(stmthp, &bindhp[1], errhp, 2, (dvoid *)buffer, 8000,
                 SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
    OCIBindByPos(stmthp, &bindhp[2], errhp, 3, (dvoid *)buffer, 2000,
                 SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
    OCISTmtExecute(svchp, stmthp, errhp, 1, 0, OCI_DEFAULT);
}
```

例 2: LOB のバインド

```
void insert()
{
    /* The following is allowed */
    ub1 buffer[8000];
    text *insert_sql = "INSERT INTO lob_long_tab (C1, L)
        VALUES (:1, :2)";
    OCISTmtPrepare(stmtthp, errhp, insert_sql, strlen((char*)insert_sql),
        (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
    OCIBindByPos(stmtthp, &bindhp[0], errhp, 1, (dvoid *)buffer, 2000,
        SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
    OCIBindByPos(stmtthp, &bindhp[1], errhp, 2, (dvoid *)buffer, 8000,
        SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
    OCISTmtExecute(svchp, stmtthp, errhp, 1, 0, OCI_DEFAULT);
}
```

例 3: LOB のバインド

```
void insert()
{
    /* The following is allowed, no matter how many rows it updates */
    ub1 buffer[8000];
    text *insert_sql = (text *)"UPDATE lob_long_tab SET
        C1 = :1, C2=:2, L=:3";
    OCISTmtPrepare(stmtthp, errhp, insert_sql, strlen((char*)insert_sql),
        (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
    OCIBindByPos(stmtthp, &bindhp[0], errhp, 1, (dvoid *)buffer, 8000,
        SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
    OCIBindByPos(stmtthp, &bindhp[1], errhp, 2, (dvoid *)buffer, 8000,
        SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
    OCIBindByPos(stmtthp, &bindhp[2], errhp, 3, (dvoid *)buffer, 2000,
        SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
    OCISTmtExecute(svchp, stmtthp, errhp, 1, 0, OCI_DEFAULT);
}
```

例 4: LOB のバインド

```
void insert()
{
    /* The following is allowed, no matter how many rows it updates */
    ub1 buffer[8000];
    text *insert_sql = (text *)"UPDATE lob_long_tab SET
        C1 = :1, C2=:2, L=:3";
    OCISTmtPrepare(stmtthp, errhp, insert_sql, strlen((char*)insert_sql),
        (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
    OCIBindByPos(stmtthp, &bindhp[0], errhp, 1, (dvoid *)buffer, 2000,
        SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
}
```



```

OCIBindByPos(stmthp, &bindhp[1], errhp, 2, (dvoid *)buffer, 2000,
             SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
OCIBindByPos(stmthp, &bindhp[2], errhp, 3, (dvoid *)buffer, 8000,
             SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
OCIStmtExecute(svchp, stmthp, errhp, 1, 0, OCI_DEFAULT);
}

```

例 5: LOB のバインド

```

void insert()
{
    /* Piecewise, callback and array insert/update operations similar to
     * the allowed regular insert/update operations are also allowed */
}

```

例 6: LOB のバインド

```

void insert()
{
    /* The following is NOT allowed because we try to insert >4000 bytes
     * to both LOB and LONG columns */
    ub1 buffer[8000];
    text *insert_sql = (text *)"INSERT INTO lob_long_tab (C1, L)
                               VALUES (:1, :2)";
    OCIStmtPrepare(stmthp, errhp, insert_sql, strlen((char*)insert_sql),
                  (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
    OCIBindByPos(stmthp, &bindhp[0], errhp, 1, (dvoid *)buffer, 8000,
                 SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
    OCIBindByPos(stmthp, &bindhp[1], errhp, 2, (dvoid *)buffer, 8000,
                 SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
    OCIStmtExecute(svchp, stmthp, errhp, 1, 0, OCI_DEFAULT);
}

```

例 7: LOB のバインド

```

void insert()
{
    /* The following is NOT allowed because we try to insert data into
     * LOB attributes */
    ub1 buffer[8000];
    text *insert_sql = (text *)"INSERT INTO lob_long_tab (CT3)
                               VALUES (lob_typ(:1))";
    OCIStmtPrepare(stmthp, errhp, insert_sql, strlen((char*)insert_sql),
                  (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
    OCIBindByPos(stmthp, &bindhp[0], errhp, 1, (dvoid *)buffer, 2000,
                 SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
    OCIStmtExecute(svchp, stmthp, errhp, 1, 0, OCI_DEFAULT);
}

```

例 8: LOB のバインド

```
void insert()
{
    /* The following is NOT allowed because we try to do insert as
     * select character data into LOB column */
    ub1 buffer[8000];
    text *insert_sql = (text *)"INSERT INTO lob_long_tab (C1) SELECT
                                :1 from FOO";
    OCISTmtPrepare(stmthp, errhp, insert_sql, strlen((char*)insert_sql),
                   (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
    OCIBindByPos(stmthp, &bindhp[0], errhp, 1, (dvoid *)buffer, 8000,
                 SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
    OCISTmtExecute(svchp, stmthp, errhp, 1, 0, OCI_DEFAULT);
}
```

その他の実行できない操作

挿入操作と同様に、更新操作にも実行できない操作があります。通常の INSERT や UPDATE で禁止されている操作に類似した、INSERT または UPDATE のピース単位操作およびコールバック操作も実行できません。

関連項目： OCILob 関数の詳細は、[第 7 章「LOB と FILE の操作」](#)を参照してください。

FILE のバインド

FILE ロケータを INSERT 文または UPDATE 文のバインド変数として使用する場合は、INSERT 文または UPDATE 文を発行する前に、(OCILobFileSetName() を使用して) ディレクトリ別名およびファイル名で、ロケータを初期化する必要があります。

OCI_DATA_AT_EXEC モードでのバインド

OCIBindByName() または OCIBindByPos() コールの *mode* パラメータが OCI_DATA_AT_EXEC に設定されているとき、アプリケーションで実行時にデータを提供するコールバック方式を使用する場合は、追加の OCIBindDynamic() コールが必要です。OCIBindDynamic() のコールは、提供されたデータまたはその一部を示すために、必要に応じてコールバック・ルーチンを設定します。

OCI_DATA_AT_EXEC モードを選択した場合でも、コールバックのかわりに標準の OCI のピース単位ポーリング・メソッドを使用する場合は、OCIBindDynamic() コールは必要ありません。

RETURN 句の変数をバインドするとき、アプリケーションでは、OCI_DATA_AT_EXEC モードを使用して、コールバックを提供する必要があります。

関連項目： ピース単位操作の詳細は、5-44 ページの「ランタイム・データ割当てとピース単位操作」を参照してください。

REF カーソル変数のバインド

REF カーソルは、バインド・データ型の `SQLT_RSET` を使用して、文ハンドルにバインドされます。

関連項目： 5-43 ページ「PL/SQL REF CURSOR および NESTED TABLE」

バインド情報の概要

次の表は、バインドの型別に必要なバインド・コールをまとめたものです。この表では、各型について、`OCIBindByName()` または `OCIBindByPos()` の `dtype` パラメータで渡すバインドのデータ型と、そのバインドを使用するときの注意事項をリストしています。

表 5-1 バインド型の違いについて

バインドのタイプ	バインドのデータ型	注意
スカラー	任意のスカラー・データ型	<code>OCIBindByName()</code> または <code>OCIBindByPos()</code> を使用して、単一のスカラーをバインドします。
スカラーの配列	任意のスカラー・データ型	<code>OCIBindByName()</code> または <code>OCIBindByPos()</code> を使用して、スカラーの配列をバインドします。
名前付きデータ型	<code>SQLT_NTY</code>	次の 2 つのバインド・コールが必要です。 <ul style="list-style-type: none">■ <code>OCIBindByName()</code> または <code>OCIBindByPos()</code>■ <code>OCIBindObject()</code>
REF	<code>SQLT_REF</code>	次の 2 つのバインド・コールが必要です。 <ul style="list-style-type: none">■ <code>OCIBindByName()</code> または <code>OCIBindByPos()</code>■ <code>OCIBindObject()</code>
LOB	<code>SQLT_BLOB</code>	<code>OCIDescriptorAlloc()</code> を使用して LOB ロケータを割り当てます。次に LOB データ型の 1 つを使用して、そのアドレス (OCILobLocator**) を <code>OCIBindByName()</code> または <code>OCIBindByPos()</code> とバインドします。
BFILE	<code>SQLT_CLOB</code>	

表 5-1 バインド型の違いについて（続き）

バインドのタイプ	バインドのデータ型	注意
構造体配列 または静的配列	可変	次の 2 つのバインド・コールが必要です。 <ul style="list-style-type: none">OCIBindByName () または OCIBindByPos ()OCIBindArrayOfStruct ()
ピース単位挿入	可変	OCIBindByName () または OCIBindByPos () が必要です。また、ピース単位コールバックを登録するために、アプリケーションで OCIBindDynamic () コールが必要な場合もあります。
REF カーソル変数	SQLT_RSET	文ハンドル、OCISmt を割り当ててから、SQLT_RSET データ型を使用して、その文ハンドルのアドレス（OCISmt **）をバインドします。

関連項目： データ型およびデータ型コードの詳細は、[第 3 章「データ型」](#)を参照してください。

定義

問合せ文は、データベースのデータをアプリケーションに戻します。問合せを処理するときには、データを検索する選択リストの各項目について、出力変数または出力変数の配列を定義する必要があります。定義ステップでは、戻される結果の格納場所と格納形式を判断する関連付けを行います。

たとえば、OCI 文で次の文を処理するとします。

```
SELECT name, ssn FROM employees
WHERE empno = :empnum
```

この場合、通常は 2 つの出力変数を定義する必要があります。1 つは、name 列から戻される値を受け取るための変数、もう 1 つは ssn 列から戻される値を受け取るための変数です。

注意： name 列の値のみを取り出す場合は、ssn 列用の出力変数を定義する必要はありません。

処理される SELECT 文が問合せに対して複数の値を戻す場合、出力変数はスカラー値のかわりに配列を定義できます。

定義ステップは、アプリケーションに応じて文の実行前または実行後に行います。アプリケーションをコーディングする時点で選択リスト項目のデータ型がわかっている場合には、文を実行する前に定義を行うことができます。これに対して、動的 SQL 文（実行時に入力する文など）、あるいは次のように明確に定義された選択リストがない文をアプリケーションで処理する場合があります。

```
SELECT * FROM employees
```

このような場合、アプリケーションでは出力変数を定義する前に文を実行し、記述情報を取り出す必要があります。

関連項目： 詳細は、4-12 ページの「[選択リスト項目の記述](#)」を参照してください。

OCI では、クライアント側で定義コールをローカルに処理します。定義ステップでは、結果を格納するバッファの位置を指示し、さらに、データがアプリケーションに戻されるときに必要なに応じて行うデータ変換の種類を判断します。

注意： 出力バッファには、2 バイトの位置揃えを行う必要があります。

OCIDefineByPos() コールの *dy* パラメータでは、出力変数のデータ型を指定します。OCI では、データを出力変数にフェッチするときに広範囲のデータ変換を行うことができます。たとえば、Oracle DATE 形式の内部データは、出力時に、自動的に文字列データ型に変換できます。

関連項目： データ型および変換の詳細は、[第 3 章「データ型」](#)を参照してください。

定義に使用するステップ

出力変数を定義するには、1 つ以上のステップを実行します。基本的な定義は、定位置コールの OCIDefineByPos() により、OCI 定義を使用して完了します。このステップでは、選択リスト項目と出力変数との間の関連付けを行います。特定のデータ型またはフェッチ・モードには追加の定義コールが必要です。

定義ステップが完了すると、OCI ライブラリでは、データベースからデータをフェッチした後、そのデータを入れる位置が明らかになります。

注意： SQL 文の準備や実行をやり直すことなく、定義コールを再度行って再定義できます。

次のコード例では、実行と記述の後にスカラー出力変数を定義しています。

```
/* The following statement was prepared, and associated with statement
   handle stmthp1.

   SELECT dname FROM dept WHERE deptno = :dept_input

   The input placeholder was bound earlier, and the data comes from the
   user input below */

printf("Enter employee dept: ");
scanf("%d", &deptno);
myfflush();

/* Execute the statement. If OCISstmtExecute() returns OCI_NO_DATA, meaning that no
data matches the query, then the department number is invalid. */
if ((status = OCISstmtExecute(svchp, stmthp1, errhp, 0, 0, 0, 0,
OCI_DEFAULT))
    && (status != OCI_NO_DATA))
{
    checkerr(errhp, status);
    do_exit(EXIT_FAILURE);
}
if (status == OCI_NO_DATA) {
    printf("The dept you entered doesn't exist.\n");
    return 0;
}

/* The next two statements describe the select-list item, dname, and
   return its length */
checkerr(errhp, OCIParamGet(stmthp1, errhp, &parmdp, (ub4) 1));
checkerr(errhp, OCIAttrGet((dvoid*) parmdp, (ub4) OCI_DTYPE_PARAM,
(dvoid*) &deptlen, (ub4 *) 0, (ub4) OCI_ATTR_DATA_SIZE,
(OCIError *) errhp ));

/* Use the retrieved length of dname to allocate an output buffer, and
   then define the output variable. If the define call returns an error,
   exit the application */
dept = (text *) malloc((int) deptlen + 1);
if (status = OCIDefineByPos(stmthp1, &defnp, errhp,
    1, (ub1 *) dept, deptlen+1,
    SQLT_STRING, (dvoid *) 0,
    (ub2 *) 0, OCI_DEFAULT))
{
    checkerr(errhp, status);
    do_exit(EXIT_FAILURE);
}
```

関連項目： 記述ステップの説明は、4-12 ページの「[選択リスト項目の記述](#)」を参照してください。

拡張定義

場合によっては、定義ステップでは `OCIDefineByPos()` のコールのみでなく、他のコールも必要です。配列のフェッチ (`OCIDefineArrayOfStruct()`) または名前付きデータ型のフェッチ (`OCIDefineObject()`) の属性を定義する追加コールがあります。たとえば、1 つの名前付きデータ型列を指定して複数の行をフェッチするには、列に対して 3 つのコールすべてを呼び出す必要がありますが、スカラー列の複数行をフェッチするには、`OCIDefineArrayOfStruct()` と `OCIDefineByPos()` で済みます。

関連項目： さらに高度な定義操作の詳細は、5-21 ページの「[拡張定義操作](#)」で説明します。

また、Oracle には、オブジェクト型属性をマップする事前定義済みの C データ型が用意されています。

関連項目： これらのデータ型 (`OCIDate` や `OCINumber` など) の定義の詳細は、[第 11 章「オブジェクト・リレーショナル・データ型」](#) を参照してください。

拡張定義操作

4-15 ページの「[定義](#)」では、アプリケーションで SQL 選択リスト項目と出力バッファ間の関連付けを行うための基本的な定義操作の方法について説明しました。

この項では、マルチステップ定義、名前付きデータ型定義、REF 定義などの拡張定義操作について説明します。

場合によっては、定義ステップでは `OCIDefineByPos()` のコールのみでなく、他のコールも必要です。配列のフェッチ (`OCIDefineArrayOfStruct()`) または名前付きデータ型のフェッチ (`OCIDefineObject()`) の属性を定義する追加コールがあります。たとえば、1 つの名前付きデータ型列を指定して複数の行をフェッチするには、列に対して 3 つのコールすべてを呼び出す必要がありますが、スカラー列の複数行をフェッチするには、`OCIDefineArrayOfStruct()` と `OCIDefineByPos()` のみで済みます。

次の項では、様々なタイプの定義に関する特定の情報を説明します。

名前付きデータ型出力変数の定義

名前付きデータ型 (オブジェクト) 出力変数の定義の詳細は、11-38 ページの「[名前付きデータ型出力変数の定義](#)」を参照してください。

REF 出力変数の定義

REF 出力変数の定義の詳細は、11-39 ページの「[REF 出力変数の定義](#)」を参照してください。

LOB 出力変数の定義

LOB を定義するには、次の 2 つの方法があります。

- 実際の LOB 値ではなく、LOB ロケータとして定義します。この場合、LOB 値の書き込みや読み込みは、LOB ロケータを `OCILob` 関数に渡すことによって行われます。
- LOB ロケータを使用せず、LOB 値を直接定義します。

これらの方法について、次に説明します。

LOB ロケータの定義

単一の定義コール内では、単一のロケータを定義することも、ロケータの配列を定義することもできます。いずれの場合も、アプリケーションは、ロケータ自体ではなく、LOB ロケータのアドレスを渡す必要があります。たとえば、アプリケーションで、次のような SQL 文を準備したとします。

```
SELECT lob1 FROM some_table;
```

この SQL 文の `lob1` は LOB 列です。`one_lob` は、次の宣言を伴った LOB 列に対応する定義変数です。

```
OCILobLocator * one_lob;
```

次のようなステップを使用して、ブレースホルダをバインドし、文を実行します。

```
/* initialize single locator */
one_lob = OCIDescriptorAlloc(...OCI_DTYPE_LOB...);
...
/* pass the address of the locator */
OCIDefineByPos(... 1, ..., (dvoid *) &one_lob, ... SQLT_CLOB, ...);
OCIStmtExecute(...,1,...) /* 1 is the iters parameter */
```

注意： ここでは例を簡潔にするために、大部分のパラメータを省略しています。

また、同一の SQL SELECT 文を使用して、配列を選択できます。このケースでは、アプリケーション内に次のコードが組み込まれている必要があります。

```
OCILobLocator * lob_array[10];
...
for (i=0; i<10, i++)
    lob_array[i] = OCIDescriptorAlloc(...OCI_DTYPE_LOB...);
                                /* initialize array of locators */
...
OCIDefineByPos(...,1, (dvoid *) lob_array,... SQLT_CLOB, ...);
OCIDefineArrayOfStruct(...);
OCISetStmtExecute(...,10,...);          /* 10 is the iters parameter */
```

記述子は、使用する前に OCIDescriptorAlloc() ルーチンによって割り当てる必要があることに注意してください。ローケータの配列の場合は、OCIDescriptorAlloc() を使用して、各配列要素を初期化する必要があります。BLOB、CLOB および NCLOB を割り当てる場合は、OCI_DTYPE_LOB を type パラメータとして使用します。BFILE を割り当てる場合は、OCI_DTYPE_FILE を使用します。

LOB データの定義

Oracle では、0 (ゼロ) 以外のあらゆるサイズの LOB で SELECT の定義が可能です。OCIDefineByPos() および PL/SQL の定義を使用すると、最大 4GB のデータを LOB 列から選択できます。1 行に複数の LOB を含めることができるため、同一の SELECT 文中の各 LOB に対して最大 4GB のデータを選択できます。

LOB の定義例

次の例で使用される SQL 文について考えます。

```
CREATE TABLE lob_tab (C1 CLOB, C2 CLOB);
```

例 1: LOB の定義

```
void select_define_before_execute()          /* A function in an OCI program */
{
    /* The following is allowed */
    ub1 buffer1[8000];
    ub1 buffer2[8000];
    text *select_sql = "SELECT c1, c2 FROM lob_tab";
    OCISetStmtPrepare(stmthp, errhp, select_sql, strlen((char*)select_sql),
                      (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
    OCIDefineByPos(stmthp, &defhp[0], errhp, 1, (dvoid *)buffer1, 8000,
                   SQLT_LNG, 0, 0, 0, (ub4) OCI_DEFAULT);
    OCIDefineByPos(stmthp, &defhp[1], errhp, 2, (dvoid *)buffer2, 8000,
                   SQLT_LNG, 0, 0, 0, (ub4) OCI_DEFAULT);
    OCISetStmtExecute(svchp, stmthp, errhp, 1, 0, OCI_DEFAULT);
}
```

例 2: LOB の定義

```
void select_execute_before_define()
{
    /* The following is allowed */
    ub1 buffer1[8000];
    ub1 buffer2[8000];
    text *select_sql = "SELECT c1, c2 FROM lob_tab";
    OCISTmtPrepare(stmthp, errhp, select_sql, strlen((char*)select_sql),
        (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
    OCISTmtExecute(svchp, stmthp, errhp, 0, 0, OCI_DEFAULT);
    OCIDefineByPos(stmthp, &bindhp[0], errhp, 1, (dvoid *)buffer1, 8000,
        SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
    OCIDefineByPos(stmthp, &bindhp[1], errhp, 2, (dvoid *)buffer2, 8000,
        SQLT_LNG, 0, 0, 0, 0, 0, (ub4) OCI_DEFAULT);
    OCISTmtFetch(stmthp, errhp, 1, OCI_FETCH_NEXT, OCI_DEFAULT);
}
```

例 3: LOB の定義

```
void select()
{
    /* Piecewise, callback and array select operations similar to
     * the allowed regular select operations are also allowed */
}
```

PL/SQL 出力変数の定義

PL/SQL ブロック内の SQL SELECT 文の選択リスト項目については、出力変数を定義するのに定義コールは使用しません。かわりに、OCI バインド・コールを使用する必要があります。

関連項目： PL/SQL 出力変数の定義に関する詳細は、11-39 ページの「[名前付きデータ型、REF 定義および PL/SQL OUT バインドの情報](#)」を参照してください。

ピース単位フェッチの定義

ピース単位フェッチを実行する場合は、最初に OCIDefineByPos() をコールする必要があります。アプリケーションで、データをフェッチするときの標準のポーリング・メカニズムではなくコールバックを使用する場合は、追加の OCIDefineDynamic() コールが必要です。

関連項目： 詳細は、5-44 ページの「[ランタイム・データ割当てとピース単位操作](#)」を参照してください。

構造体配列のバインドと定義

構造体配列を使用する場合は、最初に `OCIDefineByPos()` をコールする必要があります。構造体配列操作に必要なスキップ・パラメータなどの追加パラメータを設定するには、追加の `OCIDefineArrayOfStruct()` コールが必要です。

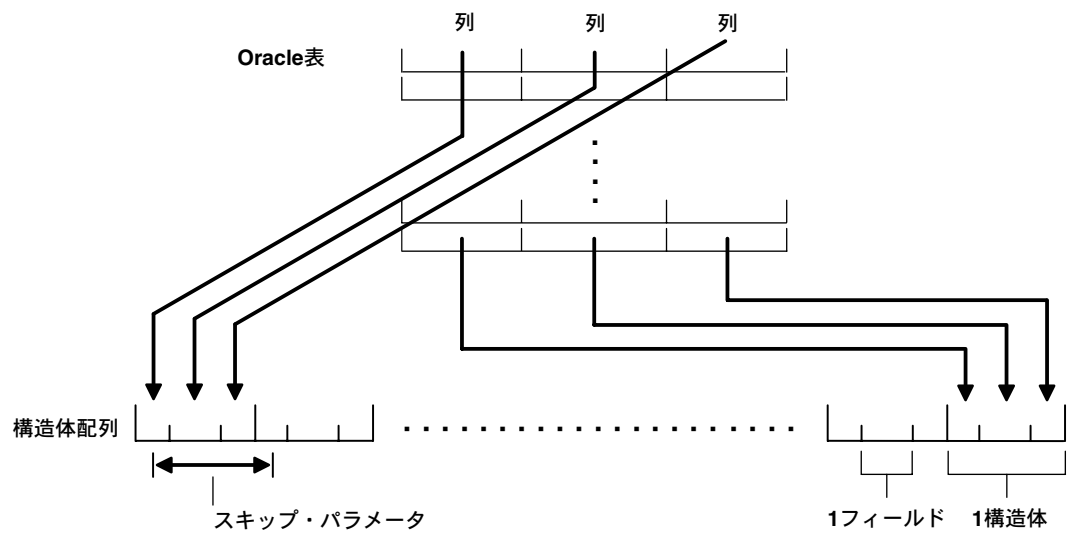
OCI の構造体配列の機能により、複数行、複数列の操作の処理を簡素化できます。OCI のプログラミングでは、関連したスカラー・データ項目の構造体を作成し、データベースの値を構造体配列へフェッチしたり、これらの構造体配列の値をデータベースに挿入できます。

たとえば、あるアプリケーションで `NAME`、`AGE` および `SALARY` という名前の 3 つの列から、複数行のデータをフェッチする必要があるとします。OCI アプリケーションでは、個別のフィールド（それぞれが、データベース表で 1 行に含まれている `NAME`、`AGE` および `SALARY` のデータを保持しています）を含む構造体の定義を組み込むことができます。アプリケーションでは、次に、これらの構造体配列へデータをフェッチします。

構造体配列を使用して、複数行、複数列の操作を実行するには、その操作に関連する各列を、構造体の中の 1 フィールドと関連付けておく必要があります。この関連付けは、`OCIDefineArrayOfStruct()` コールと `OCIBindArrayOfStruct()` コールの一部で、これにより、フェッチされたデータの格納場所や、挿入または更新されたデータの位置を指定します。

図 5-2「構造体配列へのデータのフェッチ」は、この処理をグラフィック形式で示しています。この図では、アプリケーションがデータベース行から様々なフィールドをフェッチし、構造体配列の中の 1 つの構造体に挿入しています。フェッチされた各列は、構造体内のフィールドの 1 つと対応しています。

図 5-2 構造体配列へのデータのフェッチ



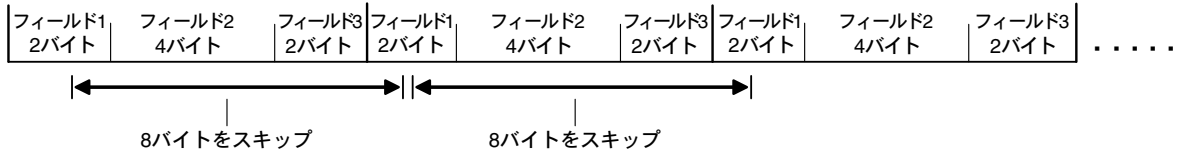
スキップ・パラメータ

構造体配列間で列データを分割すると、それは連続した列データではなくなります。単一の構造体配列は、いくつかのインターリーブされたスカラー配列で構成されているかのようにデータを格納します。このため、開発者はバインドまたは定義を行う各フィールドにスキップ・パラメータを指定する必要があります。スキップ・パラメータでは、構造体配列の中にある同じフィールドまでスキップするために、必要なバイト数を指定します。通常は、1つの構造体のバイト数と等しい値です。

次の図は、スキップ・パラメータの決定方法を示しています。このケースでは、スキップ・パラメータは、フィールド 1、フィールド 2 およびフィールド 3 のサイズの合計です。それぞれのフィールドのサイズは、8 バイトです。このサイズは、1つの構造体のサイズと同じです。

図 5-3 スキップ・パラメータの決定

構造体配列



システムによっては、スキップ・パラメータを `sizeof(struct)` ではなく、`sizeof(one_array_element)` に設定することが必要な場合があります。これは、コンパイラによっては、構造体に埋込みを挿入する場合があるからです。たとえば、**ub4** と **ub1** という 2 つのフィールドで構成された C 構造体配列のケースを考えてみます。

```
struct demo {
    ub4 field1;
    ub1 field2;
};
struct demo demo_array[MAXSIZE];
```

コンパイラによっては、この配列内の次の構造体を開始する **ub4** の位置を正しくするために、**ub1** の後に 3 バイトの埋込みを挿入することがあります。このようなコンパイラでは、次の文を使用すると、誤った値が戻される可能性があります。

```
skip_parameter = sizeof(struct demo);
```

この文を使用した場合、8 バイトの適切なスキップ・パラメータを戻すシステムもありますが、5 バイトの `skip_parameter` を戻すシステムもあります。そのような場合、スキップ・パラメータの正しい値を得るには次の文を使用します。

```
skip_parameter = sizeof(demo_array[0]);
```

標準配列のスキップ・パラメータ

構造体配列を処理する機能は、プログラム変数の配列をバインドおよび定義する機能を拡張したものです。プログラマは、(構造体配列に対するものとして) 標準の配列も使用できます。標準の配列操作を指定する場合、関連したスキップ・パラメータは、その配列のデータ型のサイズと等しくなります。たとえば、配列を次のように宣言したとします。

```
text emp_names[4][20];
```

バインドまたは定義操作のスキップ・パラメータは 20 になります。このようにして、配列の各データ要素は、構造体の一部としてでなく別個の単位として認識されます。

構造体配列で使用する OCI コール

構造体配列に関係する操作を行う際には、次の 2 つの OCI コールを使用する必要があります。OCIBindArrayOfStruct()（入力変数に対して構造体配列のフィールドをバインドする場合）および OCIDefineArrayOfStruct()（出力変数に対して構造体配列を定義する場合）です。

注意： 構造体配列をバインドまたは定義する際は、複数のコールが必要です。OCIBindByName() または OCIBindByPos() へのコールは、OCIBindArrayOfStruct() へのコールに先行し、OCIDefineByPos() へのコールは OCIDefineArrayOfStruct() に先行する必要があります。

関連項目： 構文およびパラメータの詳細は、[OCIBindArrayOfStruct\(\)](#) および [OCIDefineArrayOfStruct\(\)](#) の説明を参照してください。

構造体配列とインジケータ変数

構造体配列を実装することによって、インジケータ変数とリターン・コードの使用が可能になります。OCI アプリケーションの開発者は、フェッチ、挿入または更新された情報の配列に対応して、列レベルのインジケータ変数とリターン・コードの平行配列を宣言できます。これらの配列には、独自のスキップ・パラメータを含めることができます。スキップ・パラメータは、OCIBindArrayOfStruct() または OCIDefineArrayOfStruct() コール中に指定します。

プログラム値およびインジケータ変数の構造体配列は、様々な方法で設定できます。たとえば、データベースの 3 列から、構造体配列の 3 フィールドへデータをフェッチするアプリケーションを考えてみます。この場合、3 フィールドに対応するインジケータ変数の構造体配列を設定できます。それぞれのインジケータ変数は、データベースからフェッチされる列の中の 1 列に対する列レベルのインジケータ変数です。

注意： インジケータ構造体のフィールドと選択リスト項目の数に 1 対 1 関係は必要ありません。

関連項目： インジケータ変数の詳細は、2-36 ページの「[インジケータ変数](#)」を参照してください。

RETURNING 句を使用した DML

OCI では、SQL INSERT 文、UPDATE 文および DELETE 文での RETURNING 句の使用をサポートします。この項では、RETURNING 句を使用して DML 文を正しく実装するために、OCI アプリケーションが従う必要があるルールについて、その概要を説明します。

注意： INSERT 文、UPDATE 文または DELETE 文で RETURNING 句を使用する場合の詳細は、『Oracle9i SQL リファレンス』の各コマンドの説明を参照してください。

関連項目： コード例全体は、Oracle のインストールに含まれているデモ・プログラムを参照してください。追加情報については、[付録 B「OCI デモ・プログラム」](#)を参照してください。

RETURNING 句を使用した DML の使用方法

DML 文で RETURNING 句を使用することにより、2 つの SQL 文を本質的に 1 つに結合でき、サーバー・ラウンドトリップを削減できます。これは、追加の句を従来の UPDATE 文、INSERT 文および DELETE 文に追加することで実行できます。追加の句では、問合せを DML 文に効果的に追加します。

OCI では、OUT バインド変数を通して値がアプリケーションに戻されます。この変数をバインドするルールについては、次の項で説明しています。次の例では、:out1 のように、バインド変数の前にコロンが付いています。これらの例では、col1、col2 および col3 の 3 つの列を含む table1 という表を想定しています。

たとえば、次の文では、新しい値をデータベースに挿入して、影響を受ける行の列値をデータベースから取り出します。これにより、挿入された行をアプリケーションで操作できます。

```
INSERT INTO table1 VALUES (:1, :2, :3,)
      RETURNING col1, col2, col3
      INTO :out1, :out2, :out3
```

次の例では、UPDATE 文を使用しています。この文は、col1 の値がある範囲内にあるすべての列の値を更新し、影響を受けた行をアプリケーションへ戻します。これにより、アプリケーションは、どの行が変更されたかを判別できます。

```
UPDATE table1 SET col1 = col1 + :1, col2 = :2, col3 = :3
      WHERE col1 >= :low AND col1 <= :high
      RETURNING col1, col2, col3
      INTO :out1, :out2, :out3
```

次の DELETE 文では、col1 の値が特定の範囲内にある行を削除し、それらの行からデータを戻します。これにより、アプリケーションではそのデータをチェックできます。

```
DELETE FROM table1 WHERE col1 >= :low AND col2 <= :high
    RETURNING col1, col2, col3
    INTO :out1, :out2, :out3
```

UPDATE 文と DELETE 文の例では、文が表内の複数行に影響を与える可能性があることに注意してください。さらに、1 つの DML 文は、1 つの OCIExecute() 文の中で複数回実行できます。このように、複数の値が戻ってくる可能性があるため、OCI アプリケーションでは、実行時にどれだけのデータが戻ってくるのかを判断できない場合があります。そのため、RETURNING...INTO プレースホルダに対応する変数は、OCI_DATA_AT_EXEC モードでバインドする必要があります。その他の要件として、アプリケーションでは、OCI_DATA_AT_EXEC ポーリング・メカニズムを使用するのではなく、アプリケーション自体の動的データ処理コールバックを定義する必要があります。

注意： アプリケーションで、RETURNING 句によって戻される値が 1 つのみであることが確実な場合でも、OCI_DATA_AT_EXEC モードでバインドし、コールバックを行う必要があります。

RETURNING 句は、LOB を操作するときに特に役立ちます。通常、アプリケーションでは、空の LOB ロケータをデータベースに挿入し、再度選択 (SELECT) して操作を行います。RETURNING 句を使用すると、アプリケーションではこれら 2 つのステップを 1 つの文に結合できます。

```
INSERT INTO some_table VALUES (:in_locator)
    RETURNING lob_column
    INTO :out_locator
```

RETURNING...INTO 変数のバインド

OCI アプリケーションは、RETURNING 句のプレースホルダを純 OUT バインド変数として実装します。ただし、RETURNING 句のすべてのバインドの初期値は IN であるため、適切に初期化する必要があります。有効な値を設定するために、NULL インジケータを使用して、そのインジケータを -1 (NULL) に設定できます。

アプリケーションでは、RETURNING 句のバインド変数を操作するとき、次のルールに従う必要があります。

1. 各プレースホルダに対する OCIBindDynamic() のコールの後に、OCIBindByName() または OCIBindByPos() を使用して、OCI_DATA_AT_EXEC モードで RETURNING 句のプレースホルダをバインドします。

注意： OCI では、RETURNING 句バインドのコールバック・メカニズムのみがサポートされています。ポーリング・メカニズムはサポートされていません。

2. RETURNING 句のプレースホルダをバインドする場合は、OCIBindDynamic() コールの際の *ocbfp* パラメータとして、有効な OUT バインド関数を指定する必要があります。この関数では、戻されたデータを保持するための記憶域が必要です。
3. OCIBindDynamic() コールの *icbfp* パラメータによって、コール時に NULL 値を戻すダミーの関数を提供する必要があります。
4. OCIBindDynamic() の *piecep* パラメータを、OCI_ONE_PIECE に設定します。
5. RETURNING 句を使用した DML 文では、複製バインドは使用できません（つまり、DML セクションのバインド変数と、その文の RETURNING セクションのバインド変数を重複させることはできません）。

エラー処理

OCIBindDynamic() に提供された OUT バインド関数には、エラー時に文の結果の一部を受け取る準備が必要です。たとえば、10 回実行される DML 文をアプリケーションで発行し、エラーが 5 回目の実行中に発生した場合、サーバーは、1 回目から 4 回目までのデータを戻します。コールバック関数は、最初の 4 つの文のデータを受け取るためにデータをコールします。

RETURNING REF..INTO 句を使用した DML

RETURNING 句を使用して、データベースに挿入するオブジェクトやデータベース内で更新するオブジェクトへの REF を戻すこともできます。次の SQL 文で、その方法を示します。

```
UPDATE EXTADDR E SET E.ZIP = '12345', E.STATE='AZ'
WHERE E.STATE = 'CA' AND E.ZIP='95117'
RETURNING REF(E), ZIP
INTO :addref, :zip
```

この文では、オブジェクト表内のオブジェクトの属性をいくつか更新し、RETURNING 句でオブジェクトへの REF を（スカラー ZIP コードとともに）戻します。

OCI アプリケーションで REF 出力変数をバインドするには、次の 3 つのステップが必要です。

1. 初期バインド情報を、OCIBindByName() を使用して設定します。
2. REF (TDO を含みます) に対する追加のバインド情報は、OCIBindObject() を使用して設定します。
3. OCIBindDynamic() をコールします。

次のコード例では、前述の例で必要なバインドを実行する関数を示します。

```
sword bind_output(stmthp, bndhp, errhp)
OCIStmt *stmthp;
OCIBind *bndhp[];
OCIError *errhp;
{
    ub4 i;

                                /* get TDO for BindObject call */
    if (OCITypeByName(envhp, errhp, svchp, (CONST text *) 0,
                      (ub4) 0, (CONST text *) "ADDRESS_OBJECT",
                      (ub4) strlen((CONST char *) "ADDRESS_OBJECT"),
                      (CONST text *) 0, (ub4) 0,
                      OCI_DURATION_SESSION, OCI_TYPEGET_HEADER, &addrtdo))
    {
        return OCI_ERROR;
    }

                                /* initial bind call for both variables */
    if (OCIBindByName(stmthp, &bndhp[2], errhp,
                      (text *) ":addref", (sb4) strlen((char *) ":addref"),
                      (dvoid *) 0, (sb4) sizeof(OCIRef *), SQLT_REF,
                      (dvoid *) 0, (ub2 *) 0, (ub2 *) 0,
                      (ub4) 0, (ub4 *) 0, (ub4) OCI_DATA_AT_EXEC)
    || OCIBindByName(stmthp, &bndhp[3], errhp,
                      (text *) ":zip", (sb4) strlen((char *) ":zip"),
                      (dvoid *) 0, (sb4) MAXZIPLLEN, SQLT_CHR,
                      (dvoid *) 0, (ub2 *) 0, (ub2 *) 0,
                      (ub4) 0, (ub4 *) 0, (ub4) OCI_DATA_AT_EXEC))
    {
        return OCI_ERROR;
    }

                                /* object bind for REF variable */
    if (OCIBindObject(bndhp[2], errhp, (OCIType *) addrtdo,
                      (dvoid **) &addrref[0], (ub4 *) 0, (dvoid **) 0, (ub4 *) 0))
    {
        return OCI_ERROR;
    }
}
```

```

for (i = 0; i < MAXCOLS; i++)
    pos[i] = i;
    /* dynamic binds for both RETURNING variables */
if (OCIBindDynamic(bndhp[2], errhp, (dvoid *) &pos[0], cbf_no_data,
    (dvoid *) &pos[0], cbf_get_data)
|| OCIBindDynamic(bndhp[3], errhp, (dvoid *) &pos[1], cbf_no_data,
    (dvoid *) &pos[1], cbf_get_data))
{
    return OCI_ERROR;
}

return OCI_SUCCESS;
}

```

コールバックに関するその他の注意

コールバック関数がコールされるとき、バインド・ハンドルの

OCI_ATTR_ROWS_RETURNED 属性により、アプリケーションでは、ある特定の反復処理で戻される行数がわかります。したがって、コールバックが、ある特定の反復処理で初めてコールされるとき（索引 =0 の場合）は、そのバインド変数について戻されるすべての行に、空白を割り当てることができます。コールバックが同一の反復処理内で引き続きコールされるとき（索引 >0 の場合）は、割り当てた空白内の正確なメモリーまでバッファ・ポインタを増分させてデータを取得します。

DML RETURNING 文に対する配列インタフェース

OCI では、各反復処理で複数の行を戻す単一行 DML 操作および配列 DML 操作に追加の機能性を提供します。この機能を利用するには、クライアント・アプリケーションで、バインド・コールの OUT バッファを、OCIStmtExecute() コールで指定した反復件数以上の値に指定する必要があります。これは、コールバックを介してバインド・バッファに提供される追加機能です。

文の実行時にいずれかの反復処理で複数の行が戻された場合は、アプリケーションに OCI_SUCCESS_WITH_INFO リターン・コードが戻されます。この場合、DML 操作は正常に完了しています。このとき、アプリケーションでは、トランザクションをロールバックするか、警告を無視します。

バインドおよび定義における文字変換の問題

この項では、クライアントとサーバー間での文字変換に関する問題について説明します。

キャラクタ・セットの選択

Oracle では、データベースの文字データをサポートします。また、OCI では、文字データのバインドと定義をサポートします。文字データを含むデータベースの列が NCHAR/NVARCHAR2 列として定義されている場合、その列に関連するバインドまたは定義では、キャラクタ・セット仕様の処理について特別な配慮が必要です。

クライアントとサーバーのキャラクタ・セットの幅が異なる場合、およびクライアントとサーバー間で適切に文字変換を行うために、これらの考慮が必要です。異なるキャラクタ・セット間でデータを変換する間に、データ・サイズが、4 倍に増加したり 4 分の 1 に縮小する場合があります。そのため、データを保存するためのバッファのサイズを十分に確保する必要があります。

アプリケーションでは、NCHAR/NVARCHAR2 データをバイト数（通常の場合）ではなく文字数で処理の方が簡単な場合があります。

キャラクタ・セット・フォームとキャラクタ・セット ID

各 OCI バインドおよび定義ハンドルには、OCI_ATTR_CHARSET_FORM 属性および OCI_ATTR_CHARSET_ID 属性が関連付けられています。アプリケーションでは、バインド / 定義バッファのキャラクタ・フォームおよびキャラクタ・セット ID を指定するために、OCIAttrSet() コールを使用してこれらの属性を設定できます。

form 属性 (OCI_ATTR_CHARSET_FORM) は、バインドの場合はクライアント・バッファが含まれたキャラクタ・セットを示し、定義の場合はフェッチされたデータを格納するキャラクタ・セットを示します。この属性の値は、次のうちいずれかになります。

- SQLCS_IMPLICIT — データベース・キャラクタ・セット ID を示します。
- SQLCS_NCHAR — 各国語キャラクタ・セット ID を示します。

デフォルト値は SQLCS_IMPLICIT です。この値はバインド・バッファまたは定義バッファ用のデータベース・キャラクタ・セットを示し、バッファ内の文字データはサーバーのデータベース・キャラクタ・セットに変換されます。SQLCS_NCHAR はバインド・バッファまたは定義バッファ用の各国語キャラクタ・セット ID を示し、クライアントのバッファ・データはサーバーの各国語キャラクタ・セットに変換されます。

キャラクタ・セット ID 属性 (OCI_ATTR_CHARSET_ID) が指定されていない場合は、*form* の値に応じて、クライアントのデータベース・キャラクタ・セット ID または各国語キャラクタ・セット ID のデフォルト値が使用されます。これらの値は、環境変数 NLS_LANG または NLS_NCHAR で指定された値です。

注意：

- クライアント側バインド・バッファのキャラクタ・セット ID とフォームに割り当てられている値に関係なく、データはサーバーのデータベース・キャラクタ・セット ID または各国語キャラクタ・セット ID に従って変換され、データベースに挿入されます。
 - OCI_ATTR_CHARSET_ID に 0（ゼロ）を設定することはできません。
-
-

関連項目： NCHAR データの詳細は、『Oracle9i データベース・リファレンス』を参照してください。

CHAR と NCHAR の間の暗黙的な変換

データベース・キャラクタ・セットと各国語キャラクタ・セット間での暗黙的な変換の結果、OCI では、CHAR と NCHAR の間でのクロスバインドおよびクロス定義に対するサポートが可能となりました。たとえば、OCI_ATTR_CHARSET_FORM 属性が SQLCS_NCHAR に設定されている場合でも、OCI では、データが CHAR 列に挿入されると、データをデータベース・キャラクタ・セットに変換できます。

OCI でのクライアント・キャラクタ・セットの設定

キャラクタ・セットは、OCIEnvCreateNLS() 関数の charset パラメータおよび ncharset パラメータを使用して設定できます。両方のパラメータとも OCI_UTF16ID を設定できます。charset パラメータはメタデータおよび CHAR データのコーディングを制御し、ncharset パラメータは NCHAR データのコーディングを制御します。OCINlsEnvironmentVariableGet() 関数は、NLS_LANG のキャラクタ・セットおよび NLS_NCHAR の各国語キャラクタ・セットを戻します。

次の擬似コードは、これらの関数の使用例です。

```
OCIEnv *envhp;
ub2 ncsid = 2; /* we8dec */
ub2 hdlcsid, hdlncsid;
OraText thename[20];
utext *selstmt = UTF16("SELECT ename FROM emp"); /* make a UTF16 statement */
OCIStmt *stmthp;
OCIDefine *defhp;
OCIError *errhp;
OCIEnvNlsCreate(OCIEnv **envhp, ..., OCI_UTF16ID, ncsid);
...
OCIStmtPrepare(stmthp, ..., selstmt, ...); /* prepare UTF16 statement */
OCIDefineByPos(stmthp, defhp, ..., 1, thename, sizeof(thename), SQLT_CHR,...);
OCINlsEnvironmentVariableGet(&hdlcsid, 0, OCI_NLS_CHARSET_ID, 0, 0);
OCIAttrSet(defhp, ..., &hdlcsid, 0, OCI_ATTR_CHARSET_ID, errhp);
/* change charset id to NLS_LANG setting*/
...
```

関連項目：

- 15-14 ページ「[OCIEnvNlsCreate\(\)](#)」
- 16-176 ページ「[OCINlsEnvironmentVariableGet\(\)](#)」

OCI_ATTR_MAXDATA_SIZE 属性の使用

すべてのバインド・ハンドルは、OCI_ATTR_MAXDATA_SIZE 属性を持っています。この属性で、キャラクタ・セットの必要な変換を行った後、クライアント側バインド・データを格納するためにサーバーに割り当てるバイト数を指定します。

注意： データがサーバーに送られる際に行われるキャラクタ・セットの変換によって、データが拡張または縮小される場合があります。したがって、クライアントでのデータ・サイズがサーバーでのデータ・サイズと同一にならない場合があります。

通常、アプリケーションでは、使用方法に応じて、OCI_ATTR_MAXDATA_SIZE に、列の最大サイズまたは PL/SQL 変数のサイズを設定します。Oracle では、OCI_ATTR_MAXDATA_SIZE の値が変換後のデータを格納するのに十分でない場合、エラーが発生します。

次のシナリオでは、OCI_ATTR_MAXDATA_SIZE 属性の使用例を示します。

■ シナリオ 1: CHAR (ソース・データ) -> 非 CHAR (宛先列)

このケースでは、データに対して暗黙的バインド変換が行われます。このケースでは、ソース・バッファ・サイズに、クライアント側とサーバー側間で行われるキャラクタ・セットの変換の最大値を加算した値を、OCI_ATTR_MAXDATA_SIZE の値として設定することをお勧めします。

■ シナリオ 2: CHAR (ソース・データ) -> CHAR (宛先列) または 非 CHAR (ソース・データ) -> CHAR (宛先列)

これらのケースでは、列のサイズを、OCI_ATTR_MAXDATA_SIZE の値として設定することをお勧めします。

■ シナリオ 3: CHAR (ソース・データ) -> PL/SQL 変数

このケースでは、PL/SQL 変数のサイズを、OCI_ATTR_MAXDATA_SIZE の値として設定することをお勧めします。

OCI_ATTR_MAXCHAR_SIZE 属性の使用

バインドおよび定義ハンドルには、それぞれに関連する OCI_ATTR_MAXCHAR_SIZE 属性があります。アプリケーションでは、この属性を使用して、データをバイト数ではなく文字数で操作できます。

バインドについては、OCI_ATTR_MAXCHAR_SIZE 属性を使用して、アプリケーションがサーバーに確保する文字数を設定し、バインドするデータを格納します。この属性は OCI_ATTR_MAXDATA_SIZE 属性とともに使用され、それぞれ導出されたバイト長から 0（ゼロ）以外の最小値が使用されます。

たとえば、OCI_ATTR_MAXDATA_SIZE が 100 に設定され、OCI_ATTR_MAXCHAR_SIZE が 0（ゼロ）に設定されている場合、サーバーでのデータの変換後に可能な最大サイズは 100 バイトです。ただし、OCI_ATTR_MAXDATA_SIZE が 300 に設定され、OCI_ATTR_MAXCHAR_SIZE が 100 などの 0（ゼロ）以外の値に設定されている場合、キャラクタ・セットに 2 バイトまたは 2 文字がある場合は、割当て可能な最大サイズは 200 バイトです（300 バイトと 2×100 バイトの最小値）。

定義については、クライアント・アプリケーションのバッファに戻ることができる最大文字数を OCI_ATTR_MAXCHAR_SIZE 属性で指定します。導出されたバイト長は、OCIDefineByPos() コールで指定された *maxLength* パラメータを上書きします。

注意： バインド・コールまたは定義コールで指定されたバッファ長は、OCI_ATTR_MAXCHAR_SIZE 属性の値に関係なく、常にバイト数による長さのみなされます。また、この場合、送受信する実際の長さもバイト数になります。

バインド時のバッファの拡張

更新操作または挿入操作は、変数のバインドによって行われます。変数をバインドするときは、バインド・ハンドルの OCI_ATTR_MAXCHAR_SIZE または OCI_ATTR_MAXDATA_SIZE（あるいはその両方）を指定して、サーバーにデータを挿入するときに使用する文字制約とバイト制約を指定します。

これらの属性は、次のように定義します。

- OCI_ATTR_MAXCHAR_SIZE では、サーバー側のバッファで許容される最大文字数を設定します。
- OCI_ATTR_MAXDATA_SIZE では、サーバー側のバッファで許容される最大バイト数を設定します。

いずれの属性も設定されていない場合、OCI では最適な見積りを行ってバッファを拡張します。

OUT バインドまたは PL/SQL バインドの場合は、OCI_ATTR_MAXDATA_SIZE を設定しないでください。

OCI_ATTR_MAXDATA_SIZE は、INSERT 文または UPDATE 文の場合のみ設定してください。

バインド

基礎となる列が文字長セマンティクスを使用して作成されている場合は、OCI_ATTR_MAXCHAR_SIZE を使用して制約を指定することをお勧めします。この場合、実際のバッファに含まれる文字数が OCI_ATTR_MAXCHAR_SIZE で指定した文字数より少なければ、OCI レベルでの制約違反は発生しません。

基礎となる列がバイト長セマンティクスを使用して作成されている場合は、バインド・ハンドルの OCI_ATTR_MAXDATA_SIZE を使用して、サーバーでのバイト制約を指定します。OCI_ATTR_MAXCHAR_SIZE の値も指定している場合は、この制約がサーバー側の受信バッファの割当て時に適用されます。

動的 SQL

動的 SQL の場合は、パラメータ・ハンドルの OCI_ATTR_DATA_SIZE または OCI_ATTR_CHAR_SIZE（あるいはその両方）を取得するための明示的な記述を、バインド・ハンドルの OCI_ATTR_MAXDATA_SIZE 属性および OCI_ATTR_MAXCHAR_SIZE 属性を設定するためのガイドとして使用できます。さらに、OCI_ATTR_MAXDATA_SIZE および OCI_ATTR_MAXCHAR_SIZE を、常に実際の列幅より少ない値（バイト数および文字数）に設定すると安全です。

挿入時のバッファの拡張

次のシナリオは、挿入時のバッファ拡張による予期しない動作の発生を回避する例です。

データベース列に文字長セマンティクスがあり、その列にユーザーが OCIBindByPos() または OCIBindByName() を使用してデータの挿入を試み、OCI_ATTR_MAXCHAR_SIZE のみを 3000 バイトに設定したとします。データベース・キャラクタ・セットは UTF8 で、クライアント・キャラクタ・セットは ASCII です。この場合、クライアント側ではサイズが 3000 バイトのバッファに 3000 文字が入りますが、サーバー側ではバッファが拡張されて 4000 バイトを超えることになります。この場合は、基礎となる列が LONG 型または LOB 型でないかぎり、サーバーでエラーが発生します。この問題を回避するには、OCI_ATTR_MAXDATA_SIZE も 4000 バイトに設定します。これにより、データが 4000 バイトを超えることはありません。

定義時の制約チェック

OCI では、データ列からクライアント・バッファへの選択に事前定義の変数が使用されます。定義バッファに `OCI_ATTR_MAXCHAR_SIZE` 値を設定して、その定義バッファに対して文字長の追加の制約を適用できます。バイト単位のバッファ・サイズはバイト長の制限として機能するため、定義ハンドルに対する `OCI_ATTR_MAXDATA_SIZE` 属性はありません。したがって、通常、`OCIDefineByPos()` コールで指定された定義バッファのサイズをバイト制約として使用できます。

動的 SQL での選択

動的 SQL のバッファ・サイズを指定するときは、切捨てを回避するために、常に暗黙的な記述で `OCI_ATTR_DATA_SIZE` 値を使用します。データベース列が文字長セマンティクスを使用して作成されている場合（`OCI_ATTR_CHAR_USED` 属性によって判明します）は、`OCI_ATTR_MAXCHAR_SIZE` 値を使用して、定義バッファに追加の制約を設定できます。この場合、`OCI_ATTR_MAXCHAR_SIZE` 値を超える文字数がバッファに配置されることはありません。

戻される長さ

次の長さの値は、データベースの文字長セマンティクスに関係なく、常にバイト単位で戻されます。

- `alen` で戻される値、つまり、バインドと定義での実際の長さフィールド
- `VARCHAR` や `LONG VARCHAR` など、特殊なデータ型の前に付けられる長さの値
- 切捨てが発生した場合のためのインジケータ変数の値

この規則の唯一の例外は、`OCI_UTF16ID` の文字列バッファです。バインド / 定義ハンドルに対してキャラクタ・セット ID を `OCI_UTF16ID` で指定すると、前述の長さの値は UTF-16 単位になります。

バインド・コールと定義コールのバッファ・サイズ、`OCIGetPieceInfo()` と `OCISetPieceInfo()` のピース・サイズおよびコールバックは常にバイト単位であることに注意してください。

文字長セマンティクスの互換性に関する一般的な問題

- リリース 8.1 以下のサーバーと通信を行うリリース 1 (9.0.1) 以上のクライアントの場合、OCI_ATTR_MAXCHAR_SIZE はサーバーで認識されないため、値は無視されます。この値のみを指定した場合、OCI では、クライアント側キャラクタ・セットの各文字の最大バイト数に基づいて、対応する OCI_ATTR_MAXDATA_SIZE 値が導出されます。
- リリース 1 (9.0.1) 以上のサーバーと通信を行うリリース 8.1 以下のクライアントの場合は、クライアントで OCI_ATTR_MAXCHAR_SIZE 値を指定できないため、サーバーでは、クライアントが常にバイト長セマンティクスを使用しているものとみなします。これは、クライアントで OCI_ATTR_MAXDATA_SIZE のみを指定した場合に類似しています。

いずれの場合も、サーバーとクライアントは適切な方法で情報を交換できます。

文字変換を伴うバインドと定義のコード例

前述の概念を、次の 2 つの例で説明します。

OCI_ATTR_MAXCHAR_SIZE を使用した挿入と選択のコード例

文字数 N を指定して列が作成されると、次の表では、データベースでの実際の割当てが最大値で想定されます。割り当てられる実際のバイト数は、N の倍数、つまり N の M 倍です。現在、UTF8 では各文字の最大バイト数として、M が 3 に設定されています。

たとえば、次の表の EMP では、ENAME 列が 30 文字に定義され、ADDRESS 列は 80 文字に定義されています。これに対応するデータベースでのバイト長はそれぞれ、 $M \times 30$ または $3 \times 30 = 90$ 、および $M \times 80$ または $3 \times 80 = 240$ になります。

```
...
utext ename[31], address[81];
/* E' <= 30+ 1, D' <= 80+ 1, considering null-termination */
sb2 ename_max_chars = EC=20, address_max_chars = ED=60;
/* EC <= (E' - 1), ED <= (D' - 1) */
sb2 ename_max_bytes = EB=80, address_max_bytes = DB=200;
/* EB <= M * EC, DB <= M * DC */
text *insstmt = (text *)"INSERT INTO EMP(ENAME, ADDRESS) VALUES (:ENAME, :ADDRESS)";
text *selstmt = (text *)"SELECT ENAME, ADDRESS FROM EMP";
...
/* Inserting Column Data */
OCIStmtPrepare(stmthp1, errhp, insstmt, (ub4)strlen((char *)insstmt),
    (ub4)OCI_NTV_SYNTAX, (ub4)OCI_DEFAULT);
OCIBindByName(stmthp1, &bndlp, errhp, (text *)":ENAME",
    (sb4)strlen((char *)":ENAME",
    (dvoid *)ename, sizeof(ename), SQLT_STR, (dvoid *)&insname_ind,
    (ub2 *)alenp, (ub2 *)rcodep, (ub4)maxarr_len, (ub4 *)curelep, OCI_DEFAULT);
/* either */
OCIAttrSet((dvoid *)bndlp, (ub4)OCI_HTYPE_BIND, (dvoid *)&ename_max_bytes,
```

```

        (ub4)0, (ub4)OCI_ATTR_MAXDATA_SIZE, errhp);
/* or */
OCIAttrSet((dvoid *)bndlp, (ub4)OCI_HTYPE_BIND, (dvoid *)&ename_max_chars,
        (ub4)0, (ub4)OCI_ATTR_MAXCHAR_SIZE, errhp);
...
/* Retrieving Column Data */
OCIStmtPrepare(stmthp2, errhp, selstmt, strlen((char *)selstmt),
        (ub4)OCI_NTV_SYNTAX, (ub4)OCI_DEFAULT);
OCIDefineByPos(stmthp2, &dfnlp, errhp, (ub4)1, (dvoid *)ename,
        (sb4)sizeof (ename),
        SQLT_STR, (dvoid *)&selname_ind, (ub2 *)alenp, (ub2 *)rcodep,
        (ub4)OCI_DEFAULT);
/* if not called, byte semantics is by default */
OCIAttrSet((dvoid *)bndlp, (ub4)OCI_HTYPE_DEFINE, (dvoid *)&ename_max_chars,
        (ub4)0,
        (ub4)OCI_ATTR_MAXCHAR_SIZE, errhp);
...

```

UTF-16 のバインドと定義のコード例

CHAR/VARCHAR2 改良型ハンドルまたは NCHAR/NVARCHAR 改良型ハンドルのバインドと定義におけるキャラクタ・セット ID は、対応するバインド・コールと定義コールによって渡されるすべてのデータが UTF-16 (Unicode) エンコーディングであるとみなすように設定できます。UTF-16 を指定するには、OCI_ATTR_CHARSET_ID = OCI_UTF16ID を設定します。

関連項目： 詳細は、A-35 ページのバインド属性
[「OCI_ATTR_CHARSET_ID」](#) の説明および A-38 ページの定義属性
[「OCI_ATTR_CHARSET_ID」](#) の説明を参照してください。

OCI には、UTF-16 データのバインドと定義を容易にするため、**utext** と呼ばれる **typedef** が用意されています。**utext** の内部表現は、16 ビットの符号なし整数 (**ub2**) です。**wchar_t** データ型のコード体系が UTF-16 (符号なし 16 ビット値) に準拠しているプラットフォームでは、キャスト演算子を使用して **utext** を **wchar_t** データ型に容易に変換できます。

UTF-16 データの場合でも、バインド・コールと定義コールでのバッファ・サイズはバイト単位とみなされます。新しい **utext** データ型は、入出力データのバッファとして使用してください。

次の擬似コードは、UTF-16 データのバインドと定義の例です。

```
...
OCISstmt *stmthp1, *stmthp2;
OCIDefine *dfnlp, *dfnp2;
OCIBind *bndlp, *bnd2p;
text *insstmt=
    (text *) "INSERT INTO EMP(ENAME, ADDRESS) VALUES (:ename, :address)";
text *selname =
    (text *) "SELECT ename, address FROM emp";
utext ename[21]; /* Name - UTF-16 */
utext address[51]; /* Address - UTF-16 */
ub2 csid = OCI_UTF16ID;
sb2 ename_col_len = 20;
sb2 address_col_len = 50;
...
/* Inserting UTF-16 data */
OCISstmtPrepare (stmthp1, errhp, insstmt, (ub4)strlen((char *)insstmt),
    (ub4)OCI_NTV_SYNTAX, (ub4)OCI_DEFAULT));
OCIBindByName (stmthp1, &bndlp, errhp, (text*)" :ENAME",
    (sb4)strlen((char *)":ENAME"),
    (dvoid *) ename, sizeof(ename), SQLT_STR,
    (dvoid *)&insname_ind, (ub2 *) 0, (ub2 *) 0, (ub4) 0,
    (ub4 *)0, OCI_DEFAULT);
OCIAttrSet ((dvoid *) bndlp, (ub4) OCI_HTYPE_BIND, (dvoid *) &csid,
    (ub4) 0, (ub4)OCI_ATTR_CHARSET_ID, errhp);
OCIAttrSet((dvoid *) bndlp, (ub4) OCI_HTYPE_BIND, (dvoid *) &ename_col_len,
    (ub4) 0, (ub4)OCI_ATTR_MAXDATA_SIZE, errhp);
...
/* Retrieving UTF-16 data */
OCISstmtPrepare (stmthp2, errhp, selname, strlen((char *) selname),
    (ub4)OCI_NTV_SYNTAX, (ub4)OCI_DEFAULT);
OCIDefineByPos (stmthp2, &dfnlp, errhp, (ub4)1, (dvoid *)ename,
    (sb4)sizeof(ename), SQLT_STR,
    (dvoid *)0, (ub2 *)0, (ub2 *)0, (ub4)OCI_DEFAULT);
OCIAttrSet ((dvoid *) dfnlp, (ub4) OCI_HTYPE_DEFINE, (dvoid *) &csid,
    (ub4) 0, (ub4)OCI_ATTR_CHARSET_ID, errhp);
...
```

PL/SQL REF CURSOR および NESTED TABLE

OCI には、PL/SQL REF CURSOR および NESTED TABLE を、バインドおよび定義する機能があります。アプリケーションでは、これらの型の変数をハンドルおよび定義するために、文ハンドルを使用できます。次の例で、この PL/SQL ブロックを考えてみます。

```
static const text *plsql_block = (text *)
    "begin \
        OPEN :cursor1 FOR SELECT empno, ename, job, mgr, sal, deptno \
            FROM emp_rc WHERE job=:job ORDER BY empno; \
        OPEN :cursor2 FOR SELECT * FROM dept_rc ORDER BY deptno; \
    end;";
```

アプリケーションは、OCIHandleAlloc() をコールして、バインドのための文ハンドルを割り当てます。そして、次のコード例の中で、`:cursor1` が `stm2p` にバインドされるように、プレースホルダ `:cursor1` を文ハンドルにバインドします。ハンドル割当てコードは、この例に含まれていないことに注意してください。

```
err = OCISstmtPrepare (stm1p, errhp, (text *) nst_tab, strlen(nst_tab),
    OCI_NTV_SYNTAX, OCI_DEFAULT);
...
err = OCIBindByName (stm1p, (OCIBind **) bndp, errhp,
    (text *) ":cursor1", (sb4)strlen((char *) ":cursor1"),
    (dvoid *)&stm2p, (sb4) 0,  SQLT_RSET, (dvoid *)0,
    (ub2 *)0, (ub2 *)0, (ub4)0, (ub4 *)0, (ub4)OCI_DEFAULT);
```

このコードでは、`stm1p` は PL/SQL ブロックの文ハンドルです。一方、`stm2p` は、後のデータ検索のための REF CURSOR としてバインドされる文ハンドルです。SQLT_RSET の値は、`dtv` パラメータに渡されます。

別の例として、次のコードを考えてみます。

```
static const text *nst_tab = (text *)
    "SELECT ename, CURSOR(SELECT dname, loc FROM dept_rc) \
    FROM emp_rc WHERE ename = 'LOCKE'";
```

この場合、2 番目の位置が NESTED TABLE で、次のように、OCI アプリケーションで文ハンドルとして定義できます。ハンドル割当てコードは、この例に含まれていないことに注意してください。

```
err = OCISstmtPrepare (stm1p, errhp, (text *) nst_tab, strlen(nst_tab),
    OCI_NTV_SYNTAX, OCI_DEFAULT);
...
err = OCIDefineByPos (stm1p, (OCIDefine **) dfn2p, errhp, (ub4)2,
    (dvoid *)&stm2p,
    (sb4)0, SQLT_RSET, (dvoid *)0, (ub2 *)0,
    (ub2 *)0, (ub4)OCI_DEFAULT);
```

実行後、行を `stm2p` にフェッチすると、有効な文ハンドルになります。

注意： 複数の REF カーソルを取り出した場合は、`stm2p` へのフェッチをいつ行うか注意する必要があります。最初の REF カーソルをフェッチすると、そのデータを取り出すためにフェッチを実行できます。ただし、2 番目の REF カーソルを `stm2p` にフェッチすると、最初の REF カーソルからのデータにはアクセスできなくなります。

ランタイム・データ割当てとピース単位操作

OCI を使用すると、データをピース単位で挿入、更新およびフェッチできます。また、配列の挿入または更新の場合は、バインド値の静的配列のかわりに、OCI を使用して動的にデータを提供できます。非常に大きな列は、小さなサイズのチャンク（かたまり）が連続したものとして挿入または取出しができます。これによって、クライアント側のメモリー所要量を最小限にできます。

個々のピースのサイズは、実行時にアプリケーションにより判断されます。それぞれのピースは、他のピースと同サイズにすることも、違うサイズにすることもできます。

OCI のピース単位機能は、文字列やバイナリ・データの膨大なブロックで操作（CLOB、BLOB、LONG、RAW、LONG RAW などのデータを格納するデータベース列に関係する操作など）を行う際に特に有効です。

ピース単位操作に有効なデータ型

一部のデータ型のみが、ピース単位で操作できます。次のデータ型は、OCI アプリケーションでピース単位のフェッチ、挿入または更新を実行できます。

- VARCHAR2
- STRING
- LONG
- LONG RAW
- RAW
- CLOB
- BLOB

LOB および FILE 操作の中にも、データの読み書きに対し、ピース単位セマンティクスを提供しているものがあります。

すべてのデータ型に対してこの機能を使用する別な方法として、配列の挿入または更新のためにデータを動的に提供する方法があります。ただし、ピース単位操作をサポートしないデータ型に対しては、コールバックで、そのコールバックの *piecep* パラメータの `OCI_ONE_PIECE` を常に指定する必要があります。

関連項目： これらの操作の詳細は、16-76 ページの「[OCILobWrite\(\)](#)」および 16-69 ページの「[OCILobRead\(\)](#)」の説明を参照してください。
[OCILobWrite\(\)](#) および [OCILobRead\(\)](#) でコールバックを使用したストリームの詳細は、7-13 ページの「[LOB 読み込みおよび書き込みコールバック](#)」を参照してください。

LOB に対するバインドおよび定義

次の関数は、CLOB 列に対して、`SQLT_CHR` (`VARCHAR2`)、`SQLT_LNG` (`LONG`) および `SQLT_CLOB` (`CLOB`) を受け入れます。また、BLOB 列に対しては、`SQLT_LBI` (`LONG RAW`)、`SQLT_BIN` (`RAW`) および `SQLT_BLOB` (`BLOB`) の各データ型を受け入れます。

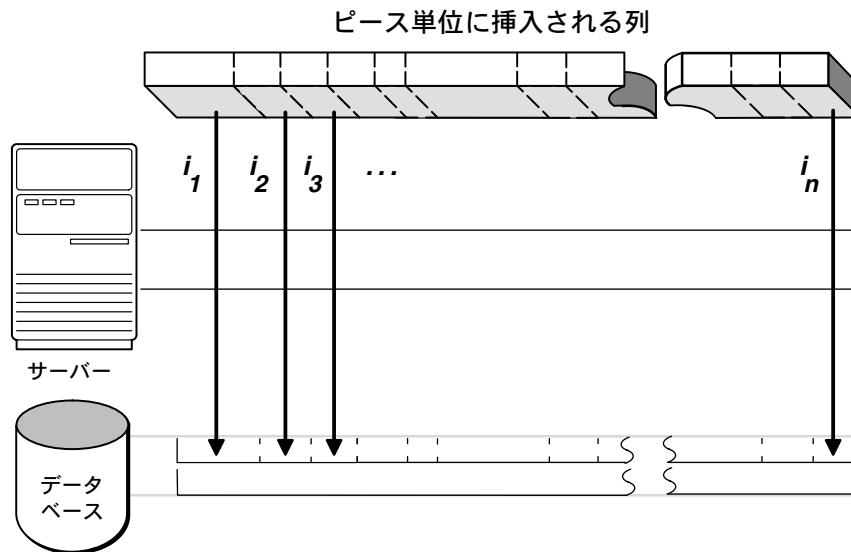
- `OCIDefineByPos()`
- `OCIBindByName()`
- `OCIBindByPos()`

たとえば、CLOB 列に対するデータ型として `SQLT_CHR` が指定されている場合、`OCIStmtFetch()` では CLOB データをフェッチします。ただし、CLOB 列に対するデータ型として `SQLT_CLOB` が指定されている場合、`OCIStmtFetch()` はリリース 1 (9.0.1) より前と同様に機能します。この関数によって CLOB ロケータが戻され、`OCILobRead()` をコールすると、CLOB データを読み込むことができます。

ピース単位操作の種類

[図 5-4 「列のピース単位の挿入」](#) では、挿入操作 (`i1`、`i2`、`i3...in`) を連続して行うことによって、データベース表に単一の列をピース単位で挿入しています。この例では、挿入するピースのサイズは一定ではありません。

図 5-4 列のピース単位の挿入



ピース単位操作は、次の 2 通りの方法で実行できます。

- ポーリング・パラダイムに基づいてピース単位操作を実行するには、リリース 7.3 で行ったように、OCI ライブラリで提供されているコールを使用します。
- 必要な情報およびデータ・ブロックを提供するには、ユーザー定義のコールバック関数を使用します。

OCIBindByPos() コールまたは OCIBindByName() コールの *mode* パラメータを OCI_DATA_AT_EXEC に設定すると、OCI アプリケーションでは実行時に INSERT または UPDATE のためのデータが動的に提供されます。

同様に、OCIDefineByPos() コールの *mode* パラメータを OCI_DYNAMIC_FETCH に設定した場合は、アプリケーションが、フェッチ時にデータを受け取るための割当てスペースを動的に提供することを示します。

どちらの場合でも、コールバック関数またはピース単位操作のいずれかの方法によって、INSERT、UPDATE または FETCH のランタイム情報を提供できます。コールバックを使用するときはコールバックを登録するために、追加のバインド・コールまたは定義コールが必要です。

次項以降では、挿入、更新およびフェッチのためのデータ割当てとピース単位操作に関する特有の情報について説明します。

注意： ピース単位操作は、SQL 文のみでなく、PL/SQL ブロックでも有効です。

実行時の INSERT または UPDATE データの提供

OCIBindByPos() コールまたは OCIBindByName() コールで OCI_DATA_AT_EXEC を指定すると、value_sz パラメータによって、実行時に提供できるデータの合計サイズが定義されます。アプリケーションでは、実行時 IN データ・バッファを、要求に応じて操作完了に必要な回数のみ OCI ライブラリに提供する用意をしておく必要があります。割り当てたバッファは、不要になった際にクライアント側で解放する必要があります。

ランタイム・データは、次の 2 つの方法のどちらかを使用して提供されます。

- OCIBindDynamic() 関数を使用したコールバックを定義できます。このコールバックは実行時にコールすると、データのピースまたはデータ全体を戻します。
- コールバックを 1 つも定義していない場合は、SQL 文を処理する OCISmtExecute() コールは、OCI_NEED_DATA エラー・コードを戻します。クライアント・アプリケーションは、次に OCISmtSetPieceInfo() コールを使用して、IN/OUT データ・バッファまたはピースを提供します。OCISmtGetPieceInfo() コールにより、どのバインドおよびピースが使用されているかについての情報が提供されます。

ピース単位の挿入または更新の実行

OCI 環境が初期化され、データベース接続およびセッションが確立すると、SQL 文または PL/SQL 文を準備するコールと入力値をバインドするコールが実行され、ピース単位挿入が開始されます。ユーザー定義のコールバックではなく、標準の OCI コールを使用するピース単位操作では、OCIBindDynamic() をコールする必要はありません。

注意： ピース単位操作の一部でない文にある追加のバインド変数は、データ型に応じて、追加のバインド・コールが必要な場合があります。

文の準備およびバインドの次に、アプリケーションで OCISmtExecute()、OCISmtGetPieceInfo() および OCISmtSetPieceInfo() を連続してコールし、ピース単位操作を完了します。それぞれの OCISmtExecute() コールは、次に実行する処理を決定する値を戻します。一般的には、アプリケーションで次のピースの挿入を指示する値を取り出し、そのピースをバッファに移してから挿入を行います。最後のピースを挿入すると操作は完了します。

挿入バッファは任意のサイズにでき、実行時に提供されることに注意してください。また、挿入する各ピースは、同じサイズである必要はありません。挿入するピースのサイズは、それぞれ OCISmtSetPieceInfo() コールで設定します。

注意： すべての挿入で同じサイズのピースが使用され、挿入されるデータのサイズがそのピースのサイズで均等に割り切れない場合は、挿入される最後のピースは、その前に挿入されたピースよりも小さくなります。たとえば、10,050,036 バイトの長さのデータの値を、それぞれ 500 バイトのチャンクで挿入すると、最後に残るピースは 36 バイトにしかありません。プログラマはこのことを考慮して、最後の `OCIStmtSetPieceInfo()` コールで小さいサイズを指定する必要があります。

次のステップは、ピース単位の挿入または更新の実行に関する手順の概略です。5-49 ページの図 5-5「ピース単位挿入実行の手順」にこの手順を示しています。

ステップ 1. OCI 環境を初期化して必要なハンドルを割り当てます。サーバーに接続してユーザーを認証し、文の要求を準備します。これらのステップは 2-20 ページの「OCI プログラミング・ステップ」で説明しています。

ステップ 2. `OCIBindByName()` または `OCIBindByPos()` を使用して、プレースホルダをバインドします。ここでは、使用するピースの実際のサイズを指定する必要はありません。実行時に提供できるデータの全体のサイズを指定します。

7.x アップグレードの注意： 以前は `obindps()` ルーチンおよび `ogetpi()` ルーチンの一部であったコンテキスト・ポインタは、リリース 8.0 以上では存在しません。クライアント独自のコンテキストを提供する場合は、コールバック方式を使用することができます。

ステップ 3. `OCIStmtExecute()` の最初のコールを実行します。この時点では、データは実際には挿入されていません。アプリケーションに `OCI_NEED_DATA` エラー・コードが戻されます。

その他の値が戻された場合は、エラーが起きたことを示しています。

ステップ 4. `OCIStmtGetPieceInfo()` をコールし、挿入する必要があるピースの情報を取り出します。`OCIStmtGetPieceInfo()` のパラメータには、必要なピースが最初のピース (`OCI_FIRST_PIECE`) なのか、後続のピース (`OCI_NEXT_PIECE`) なのかを示す値を戻すポインタが含まれています。

ステップ 5. アプリケーションは、挿入するデータのピースをバッファに移し、`OCIStmtSetPieceInfo()` をコールします。`OCIStmtSetPieceInfo()` に渡されるパラメータには、次のポインタと値が含まれています。

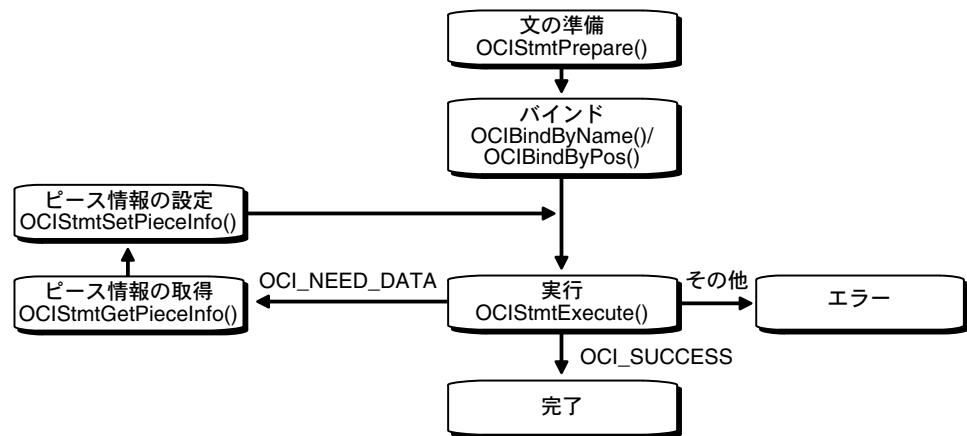
- ピースへのポインタ
- ピースの長さへのポインタ

- 次のピースかどうかを示す値
 - * 最初のピース (OCI_FIRST_PIECE)
 - * 中間のピース (OCI_NEXT_PIECE)
 - * 最後のピース (OCI_LAST_PIECE)

ステップ 6. 再度、OCIStmtExecute() をコールします。ステップ 5 で OCI_LAST_PIECE が示され、OCIStmtExecute() が OCI_SUCCESS を戻した場合は、すべてのピースが正常に挿入されています。OCIStmtExecute() で OCI_NEED_DATA が戻された場合は、ステップ 3 に戻り、次の挿入を行います。OCIStmtExecute() がこれ以外の値を戻した場合は、エラーが発生しています。

最後のピースが正常に挿入されると、ピース単位操作は完了します。この完了は、最後の OCIStmtExecute() コールによる OCI_SUCCESS 戻り値によって示されます。

図 5-5 ピース単位挿入実行の手順



ピース単位更新も同様の方法で実行します。ピース単位更新操作では、更新対象のデータが挿入バッファに移され、OCIStmtExecute() がコールされて更新が実行されます。

注意： ピース単位操作に関する他の重要な情報は、5-52 ページの「[コールバックなしのピース単位操作に関する追加情報](#)」を参照してください。

PL/SQL でのピース単位操作

OCI アプリケーションでは、前に概説した方法と似た方法で、IN、OUT および IN/OUT バインド変数について、PL/SQL でのピース単位操作を実行できます。PL/SQL 文のすべてのプレースホルダは、定義ではなくバインドされることに注意してください。

OCIBindDynamic() のコールでは、OUT パラメータまたは IN/OUT パラメータについて、適切なコールバックを指定します。

実行時のフェッチ情報の提供

mode パラメータに OCI_DYNAMIC_FETCH を設定して OCIDefineByPos() をコールすると、アプリケーションでは、フェッチ時のデータ・バッファに関する情報を指定できます。また、OCIDefineDynamic() をコールして、データ・バッファに関する情報を取得するために呼び出すコールバック関数を設定する必要がある場合があります。

ランタイム・データは、次の 2 つの方法のいずれかで提供されます。

- OCIDefineDynamic() コールを使用したコールバックを定義できます。実行時に提供されるデータの最大サイズは、value_sz パラメータによって定義されます。クライアント・ライブラリで、フェッチ済みデータを戻すためのバッファが必要な場合は、このコールバックが呼び出され、ピースまたは全体のデータを戻すための実行時バッファが提供されます。
- コールバックが 1 つも定義されていない場合は、OCI_NEED_DATA エラー・コードが戻り、クライアント・アプリケーションで OCISstmtSetPieceInfo() コールを使用して、OUT データ・バッファまたはピースを提供します。OCISstmtGetPieceInfo() コールにより、どの定義およびピースが含まれるかについての情報が提供されます。

関連項目：ピース単位操作に有効なデータ型の詳細は、5-44 ページの「[ピース単位操作に有効なデータ型](#)」を参照してください。

ピース単位フェッチの実行

OCI 環境が初期化され、データベース接続およびセッションが確立すると、SQL 文または PL/SQL 文を準備するコールと出力変数を定義するコールが実行され、ピース単位フェッチが開始されます。ユーザー定義のコールバックではなく、標準の OCI コールを使用するピース単位操作では、OCIDefineDynamic() をコールする必要はありません。

文の準備と定義の次に、アプリケーションで OCISstmtFetch()、OCISstmtGetPieceInfo() および OCISstmtSetPieceInfo() を連続してコールし、ピース単位操作を完了します。それぞれの OCISstmtFetch() コールは、次に実行する処理を決定するための値を戻します。一般的には、アプリケーションで次のピースのフェッチを指示する値を取り出し、そのピースをバッファにフェッチします。最後のピースをフェッチすると操作は完了します。

フェッチ・バッファは、任意のサイズにすることができます。また、フェッチする各ピースは、同じサイズである必要はありません。ただし、最後にフェッチするサイズは、残っている最後のピースに一致する必要があります。フェッチする各ピースのサイズは、`OCIStmtSetPieceInfo()` コールでそれぞれ設定します。

次のステップは、ピース単位に行をフェッチする方法の概略です。

関連項目： [図 5-6「ピース単位フェッチ実行のステップ」](#) にこのプロセスを示しています。

ステップ 1. OCI 環境を初期化して必要なハンドルを割り当てます。データベースに接続してユーザーを認証します。文を準備して実行します。これらのステップは 2-20 ページの「[OCI プログラミング・ステップ](#)」で説明しています。

ステップ 2. `mode` を `OCI_DYNAMIC_FETCH` に設定し、`OCIDefineByPos()` を使用して出力変数を定義します。この時点では、使用するピースの実際のサイズを指定する必要はありません。実行時にフェッチするデータの全体のサイズを指定します。

7.x アップグレードの注意： `odefins()` ルーチンおよび `ogetpi()` ルーチンの一部であったコンテキスト・ポインタは、リリース 8.x 以上では存在しません。クライアント独自のコンテキストを提供する場合は、コールバック方式を使用することができます。

ステップ 3. `OCIStmtFetch()` の最初のコールを実行します。この時点では、データは実際には取り出されていません。アプリケーションに `OCI_NEED_DATA` エラー・コードが戻されます。

その他の値が戻された場合は、エラーが起きています。

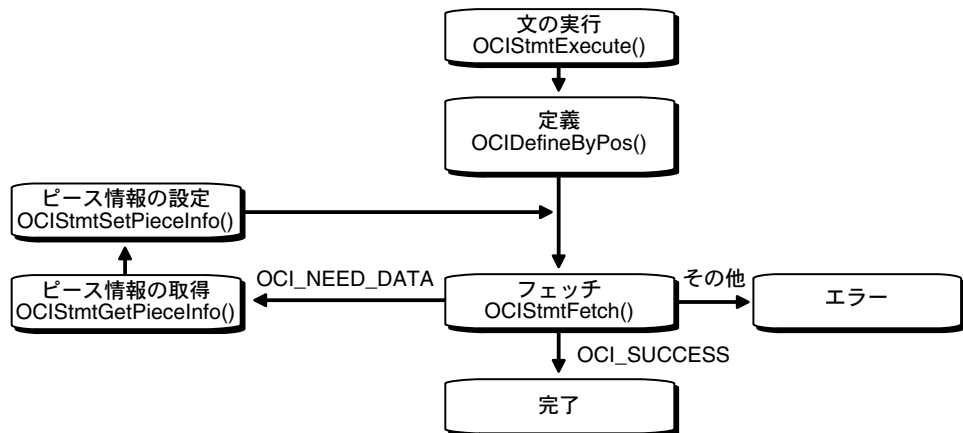
ステップ 4. `OCIStmtGetPieceInfo()` をコールし、フェッチするピースに関する情報を取得します。`piecep` パラメータは、そのピースが最初のピース (`OCI_FIRST_PIECE`) であるか、その後続のピース (`OCI_NEXT_PIECE`) であるか、または最後のピース (`OCI_LAST_PIECE`) であるかを示します。

ステップ 5. `OCIStmtSetPieceInfo()` をコールし、ピースをフェッチするバッファを指定します。

ステップ 6. 再度、`OCIStmtFetch()` をコールし、実際のピースを取り出します。`OCIStmtFetch()` で `OCI_SUCCESS` が戻された場合は、すべてのピースが正常にフェッチされています。`OCIStmtFetch()` で `OCI_NEED_DATA` が戻された場合は、ステップ 4 に戻って次のピースを処理します。その他の値が戻された場合は、エラーが発生しています。

最後の `OCIStmtFetch()` コールが `OCI_SUCCESS` の値を戻すと、ピース単位フェッチが完了します。

図 5-6 ピース単位フェッチ実行のステップ



コールバックなしのピース単位操作に関する追加情報

ピース単位のフェッチおよび挿入では、操作を正常に完了するために必要なコールの順序を理解することが重要です。ピース単位挿入では、コールバックが使用されない場合、挿入するピースの数よりも1回多く `OCIStmtExecute()` をコールする必要があることを、特に注意してください。これは、最初の `OCIStmtExecute()` コールが、単に、最初のピースの挿入が必要であることを示す値を返すのみであるからです。結果として、 n 個のピースを挿入する場合は、`OCIStmtExecute()` を合計 $n+1$ 回コールする必要があります。

同様に、ピース単位フェッチを実行するときは、フェッチするピースの数より1回多く `OCIStmtFetch()` をコールする必要があります。

PL/SQL 索引付き表をバインドする場合は、`OCIStmtGetPieceInfo()` コール中に、その表のカレント索引へのポインタを取り出すことができます。

スキーマ・メタデータの記述

この章では、[OCIDescribeAny\(\)](#) 関数を使用して、スキーマ要素に関する情報を取得する方法を説明します。この章は、次の項目で構成されています。

- [スキーマ・メタデータの記述](#)
- [OCIDescribeAny\(\) の使用](#)
- [OCIDescribeAny\(\) の使用例](#)

スキーマ・メタデータの記述

この章では、スキーマ・オブジェクトを記述するための、OCIDescribeAny() 関数の使用方法を説明します。

関連項目：

- 選択リスト項目の記述の詳細は、4-12 ページの「[選択リスト項目の記述](#)」を参照してください。
- [OCIDescribeAny\(\)](#) コールとそのパラメータの追加情報は、15-91 ページの関数の説明を参照してください。

OCIDescribeAny() の使用

OCIDescribeAny() 関数を使用すると、次のスキーマ・オブジェクトのいずれか、およびそのサブスキーマ・オブジェクトを明示的に記述できます。

- 表とビュー
- シノニム
- プロシージャ
- ファンクション
- パッケージ
- 順序
- コレクション
- 型
- スキーマ
- データベース

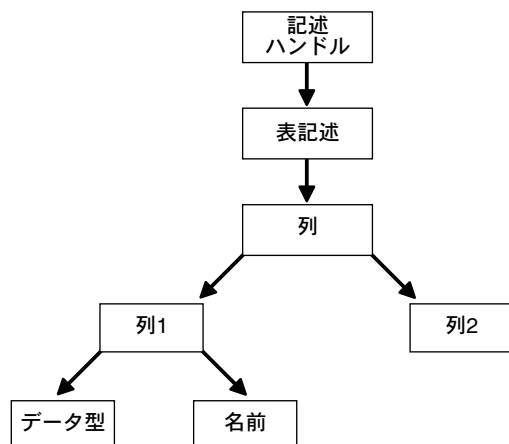
他のスキーマ要素（プロシージャ / ファンクションの引数、列、型属性および型メソッド）の情報は、前述のスキーマ・オブジェクトのいずれかの記述、あるいはサブスキーマ・オブジェクトの明示的な記述によって取得できます。

アプリケーションで表を記述する場合は、その表の列の情報も取り出すことができます。さらに、OCIDescribeAny() では、表の列、関数のパッケージ、型のフィールドなどのサブスキーマ・オブジェクトの名前がわかっている場合、それらのサブスキーマ・オブジェクトを直接記述できます。

OCIDescribeAny() コールには、パラメータの 1 つとして記述ハンドルが必要です。記述ハンドルは、[OCIHandleAlloc\(\)](#) へのコールで事前に割り当ててください。アプリケーションでは、OCIDescribeAny() をコールした後、記述ハンドルから記述オブジェクトに関する情報を取り出せます。

OCIDescribeAny() で戻された情報は、ツリーのように階層的に編成されています。たとえば、図 6-1 は特定の表の記述の編成方法を示しています。

図 6-1 OCIDescribeAny() の表記述



OCIDescribeAny() によって戻される記述ハンドルには、このような記述ツリーを指し示す属性 OCI_ATTR_PARAM があります。ツリーの各ノードには、そのノードに関連する属性と、再帰的な記述ハンドルのように情報を含むサブツリーを指し示す属性があります。列リストなどのリストの要素のように、すべての属性が同じ構造である場合、それらの属性をパラメータと呼びます。この章では、ハンドルとパラメータは同じ意味で使用します。ノードに関連付けられた属性は OCIAttrGet() によって、パラメータは OCIParamGet() によって戻されます。

たとえば、表の記述ハンドルについての OCIAttrGet() は、列リストの情報のハンドルを戻します。アプリケーションでは、OCIParamGet() を使用して、列リスト内の特定の列の列記述へのハンドルを取り出します。列記述子へのハンドルを OCIAttrGet() へ受け渡すことによって、名前やデータ型などの列に関する詳細を得ることができます（前述の図の左側をたどるとそれがわかります）。

後続の OCIAttrGet() コールまたは OCIParamGet() コールでは、すべての記述が OCIDescribeAny() によってクライアント側のキャッシュに保存されているので、追加のネットワーク・ラウンドトリップは発生しません。

制限

OCIDescribeAny() コールは基本情報に戻される情報を制限し、それが別の記述になる場合、ノードの拡張を停止します。たとえば、表の列がオブジェクト型である場合、OCI は型を記述しているサブツリーを戻しません。その情報は別の記述によって取得できるからです。

型および属性の注意

記述操作を行う際には、次の注意があります。

データ型コードの注意

関連項目： 型コードの詳細（OCI_ATTR_TYPECODE 属性に戻される OCI_TYPECODE 値、OCI_ATTR_DATA_TYPE 属性に戻される SQLT 型コードなど）は、3-29 ページの「[型コード](#)」を参照してください。

OCI_ATTR_TYPECODE は、CREATE TYPE 文を使用して新しい型が作成されたときに、ユーザーが指定した型を表す型コードを戻します。これらの型コードは、数え上げ可能な型の **OCITypeCode** で、OCI_TYPECODE 定数で表されます。内部 PL/SQL 型（ブール、索引付き表）はサポートされません。

OCI_ATTR_DATA_TYPE は、データベースの列に格納されるデータ型を表す型コードを戻します。これらは、Oracle の以前のバージョンに戻される記述値に似ています。これらの値は、SQLT 定数（**ub2** 値）で表されます。BOOLEAN 型は SQLT_BOL を戻します。

型の記述上の注意

型オブジェクトを記述するには、OCI プロセスをオブジェクト・モードで初期化する必要があります。

```
/* Initialize the OCI Process */
if (OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0,
                  (dvoid * (*)(dvoid *, size_t)) 0,
                  (dvoid * (*)(dvoid *, dvoid *, size_t))0,
                  (void *) (dvoid *, dvoid *) 0 ))
{ (void) printf("FAILED: OCIInitialize()\n");
  return OCI_ERROR; }
```

関連項目： この関数の詳細は、15-18 ページの「[OCIInitialize\(\)](#)」の説明を参照してください。

暗黙的記述および明示的記述の注意

OCIStmtExecute() による暗黙的記述および OCIDescribeAny() による明示的記述を使用して、列属性 OCI_ATTR_PRECISION を戻すことができます。暗黙的記述を使用する場合は、精度を **sb2** に設定する必要があります。明示的記述を使用する場合は、精度をプレースホルダに対して **ub1** に設定する必要があります。この設定は、ディクショナリ内の精度のデータ型と一致させるために必要です。

OCI_ATTR_LIST_ARGUMENTS についての注意

型メソッド用の OCI_ATTR_LIST_ARGUMENTS 属性は、そのメソッドの 2 番目のレベルの引数を表します。

たとえば、次のように、レコード my_type と、型 my_type の引数が指定されたプロシージャ my_proc を指定します。

```
my_type record(a number, b char)
my_proc (my_input my_type)
```

OCI_ATTR_LIST_ARGUMENTS 属性は、my_type レコードの引数 a と b に適用されます。

パラメータ属性

パラメータは、OCIParamGet() によって戻されます。パラメータは、異なる種類のオブジェクトや情報を記述できます。パラメータの属性は、そのパラメータに含まれる記述の型によって異なります。これらは、型固有の属性です。この項では、様々なパラメータに所属する属性およびハンドルについて説明します。

次の表は、すべてのパラメータに所属する属性のリストです。

表 6-1 すべてのパラメータに所属する属性

属性	説明	属性データ型
OCI_ATTR_NUM_PARAMS	パラメータの数。	ub2
OCI_ATTR_OBJ_ID	オブジェクト ID またはスキーマ ID。	ub4
OCI_ATTR_OBJ_NAME	スキーマ内のデータベース名またはオブジェクト名。	text*
OCI_ATTR_OBJ_SCHEMA	そのオブジェクトが存在しているスキーマの名前。	text*

表 6-1 すべてのパラメータに所属する属性（続き）

属性	説明	属性データ型
OCI_ATTR_PTYPE	パラメータによって記述される情報の種類。可能な値は次のとおりです。 OCI_PTYPE_TABLE – 表 OCI_PTYPE_VIEW – ビュー OCI_PTYPE_PROC – プロシージャ OCI_PTYPE_FUNC – ファンクション OCI_PTYPE_PKG – パッケージ OCI_PTYPE_TYPE – 型 OCI_PTYPE_TYPE_ATTR – 型の属性 OCI_PTYPE_TYPE_COLL – コレクション型情報 OCI_PTYPE_TYPE_METHOD – 型のメソッド OCI_PTYPE_SYN – シノニム OCI_PTYPE_SEQ – 順序 OCI_PTYPE_COL – 表またはビューの列 OCI_PTYPE_ARG – ファンクションまたはプロシージャの引数 OCI_PTYPE_TYPE_ARG – 型メソッドの引数 OCI_PTYPE_TYPE_RESULT – メソッドの結果 OCI_PTYPE_LIST – 表およびビューの列のリスト、ファンクションおよびプロシージャの引数リスト、またはパッケージのサブプログラム・リスト OCI_PTYPE_SCHEMA – スキーマ OCI_PTYPE_DATABASE – データベース	ub1
OCI_ATTR_TIMESTAMP	この記述の基礎となるオブジェクトのタイムスタンプ（Oracle 日付書式）。	ub1 *

これ以降の各項では、様々なパラメータの型に特有の属性およびハンドルのリストを示します。

OCI_PTYPE_TABLE 型または OCI_PTYPE_VIEW 型

表またはビューのパラメータ（OCI_PTYPE_TABLE 型または OCI_PTYPE_VIEW 型）の場合、型特有の属性は次のとおりです。

表 6-2 表またはビューに所属する属性

属性	説明	属性データ型
OCI_ATTR_OBJID	オブジェクト ID。	ub4
OCI_ATTR_NUM_COLS	列の数。	ub2
OCI_ATTR_LIST_COLUMNS	列リスト（OCI_PTYPE_LIST 型）。	dvoid *
OCI_ATTR_REF_TDO	エクステンント表の場合の、基礎型の TDO への REF。	OCIRef *
OCI_ATTR_IS_TEMPORARY	表が一時表であるかどうか。	ub1
OCI_ATTR_IS_TYPED	表に型が指定されているかどうか。	ub1
OCI_ATTR_DURATION	一時表の継続時間。可能な値は次のとおりです。 OCI_DURATION_SESSION — セッション OCI_DURATION_TRANS — トランザクション OCI_DURATION_NULL — 非一時表	OCIDuration

その他に、表に所属する次の属性があります。

表 6-3 表に特有の属性

属性	説明	属性データ型
OCI_ATTR_DBA	セグメント・ヘッダーのデータ・ブロック・アドレス。	ub4
OCI_ATTR_TABLESPACE	表が存在する表領域。	word
OCI_ATTR_CLUSTERED	表がクラスタ化表であるかどうか。	ub1
OCI_ATTR_PARTITIONED	表がパーティション表であるかどうか。	ub1
OCI_ATTR_INDEX_ONLY	表が索引のみの表であるかどうか。	ub1

プロシージャ/ファンクション/サブプログラム属性

プロシージャまたはファンクションのパラメータ（OCI_PTYPE_PROC 型または OCI_PTYPE_FUNC 型）の場合、型特有の属性は次のとおりです。

表 6-4 プロシージャまたはファンクションに所属する属性

属性	説明	属性データ型
OCI_ATTR_LIST_ARGUMENTS	引数リスト。6-19 ページの「 リスト属性 」を参照。	dvoid *
OCI_ATTR_IS_INVOKER_RIGHTS	プロシージャまたはファンクションが実行者権限 プロシージャまたは実行者権限ファンクションで あるかどうか。	ub1

次の属性は、パッケージ・サブプログラムでのみ定義されます。

表 6-5 パッケージ・サブプログラムに特有の属性

属性	説明	属性データ型
OCI_ATTR_NAME	プロシージャまたはファンクションの名前。	text *
OCI_ATTR_OVERLOAD_ID	ID 番号のオーバーロード（プロシージャまたは ファンクションがパッケージの一部で、オーバー ロードされる場合）。戻される値が、PL/SQL ファンクションまたはプロシージャの直接問合せ とは異なる場合があります。	ub2

パッケージ属性

パッケージのパラメータ（OCI_PTYPE_PKG 型）の場合、型特有の属性は次のとおりです。

表 6-6 パッケージに所属する属性

属性	説明	属性データ型
OCI_ATTR_LIST_SUBPROGRAMS	サブプログラム・リスト。6-19 ページの「 リスト属性 」を参照。	dvoid *
OCI_ATTR_IS_INVOKER_RIGHTS	パッケージが実行者権限パッケージであるかどうか。	ub1

型属性

型のパラメータの場合（OCI_PTYPE_TYPE 型）、その属性は表 6-7 にリストされているとおりです。これらの属性は、OCIInitialize() のコールの際に、アプリケーションによって OCI_OBJECT モードで OCI プロセスが初期化される場合にのみ有効です。

表 6-7 型に所属する属性

属性	説明	属性データ型
OCI_ATTR_REF_TDO	列の型がオブジェクト型の場合は、その型の型記述子オブジェクトのメモリー内 REF を戻します。OCIRef 用の領域が確保されていない場合は、領域がキャッシュ内に暗黙的に割り当てられます。次に、コール元は OCIObjectPin() を使用して TDO を確保できます。	OCIRef *
OCI_ATTR_TYPECODE	型コード。6-4 ページの「データ型コードの注意」を参照。現在は、OCI_TYPECODE_OBJECT または OCI_TYPECODE_NAMEDCOLLECTION のみが使用できます。	OCITypeCode
OCI_ATTR_COLLECTION_TYPECODE	型がコレクションの場合はコレクションの型コードで、それ以外の場合は無効です。6-4 ページの「データ型コードの注意」を参照。現在は、OCI_TYPECODE_VARRAY または OCI_TYPECODE_TABLE のみが使用できます。この属性が非コレクション型に対して問合せされた場合は、エラーが戻されます。	OCITypeCode
OCI_ATTR_IS_INCOMPLETE_TYPE	不完全な型かどうか。	ub1
OCI_ATTR_IS_SYSTEM_TYPE	システム型かどうか。	ub1
OCI_ATTR_IS_PREDEFINED_TYPE	事前定義済みの型かどうか。	ub1
OCI_ATTR_IS_TRANSIENT_TYPE	一時的な型かどうか。	ub1
OCI_ATTR_IS_SYSTEM_GENERATED_TYPE	システム生成型かどうか。	ub1
OCI_ATTR_HAS_NESTED_TABLE	NESTED TABLE 属性を含む型であるかどうか。	ub1
OCI_ATTR_HAS_LOB	LOB 属性を含む型であるかどうか。	ub1
OCI_ATTR_HAS_FILE	FILE 属性を含む型であるかどうか。	ub1

表 6-7 型に所属する属性（続き）

属性	説明	属性データ型
OCI_ATTR_COLLECTION_ELEMENT	コレクション要素へのハンドル。6-13 ページの「 コレクション属性 」を参照。	dvoid *
OCI_ATTR_NUM_TYPE_ATTRS	型属性の数。	ub2
OCI_ATTR_LIST_TYPE_ATTRS	型属性のリスト。6-19 ページの「 リスト属性 」を参照。	dvoid *
OCI_ATTR_NUM_TYPE_METHODS	型メソッドの数。	ub2
OCI_ATTR_LIST_TYPE_METHODS	型メソッドのリスト。6-19 ページの「 リスト属性 」を参照。	dvoid *
OCI_ATTR_MAP_METHOD	型のマップ・メソッド。6-12 ページの「 型メソッド属性 」を参照。	dvoid *
OCI_ATTR_ORDER_METHOD	型のオーダー・メソッド。6-12 ページの「 型メソッド属性 」を参照。	dvoid *
OCI_ATTR_IS_INVOKER_RIGHTS	型が実行者権限型であるかどうか。	ub1
OCI_ATTR_NAME	型属性名である文字列へのポインタ。	text *
OCI_ATTR_SCHEMA_NAME	型作成時のスキーマ名が付いた文字列。	text *
OCI_ATTR_IS_FINAL_TYPE	最後の型かどうか。	ub1
OCI_ATTR_IS_INSTANTIABLE_TYPE	インスタンス化可能な型かどうか。	ub1
OCI_ATTR_IS_SUBTYPE	サブタイプかどうか。	ub1
OCI_ATTR_SUPERTYPE_SCHEMA_NAME	スーパータイプを含むスキーマの名前。	text *
OCI_ATTR_SUPERTYPE_NAME	スーパータイプの名前。	text *

型属性の属性

型の属性のパラメータの場合（OCI_PTYPE_TYPE_ATTR 型）、その属性は表 6-8 にリストされているとおりです。

表 6-8 型属性に所属する属性

属性	説明	属性データ型
OCI_ATTR_DATA_SIZE	型属性の最大サイズ。この長さは、文字列と行について、文字ではなくバイト単位で戻されます。NUMBER については 22 を戻します。	ub4
OCI_ATTR_TYPECODE	型コード。6-4 ページの「データ型コードの注意」を参照。	OCITypeCode
OCI_ATTR_DATA_TYPE	型属性のデータ型。6-4 ページの「データ型コードの注意」を参照。	ub2
OCI_ATTR_NAME	型属性名である文字列へのポインタ。	text *
OCI_ATTR_PRECISION	数値型属性の精度。精度が 0（ゼロ）以外でスケールが -127 の場合は FLOAT で、それ以外の場合は NUMBER（精度、スケール）です。精度が 0（ゼロ）の場合は、NUMBER（精度、スケール）を単に NUMBER と表すことができます。	明示的記述の場合は ub1 暗黙的記述の場合は sb2
OCI_ATTR_SCALE	数値型属性のスケール。精度が 0（ゼロ）以外でスケールが -127 の場合は FLOAT で、それ以外の場合は NUMBER（精度、スケール）です。精度が 0（ゼロ）の場合は、NUMBER（精度、スケール）を単に NUMBER と表すことができます。	sb1
OCI_ATTR_TYPE_NAME	型名である文字列。データ型が SQLT_NTY または SQLT_REF の場合は、戻される値には型名が含まれます。データ型が SQLT_NTY の場合は、名前付きデータ型の型名が戻されます。データ型が SQLT_REF の場合は、REF によって示されている名前付きデータ型の型名が戻されます。	text *
OCI_ATTR_SCHEMA_NAME	型作成時のスキーマ名が付いた文字列。	text *
OCI_ATTR_REF_TDO	列の型がオブジェクト型の場合は、その型の TDO のメモリー内 REF を戻します。OCIRef 用の領域が確保されていない場合は、領域がキャッシュ内に暗黙的に割り当てられます。次に、コール元は OCIObjectPin() を使用して TDO を確保できます。	OCIRef *
OCI_ATTR_CHARSET_ID	型属性が文字列 / キャラクタ・タイプの場合は、キャラクタ・セット ID。	ub2

表 6-8 型属性に所属する属性（続き）

属性	説明	属性データ型
OCI_ATTR_CHARSET_FORM	型属性が文字列 / キャラクタ・タイプの場合は、 キャラクタ・セット・フォーム。	ub1
OCI_ATTR_FSPRECISION	日時または時間隔の小数秒の精度。	ub1
OCI_ATTR_LFPRECISION	時間隔の先行フィールドの精度。	ub1

型メソッド属性

型のメソッドのパラメータの場合（OCI_PTYPE_TYPE_METHOD 型）、その属性は表 6-9 にリストされているとおりです。

表 6-9 型メソッドに所属する属性

属性	説明	属性データ型
OCI_ATTR_NAME	メソッド名（プロシージャまたはファンクション）。	text *
OCI_ATTR_ENCAPSULATION	メソッドのカプセル化レベル (OCI_TYPEENCAP_PRIVATE または OCI_TYPEENCAP_PUBLIC のいずれか)。	OCITYPEENCAP
OCI_ATTR_LIST_ARGUMENTS	引数リスト。6-5 ページの 「OCI_ATTR_LIST_ARGUMENTS についての 注意」または 6-19 ページの「リスト属性」を 参照。	dvoid *
OCI_ATTR_IS_CONSTRUCTOR	メソッドがコンストラクタかどうか。	ub1
OCI_ATTR_IS_DESTRUCTOR	メソッドがデストラクタかどうか。	ub1
OCI_ATTR_IS_OPERATOR	メソッドが演算子かどうか。	ub1
OCI_ATTR_IS_SELFISH	メソッドが自己参照的かどうか。	ub1
OCI_ATTR_IS_MAP	メソッドがマップ・メソッドかどうか。	ub1
OCI_ATTR_IS_ORDER	メソッドがオーダー・メソッドかどうか。	ub1
OCI_ATTR_IS_RNDS	「Read No Data State」がメソッドに設定され ているかどうか。	ub1
OCI_ATTR_IS_RNPS	「Read No Process State」がメソッドに設定され れているかどうか。	ub1
OCI_ATTR_IS_WNDS	「Write No Data State」がメソッドに設定され ているかどうか。	ub1

表 6-9 型メソッドに所属する属性（続き）

属性	説明	属性データ型
OCI_ATTR_IS_WNPS	「Write No Process State」がメソッドに設定されているかどうか。	ub1
OCI_ATTR_IS_FINAL_METHOD	最後のメソッドかどうか。	ub1
OCI_ATTR_IS_INSTANTIABLE_METHOD	インスタンス化可能なメソッドかどうか。	ub1
OCI_ATTR_IS_OVERRIDING_METHOD	オーバーライドするメソッドかどうか。	ub1

コレクション属性

コレクション型のパラメータの場合（OCI_PTYPE_COLL 型）、その属性は表 6-10 にリストされているとおりです。

表 6-10 コレクション型に所属する属性

属性	説明	属性データ型
OCI_ATTR_DATA_SIZE	型属性の最大サイズ。この長さは、文字列と行について、文字ではなくバイト単位で戻されます。NUMBER については 22 を戻します。	ub2
OCI_ATTR_TYPECODE	型コード。6-4 ページの「データ型コードの注意」を参照。	OCITypeCode
OCI_ATTR_DATA_TYPE	型属性のデータ型。6-4 ページの「データ型コードの注意」を参照。	ub2
OCI_ATTR_NUM_ELEMENTS	配列の要素の数。コレクションが配列である場合にのみ有効です。	ub4
OCI_ATTR_NAME	型属性名である文字列へのポインタ。	text *
OCI_ATTR_PRECISION	数値型属性の精度。精度が 0（ゼロ）以外でスケールが -127 の場合は FLOAT で、それ以外の場合は NUMBER（精度、スケール）です。精度が 0（ゼロ）の場合は、NUMBER（精度、スケール）を単に NUMBER と表すことができます。	明示的記述の場合は ub1 暗黙的記述の場合は sb2
OCI_ATTR_SCALE	数値型属性のスケール。精度が 0（ゼロ）以外でスケールが -127 の場合は FLOAT で、それ以外の場合は NUMBER（精度、スケール）です。精度が 0（ゼロ）の場合は、NUMBER（精度、スケール）を単に NUMBER と表すことができます。	sb1

表 6-10 コレクション型に所属する属性（続き）

属性	説明	属性データ型
OCI_ATTR_TYPE_NAME	型名である文字列。データ型が <code>SQLT_NTY</code> または <code>SQLT_REF</code> の場合は、戻される値には型名が含まれます。データ型が <code>SQLT_NTY</code> の場合は、名前付きデータ型の型名が戻されます。データ型が <code>SQLT_REF</code> の場合は、REF によって示されている名前付きデータ型の型名が戻されます。	text *
OCI_ATTR_SCHEMA_NAME	型作成時のスキーマ名が付いた文字列。	text *
OCI_ATTR_REF_TDO	列の型がオブジェクト型の場合は、その型の TDO のメモリー内 REF を戻します。OCISRef 用の領域が確保されていない場合は、領域がキャッシュ内に暗黙的に割り当てられます。次に、コール元は OCIOBJECTPIN() を使用して TDO を確保できます。	OCISRef *
OCI_ATTR_CHARSET_ID	型属性が文字列 / キャラクタ・タイプの場合は、キャラクタ・セット ID。	ub2
OCI_ATTR_CHARSET_FORM	型属性が文字列 / キャラクタ・タイプの場合は、キャラクタ・セット・フォーム。	ub1

シノニム属性

シノニムのパラメータの場合（型 `OCI_PTYPE_SYN`）、その属性は表 6-11 にリストされているとおりです。

表 6-11 シノニムに所属する属性

属性	説明	属性データ型
OCI_ATTR_OBJID	オブジェクト ID。	ub4
OCI_ATTR_SCHEMA_NAME	シノニム翻訳のスキーマ名を含む文字列。	text *
OCI_ATTR_NAME	シノニム翻訳のオブジェクト名を含む、ヌル文字で終了する文字列。	text *
OCI_ATTR_LINK	シノニム翻訳のデータベース・リンク名を含む、ヌル文字で終了する文字列。	text *

順序属性

順序のパラメータの場合（型 OCI_PTYPE_SEQ）、その属性は表 6-12 にリストされており
おりです。

表 6-12 順序に所属する属性

属性	説明	属性データ型
OCI_ATTR_OBJID	オブジェクト ID。	ub4
OCI_ATTR_MIN	最小値（Oracle 数値書式）。	ub1 *
OCI_ATTR_MAX	最大値（Oracle 数値書式）。	ub1 *
OCI_ATTR_INCR	増分（Oracle 数値書式）。	ub1 *
OCI_ATTR_CACHE	キャッシュされた順序番号の数。順序がキャッ シュされた順序でない場合は、0（ゼロ）です （Oracle 数値書式）。	ub1 *
OCI_ATTR_ORDER	順序が順序指定されているかどうか。	ub1
OCI_ATTR_HW_MARK	最高水位標（Oracle 数値書式）。	ub1 *

列属性

表またはビューの列のパラメータの場合（型 OCI_PTYPE_COL）、その属性は表 6-13 にリス
トされておりとおりです。

表 6-13 表またはビューの列に所属する属性

属性	説明	属性データ型
OCI_ATTR_CHAR_USED	列の長さセマンティクスの型を戻します。0（ゼ ロ）はバイト長セマンティクスを示し、1 は文字 長セマンティクスを示します。6-21 ページの「 記 述での文字長セマンティクスのサポート 」を参 照。	ub4
OCI_ATTR_CHAR_SIZE	列の文字長を戻します。この文字長は列で許容で きる文字数です。これに対して、バイト長を取得 するのは OCI_ATTR_DATA_SIZE です。6-21 ページの「 記述での文字長セマンティクスのサ ポート 」を参照。	ub2
OCI_ATTR_DATA_SIZE	列の最大サイズ。この長さは、文字列と行につい て、文字ではなくバイト単位で戻されます。 NUMBER については 22 を戻します。	ub2

表 6-13 表またはビューの列に所属する属性（続き）

属性	説明	属性データ型
OCI_ATTR_DATA_TYPE	列のデータ型。6-4 ページの「データ型コードの注意」を参照。	ub2
OCI_ATTR_NAME	列名である文字列へのポインタ。	text *
OCI_ATTR_PRECISION	数値列の精度。精度が 0（ゼロ）以外でスケールが -127 の場合は FLOAT で、それ以外の場合は NUMBER（精度、スケール）です。精度が 0（ゼロ）の場合は、NUMBER（精度、スケール）を単に NUMBER と表すことができます。	明示的記述の場合は ub1 暗黙的記述の場合は sb2
OCI_ATTR_SCALE	数値列のスケール。精度が 0（ゼロ）以外でスケールが -127 の場合は FLOAT で、それ以外の場合は NUMBER（精度、スケール）です。精度が 0（ゼロ）の場合は、NUMBER（精度、スケール）を単に NUMBER と表すことができます。	sb1
OCI_ATTR_IS_NULL	列に NULL 値が許可されていない場合に、0（ゼロ）を戻します。	ub1
OCI_ATTR_TYPE_NAME	型名である文字列を戻します。データ型が SQLT_NTY または SQLT_REF の場合は、戻される値には型名が含まれます。データ型が SQLT_NTY の場合は、名前付きデータ型の型名が戻されます。データ型が SQLT_REF の場合は、REF によって示されている名前付きデータ型の型名が戻されます。	text *
OCI_ATTR_SCHEMA_NAME	型の作成時に使用したスキーマ名を持つ文字列を戻します。	text *
OCI_ATTR_REF_TDO	列の型がオブジェクト型の場合は、その型の TDO の REF。	OCISRef *
OCI_ATTR_CHARSET_ID	列が文字列 / キャラクタ・タイプの場合は、キャラクタ・セット ID。	ub2
OCI_ATTR_CHARSET_FORM	列が文字列 / キャラクタ・タイプの場合は、キャラクタ・セット・フォーム。	ub1

引数 / 結果属性

プロシージャ / ファンクションの引数のパラメータ (OCI_PTYPE_ARG 型)、型メソッドの引数のパラメータ (OCI_PTYPE_TYPE_ARG 型) またはメソッド結果のパラメータ (OCI_PTYPE_TYPE_RESULT 型) の場合、属性は表 6-14 にリストされているとおりです。

表 6-14 引数 / 結果に所属する属性

属性	説明	属性データ型
OCI_ATTR_NAME	引数名である文字列へのポインタを戻します。	text *
OCI_ATTR_POSITION	引数リストの引数の位置。常に 0 (ゼロ) を戻します。	ub2
OCI_ATTR_TYPECODE	型コード。6-4 ページの「データ型コードの注意」を参照。	OCITypeCode
OCI_ATTR_DATA_TYPE	引数のデータ型。6-4 ページの「データ型コードの注意」を参照。	ub2
OCI_ATTR_DATA_SIZE	引数のデータ型のサイズ。この長さは、文字列と行について、文字ではなくバイト単位で戻されます。NUMBER については 22 を戻します。	ub2
OCI_ATTR_PRECISION	数値引数の精度。精度が 0 (ゼロ) 以外でスケールが -127 の場合は FLOAT で、それ以外の場合は NUMBER (精度、スケール) です。精度が 0 (ゼロ) の場合は、NUMBER (精度、スケール) を単に NUMBER と表すことができます。	明示的記述の場合は ub1 暗黙的記述の場合は sb2
OCI_ATTR_SCALE	数値引数のスケール。精度が 0 (ゼロ) 以外でスケールが -127 の場合は FLOAT で、それ以外の場合は NUMBER (精度、スケール) です。精度が 0 (ゼロ) の場合は、NUMBER (精度、スケール) を単に NUMBER と表すことができます。	sb1
OCI_ATTR_LEVEL	データ型のレベル。この属性は常に 0 (ゼロ) を戻します。	ub2
OCI_ATTR_HAS_DEFAULT	引数にデフォルト値があるかどうかを示します。	ub1
OCI_ATTR_LIST_ARGUMENTS	次のレベルの引数のリスト (引数がレコード型または表型のとき)。	dvoid *

表 6-14 引数 / 結果に所属する属性（続き）

属性	説明	属性データ型
OCI_ATTR_IOMODE	次の引数モードを示します。 0 は IN (OCI_TYPEPARAM_IN) 1 は OUT (OCI_TYPEPARAM_OUT) 2 は IN/OUT (OCI_TYPEPARAM_INOUT)	OCITypeParamMode
OCI_ATTR_RADIX	基数を戻します（数値型の場合）。	ub1
OCI_ATTR_IS_NULL	列に NULL 値が許可されていない場合に、0（ゼロ）を戻します。	ub1
OCI_ATTR_TYPE_NAME	型名である文字列、またはパッケージのローカル型の場合は、パッケージ名の文字列を戻します。データ型が <code>SQLT_NTY</code> または <code>SQLT_REF</code> の場合は、戻される値には型名が含まれます。データ型が <code>SQLT_NTY</code> の場合は、名前付きデータ型の型名が戻されます。データ型が <code>SQLT_REF</code> の場合は、REF によって示されている名前付きデータ型の型名が戻されます。	text *
OCI_ATTR_SCHEMA_NAME	<code>SQLT_NTY</code> または <code>SQLT_REF</code> については、型の作成時に使用したスキーマ名を持つ文字列を戻すか、パッケージのローカル型の場合には、そのパッケージの作成時に使用したスキーマ名を持つ文字列を戻します。	text *
OCI_ATTR_SUB_NAME	<code>SQLT_NTY</code> または <code>SQLT_REF</code> については、パッケージのローカル型の場合に、型名を持つ文字列を戻します。	text *
OCI_ATTR_LINK	<code>SQLT_NTY</code> または <code>SQLT_REF</code> については、型が存在しているデータベースのデータベース・リンク名を持つ文字列を戻します。この動作は、パッケージがリモートで、そのパッケージがローカル型の場合にのみ発生します。	text *
OCI_ATTR_REF_TDO	引数の型がオブジェクトの場合に、その型の TDO の REF を戻します。	OCIRef *
OCI_ATTR_CHARSET_ID	引数が文字列 / キャラクタ・タイプの場合に、キャラクタ・セット ID を戻します。	ub2
OCI_ATTR_CHARSET_FORM	引数が文字列 / キャラクタ・タイプの場合に、キャラクタ・セット・フォームを戻します。	ub1

リスト属性

列または引数、サブプログラムのリストのパラメータ（型 OCI_PTYPE_LIST）の場合、型特有の属性およびハンドル（パラメータ）は次のとおりです。

リストには、リスト型を指定する OCI_ATTR_LIST_TYPE 属性があります。リストの横断時に可能な値とその下限は次のとおりです。

表 6-15 リスト属性

リスト属性	説明	下限
OCI_LTYPE_COLUMN	列リスト。	1
OCI_LTYPE_ARG_PROC	プロシージャ引数リスト。	1
OCI_LTYPE_ARG_FUNC	ファンクション引数リスト。	0
OCI_LTYPE_SUBPRG	サブプログラム・リスト。	0
OCI_LTYPE_TYPE_ATTR	型属性リスト。	1
OCI_LTYPE_TYPE_METHOD	型メソッド・リスト。	1
OCI_LTYPE_TYPE_ARG_PROC	結果引数なしの型メソッド・リスト。	0
OCI_LTYPE_TYPE_ARG_FUNC	結果引数なしの型メソッド・リスト。	1
OCI_LTYPE_SCH_OBJ	スキーマ内のオブジェクト・リスト。	0
OCI_LTYPE_DB_SCH	データベース内のスキーマ・リスト。	0

- リストには、リスト内の要素数を調べる OCI_ATTR_NUM_PARAMS 属性があります。
- 各リストには、LowerBound .. OCI_ATTR_NUM_PARAMS パラメータがあります。LowerBound は、表 6-15「リスト属性」の「下限」列の値です。ファンクション引数リストの場合、位置 0 には、戻り値（型 OCI_PTYPE_ARG）のパラメータがあります。

スキーマ属性

スキーマ型のパラメータの場合（型 OCI_PTYPE_SCHEMA）、その属性は表 6-16 にリストされているとおりです。

表 6-16 スキーマ固有の属性

属性	説明	属性データ型
OCI_ATTR_LIST_OBJECTS	スキーマ内のオブジェクトのリスト。	text*

データベース属性

データベース型のパラメータの場合（型 OCI_PTYPE_DATABASE）、その属性は表 6-17 にリストされているとおりです。

表 6-17 データベース固有の属性

属性	説明	属性データ型
OCI_ATTR_VERSION	データベースのバージョン。	text*
OCI_ATTR_CHARSET_ID	サーバー・ハンドルからのデータベース・キャラクタ・セット ID。	ub2
OCI_ATTR_NCHARSET_ID	サーバー・ハンドルからのデータベース・キャラクタ・セット ID。	ub2
OCI_ATTR_LIST_SCHEMAS	データベース内のスキーマのリスト (OCI_PTYPE_SCHEMA 型)。	OCI_PTYPE_LIST
OCI_ATTR_MAX_PROC_LEN	プロシージャ名の最大長。	ub4
OCI_ATTR_MAX_COLUMN_LEN	列名の最大長。	ub4
OCI_ATTR_CURSOR_COMMIT_BEHAVIOR	カーソルおよびデータベース内のプリコンパイルされた SQL 文に、コミット操作を反映させる方法。可能な値は次のとおりです。 OCI_CURSOR_OPEN – カーソルの状態を COMMIT 操作の前の状態に保持します。 OCI_CURSOR_CLOSED – カーソルは、COMMIT 操作時にクローズします。ただし、アプリケーションでは、文を再度プリコンパイルせずに再実行できます。	ub1
OCI_ATTR_MAX_CATALOG_NAMELEN	カタログ（データベース）名の最大長。	ub1
OCI_ATTR_CATALOG_LOCATION	修飾表内のカタログの位置。値は OCI_CL_START および OCI_CL_END です。	ub1
OCI_ATTR_SAVEPOINT_SUPPORT	データベースがセーブポイントをサポートするかどうか。値は OCI_SP_SUPPORTED および OCI_SP_UNSUPPORTED です。	ub1

表 6-17 データベース固有の属性 (続き)

属性	説明	属性データ型
OCI_ATTR_NOWAIT_SUPPORT	データベースが NOWAIT 句をサポートするかどうか。値は OCI_NW_SUPPORTED および OCI_NW_UNSUPPORTED です。	ub1
OCI_ATTR_AUTOCOMMIT_DDL	DDL 文で自動コミット・モードが必要かどうか。値は OCI_AC_DDL および OCI_NO_AC_DDL です。	ub1
OCI_ATTR_LOCKING_MODE	データベースのロック・モード。値は OCI_LOCK_IMMEDIATE および OCI_LOCK_DELAYED です。	ub1

記述での文字長セマンティクスのサポート

Oracle9i 以上での問合せ情報と列情報は文字長セマンティクスでサポートされます。

記述ハンドルの次の属性は、文字長セマンティクスをサポートします。

OCI_ATTR_CHAR_SIZE。列の文字長を取得します。この文字長は列で許容できる文字数です。これに対して、バイト長を取得するのは **OCI_ATTR_DATA_SIZE** です。

OCI_ATTR_CHAR_USED。列の長さセマンティクスの型を取得します。0 (ゼロ) はバイト長セマンティクスを示し、1 は文字長セマンティクスを示します。

アプリケーションでは、**OCIStmtExecute()** を使用して選択リスト (問合せ) を暗黙的にも明示的にも記述できます。その他のスキーマ要素は、**OCIDescribeAny()** を使用して明示的に記述する必要があります。

暗黙的記述

データベース列が文字長セマンティクスを使用して作成されている場合、暗黙的な記述情報には、文字長とバイト長、およびデータベース列の作成方法を示すフラグが含まれます。つまり、**OCI_ATTR_CHAR_SIZE** は、列または式の文字長です。この場合、**OCI_ATTR_CHAR_USED** フラグは、列または式が文字長セマンティクスを使用して作成されたことを示す 1 になります。

OCI_ATTR_DATA_SIZE は、すべてのデータ (**OCI_ATTR_CHAR_SIZE** の文字数と同じ) を保持するのに十分な大きさの値になります。**OCI_ATTR_DATA_SIZE** は通常、**OCI_ATTR_CHAR_SIZE** にクライアントの各文字の最大バイトを乗算した値に設定されます。

データベース列がバイト長セマンティクスを使用して作成されている場合、暗黙的な記述による動作は、リリース 1 (9.0.1) より前と同じになります。戻される

OCI_ATTR_DATA_SIZE は、列のバイト長に、クライアントとサーバーのキャラクタ・セット間での最大変換率を乗算した値になります。つまり、列のバイト長 ÷ サーバーの各文字の

最大バイト数×クライアントの各文字の最大バイト数になります。

OCI_ATTR_CHAR_USED は 0 で、OCI_ATTR_CHAR_SIZE は OCI_ATTR_DATA_SIZE と同じ値に設定されます。

明示的記述

表の明示的記述には、バイト数で表された列のサイズ（サーバーで表示されるサイズと同じ）を取得する OCI_ATTR_DATA_SIZE 属性、OCI_ATTR_CHAR_SIZE の長さ（文字数）、および列の作成方法を示すフラグの OCI_ATTR_CHAR_USED が含まれます。

OCI_ATTR_CHAR_USED フラグが設定されている場合は、挿入時に、バインド・ハンドルの OCI_ATTR_MAXCHAR_SIZE（バインドの説明を参照）を、明示的な記述からパラメータ・ハンドルの OCI_ATTR_CHAR_SIZE によって戻される値に設定できます。これにより、列のサイズ制限に違反することはありません。

関連項目： 5-38 ページ「[バインド](#)」

明示的記述を行う別の方法は OCIDescribeAny() 関数を使用する方法です。この関数は、表、ビュー、シノニム、プロシージャ、ファンクション、パッケージ、順序、型、スキーマ、データベースなどの既存のスキーマ・オブジェクトに対する汎用的な記述コールです。このコールでは、表内の列などのサブスキーマ・オブジェクトも記述できます。このコールは、OCIAttrGet() コールを使用して取得できるオブジェクト固有の属性を記述ハンドルに移入します。

パラメータ記述子に対して OCIAttrGet() をコールすると、ストアド・プロシージャやストアド・ファンクションのパラメータの特定の属性、あるいは表の列記述子が戻されます。OCIDescribeAny() によってスキーマ・オブジェクト記述全体がクライアント側にキャッシュされているため、これらの後続コールは、サーバーへのラウンドトリップを別に行う必要がありません。

属性 OCI_ATTR_CHAR_SIZE で OCIAttrGet() をコールした場合、ストアド・プロシージャ・パラメータはバインドされないため、データは戻されません。

アプリケーションで表を記述する場合は、その表の列の情報も取り出すことができます。さらに、OCIDescribeAny() では、表の列、関数のパッケージ、型のフィールドなどのサブスキーマ・オブジェクトの名前がわかっている場合、それらのサブスキーマ・オブジェクトを直接記述できます。すべての場合において、列およびデータ型に関する特定の情報は、ハンドル属性を読み込むことによって検索されます。

SQL 文を実行した後、文ハンドルの属性として選択リストに関する情報を取得できます。明示的な記述コールは必要ありません。アプリケーションで文ハンドルから選択リストに関する情報を取り出すには、選択リストの各位置に対して OCIParamGet() を 1 回ずつコールし、その位置のパラメータ記述子を割り当てる必要があります。

アプリケーションでは、選択リストの位置のパラメータ記述子を割り当てた後、パラメータ記述子について OCIAttrGet() をコールすることによって、特定の情報を取り出せます。パラメータ記述子から取得できる情報として、データ型とパラメータの最大サイズがあります。

記述に関するクライアントとサーバー間の互換性の問題

Oracle9i 以上のクライアントが Oracle8i 以下のサーバーと通信を行うと、そのクライアントは、データベースがバイト長セマンティクスのみを使用している場合と同じように動作します。

Oracle8i 以下のクライアントが Oracle9i 以上のサーバーと通信を行うと、クライアント側で OCI_ATTR_CHAR_SIZE 属性と OCI_ATTR_CHAR_USED 属性を使用することはできません。

いずれの場合も、サーバーまたはクライアントが Oracle8i 以下のソフトウェア・リリースである場合、文字長セマンティクスは記述できません。

OCIDecribeAny() の使用例

次の例では、異なる種類のスキーマ・オブジェクトを記述するための OCIDecribeAny() の使用方法を示します。より詳細なコード例は、Oracle のインストールに含まれているデモ・プログラム cdemodsa.c を参照してください。

関連項目： デモ・プログラムの詳細は、[付録 B「OCI デモ・プログラム」](#) を参照してください。

表用の列データ型の抽出し

この例では、明示的な記述の使用法を示しています。表の列データ型を取り出すアプリケーションを例として考えます。次のコード・フラグメントは、アプリケーションで記述インタフェースを使用する例です。

```
...
text objptr[] = "tablename";
OCIParam      *mypard;
ub4           counter;
ub2           dtype;
text          *col_name;
ub4           counter, col_name_len, char_semantics, col_width;
ub4 objp_len = strlen("tablename");
OCIParam *parmh;          /* parameter handle */
OCIParam *collsthd;       /* handle to list of columns */
OCIParam *colhd;          /* column handle */
OCIDecribe *dschp;        /* describe handle */

...
/* get the describe handle for the table */
if (OCIDecribeAny(svch, errh, objptr, objp_len, OCI_OTYPE_NAME, 0,
    OCI_PTYPE_TABLE, dschp))
    return error;
/* get the parameter handle */
```

```
if (OCIAttrGet(dschp, OCI_HTYPE_DESCRIBE, &parmh, 0, OCI_ATTR_PARAM,
    errh))
    return error;
/* The type information of the object, in this case, OCI_PTYPE_TABLE,
is obtained from the parameter descriptor returned by the OCIAttrGet(). */
/* get the number of columns in the table */
if (OCIAttrGet(parmh, OCI_DTYPE_PARAM, &numcols, 0,
    OCI_ATTR_NUM_COLS, errh))
    return error;
/* get the handle to the column list of the table */
if (OCIAttrGet(parmh, OCI_DTYPE_PARAM, &collsthd, 0,
    OCI_ATTR_LIST_COLUMNS, errh) == OCI_NO_DATA)
    return error;
/* go through the column list and retrieve the data-type of each column,
and then recursively describe column types. */

for (i = 1; i <= numcols; i++)
{
    /* get parameter for column i */
    if (OCIParamGet(collsthd, OCI_DTYPE_PARAM, errh, &colhd, i))
        return error;
    /* for example, get data type for ith column */
    if (OCIAttrGet(colhd, OCI_DTYPE_PARAM, &datatype[i-1], 0,
        OCI_ATTR_DATA_TYPE, errh))
        return error;
    /* Retrieve the length semantics for the column */
    OCIAttrGet((dvoid*) mypard, (ub4) OCI_DTYPE_PARAM,
        (dvoid*) &char_semantics, (ub4 *) 0, (ub4) OCI_ATTR_CHAR_USED,
        (OCIError *) errhp );
    if (char_semantics)
        /* Retrieve the column width in characters */
        OCIAttrGet((dvoid*) mypard, (ub4) OCI_DTYPE_PARAM,
            (dvoid*) &col_width, (ub4 *) 0, (ub4) OCI_ATTR_CHAR_SIZE,
            (OCIError *) errhp );
    else
        /* Retrieve the column width in bytes */
        OCIAttrGet((dvoid*) mypard, (ub4) OCI_DTYPE_PARAM,
            (dvoid*) &col_width, (ub4 *) 0, (ub4) OCI_ATTR_DATA_SIZE,
            (OCIError *) errhp );
}
...
```

ストアド・プロシージャの記述

ストアド・プロシージャとファンクションについて考えてみます。プロシージャとファンクションの違いは、ファンクションには引数リストに位置 0（ゼロ）の戻り型がありますが、プロシージャには、引数リストの位置 0（ゼロ）に対応する引数はありません。型メソッドの記述に必要なステップ（ファンクションとプロシージャへの分割も行います）は、通常の PL/SQL ファンクションおよびプロシージャのステップとまったく同じです。プロシージャおよびファンクションは、オブジェクトのデフォルト型を引数として使用できることに注意してください。次のプロシージャがあるとします。

```
P1 (arg1 emp.sal%type, arg2 emp%rowtype)
```

さらに、`emp` 表の各行には、`name (VARCHAR2(20))` および `sal (NUMBER)` の 2 列があるとします。つまり、`P1` の引数リストには、`arg1` と `arg2` の 2 つの引数がそれぞれレベル 0 の位置 1 と位置 2 にあり、引数の `name` と `sal` がそれぞれレベル 1 の位置 1 と位置 2 にあります。`P1` の記述では、引数の数として 2 を戻し、レベル 0 を超える (> 0) 引数を、レベル 0（ゼロ）の引数の属性として戻します。

次のコード・フラグメントは、`P1` の記述を説明しています。

```
...
text objptr[] = "P1";          /* procedure name */
ub4 objp_len = strlen("P1");
OCIParam *parmh;               /* parameter handle */
OCIParam *arglst;              /* list of args */
OCIParam *arg;                 /* argument handle */
OCIDescribe *dschp;            /* describe handle */

ub2 numargs, pos, level;
text *name;
ub4 namelen;

...
/* get the describe handle for the table */
if (OCIDescribeAny(svch, errh, objptr, objp_len, OCI_OTYPE_NAME, 0,
    OCI_PTYPE_PROC, dschp))
    return error;

/* get the parameter handle */
if (OCIAttrGet(dschp, OCI_HTYPE_DESCRIBE, &parmh, 0, OCI_ATTR_PARAM,
    errh))
    return error;
```

```

/* Get the number of arguments and the arg list */
if (OCIAttrGet (parmh, OCI_DTYPE_PARAM, &arglst,
0, OCI_ATTR_LIST_ARGUMENTS, errh))
    return error;
if (OCIAttrGet (parmh, OCI_DTYPE_PARAM, &numargs, 0,
OCI_ATTR_NUM_PARAMS, errh))
    return error;

/* For a procedure, we begin with i = 1; for a
function, we begin with i = 0. */

for (i = 1; i < numargs; i++) {
OCIParamGet (arglst, OCI_DTYPE_PARAM, errh, &arg, i);
OCIAttrGet (arg, OCI_DTYPE_PARAM, &name, &namelen, OCI_ATTR_NAME,
errh);
...
/* to print the attributes of the argument of type record
(arguments at the next level), traverse the argument list */

OCIAttrGet (arg, OCI_DTYPE_PARAM, &arglst1, 0,
OCI_ATTR_LIST_ARGUMENTS, errh);

/* check if the current argument is a record. For arg1 in P1
arglst1 is NULL. */

if (arglst1) {
OCIAttrGet (arg, OCI_DTYPE_PARAM, &numargs1, 0, OCI_ATTR_NUM_PARAMS,
errh);

/* Note that for both functions and procedures, the next higher level
arguments start from index 1. For arg2 in P1, the number of arguments at
the level 1 would be 2 */

for (i = 1; i < numargs1, i++) {
OCIParamGet (arglst1, OCI_DTYPE_PARAM, errh, &arg1, i);
OCIAttrGet (arg1, OCI_DTYPE_PARAM, &name1, &namelen1,
OCI_ATTR_NAME, errh);
...
}
}
}
...

```


オブジェクト型の属性の取出し

この例では、名前付きオブジェクト型についての明示的な記述の使用法を示します。オブジェクトを名前またはオブジェクト参照 (**OCIRef**) によって記述する方法を説明します。次のコード・フラグメントによって、オブジェクト型の属性について各データ型値の取出しが試みられます。

関連項目： 次の例は 6-23 ページの「[表用の列データ型の取出し](#)」の最初の例に類似しています。

```
...
text type_name[] = "typename";
ub4 type_name_len = strlen("typename");
OCIRef *type_ref = ...;
un4 numattrs;
OCIDescribe *dschp;      /* describe handle */
OCIParam *parmh;         /* parameter handle */
OCIParam *attrlsthd;     /* handle to list of attrs */
OCIParam *attrrh;        /* attribute handle */
...
/* allocate describe handle */
if (OCIHandleAlloc((dvoid *)envh, (dvoid **)&dschp,
                  (ub4)OCI_HTYPE_DESCRIBE, (size_t)0, (dvoid **)0))
    return error;

/* get the describe handle for the type */
if (describe_by_name)
    if (OCIDescribeAny(svch, errh, (dvoid*)type_name, type_name_len,
                      OCI_OTYPE_NAME, 0, OCI_PTYPE_TYPE, dschp))
        return error;
else
    if (OCIDescribeAny(svch, errh, (dvoid*)type_ref, 0, OCI_OTYPE_REF,
                      0, OCI_PTYPE_TYPE, dschp))
        return error;

/* get the parameter handle */
if (OCIAttrGet(dschp, OCI_HTYPE_DESCRIBE, &parmh, 0, OCI_ATTR_PARAM,
              errh))
    return error;

/* The type information of the object, in this case, OCI_PTYPE_TYPE, is
obtained from the parameter descriptor returned by the OCIAttrGet */

/* get the number of attributes in the type */
if (OCIAttrGet(parmh, OCI_DTYPE_PARAM, &numattrs, 0,
              OCI_ATTR_NUM_TYPE_ATTRS, errh))
    return error;
```

```

/* get the handle to the attribute list of the type */
if (OCIAttrGet(parmh, OCI_DTYPE_PARAM, (dvoid *)&attrlsthd, 0,
    OCI_ATTR_LIST_TYPE_ATTRS, errh)==OCI_NO_DATA)
    return error;

/* go through the attribute list and retrieve the data-type of each attribute, and
then recursively describe attribute types. */

for (i = 1; i <= numattrs; i++)
{
/* get parameter for attribute i */
if (OCIParamGet(attrlsthd, OCI_DTYPE_PARAM, errh, &attrhd, i))
    return error;

/* for example, get data type and typecode for attribute; note that
OCI_ATTR_DATA_TYPE returns the SQLT code, while OCI_ATTR_TYPECODE returns the Oracle
Type System typecode. */
if (OCIAttrGet(attrhd, OCI_DTYPE_PARAM,&datatype[i-1], 0,
    OCI_ATTR_DATA_TYPE,errh))
    return error;
/* for example, get data type for attribute*/
if (OCIAttrGet(attrhd, OCI_DTYPE_PARAM,&typecode[i-1], 0,
    OCI_ATTR_TYPECODE, errh))
    return error;

/* if attribute is an object type, recursively describe it */
if (typecode[i-1] == OCI_TYPECODE_OBJECT)
{
OCIRef *attr_type_ref;
OCIDescribe *nested_dschnp;

/* allocate describe handle */
if (OCIHandleAlloc((dvoid *)envh,(dvoid**)&dschnp,
    (ub4)OCI_HTYPE_DESCRIBE,(size_t)0, (dvoid **)0))
    return error;

if (OCIAttrGet(attrhd, OCI_DTYPE_PARAM,
    &attr_type_ref, 0, OCI_ATTR_REF_TDO,errh))
    return error;
OCIDescribeAny(svch, errh,(dvoid*)attr_type_ref, 0,
    OCI_OTYPE_REF, 0, OCI_PTYPE_TYPE, nested_dschnp);
/* go on describing the type... */
}
}

```

名前付きコレクション型のコレクション要素のデータ型の取出し

この例では、名前付きコレクション型についての明示的な記述の使用方を示しています。オブジェクトを名前またはオブジェクト参照（**OCIRef**）によって記述する方法を説明します。次のコード・フラグメントによって、オブジェクト型の属性について各データ型値の取出しが試みられます。

関連項目： 次の例は 6-23 ページの「[表用の列データ型の取出し](#)」の最初の例に類似しています。

```
...
text type_name[] = "typename";
ub4 type_name_len = strlen("typename");
OCIRef *type_ref = ...;
un4 numattrs;
OCIDescribe *dschp;      /* describe handle */
OCIParam *parmh;         /* parameter handle */
OCIParam *attrlsthd;     /* handle to list of attrs */
OCIParam *attrrh;        /* attribute handle */
...
/* allocate describe handle */
if (OCIHandleAlloc((dvoid *)envh, (dvoid **)&dschp,
                  (ub4)OCI_HTYPE_DESCRIBE, (size_t)0, (dvoid **)0))
    return error;

/* get the describe handle for the type */
if (describe_by_name)
    if (OCIDescribeAny(svch, errh, (dvoid*)type_name, type_name_len,
                      OCI_OTYPE_NAME, 0, OCI_PTYPE_TYPE, dschp))
        return error;
else
    if (OCIDescribeAny(svch, errh, (dvoid*)type_ref, 0, OCI_OTYPE_REF, 0,
                      OCI_PTYPE_TYPE, dschp))
        return error;

/* get the parameter handle */
if (OCIAttrGet(dschp, OCI_HTYPE_DESCRIBE, &parmh, 0, OCI_ATTR_PARAM,
              errh))
    return error;

/* get the Oracle Type System type code of the type to determine that this is a
collection type */
if (OCIAttrGet(attrrh, OCI_DTYPE_PARAM, &typecode, 0, OCI_ATTR_TYPECODE,
              errh))
    return error;
```

```

/* if typecode is OCI_TYPECODE_NAMEDCOLLECTION,
   proceed to describe collection element */
if (typecode == OCI_TYPECODE_NAMEDCOLLECTION)
{
    /* get the collection's type: ie, OCI_TYPECODE_VARRAY or OCI_TYPECODE_TABLE */

    if (OCIAttrGet(parmh, OCI_DTYPE_PARAM, (dvoid *)&collection_typecode, 0,
        OCI_ATTR_COLLECTION_TYPECODE, errh))
        return error;

    /* get the collection element; you MUST use this to further retrieve information
       about the collection's element */
    if (OCIAttrGet(parmh, OCI_DTYPE_PARAM, &collection_element_parmh, 0,
        OCI_ATTR_COLLECTION_ELEMENT, errh))
        return error;

    /* get the number of elements if collection is a VARRAY; not valid for nested tables
       */
    if (collection_typecode == OCI_TYPECODE_VARRAY)
        if (OCIAttrGet(collection_element_parmh, OCI_DTYPE_PARAM,
            (dvoid *)&num_elements, 0, OCI_ATTR_NUM_ELEMENTS, errh))
            return error;

    /* now use the collection_element parameter handle to retrieve information about the
       collection element */
    if (OCIAttrGet(collection_element_parmh, OCI_DTYPE_PARAM,
        (dvoid *)&element_typecode, 0, OCI_ATTR_TYPECODE, errh))
        return error;

    /* do the same to describe additional collection element information; this is very
       similar to describing type attributes */
    ...
}

```

文字長セマンティクスを使用した記述

次のコード例では、問合せ実行に続く問合せに対応した列名やデータ型を取り出すループを示しています。この問合せは、事前に OCISetStmtPrepare() のコールによって文ハンドルと関連付けられています。

```
...
OCIParam      *mypard;
ub4           counter;
ub2           dtype;
text          *col_name;
ub4           counter, col_name_len, char_semantics, col_width;
sb4           parm_status;
...
/* Request a parameter descriptor for position 1 in the select-list */
counter = 1;
parm_status = OCIParamGet(stmthp, OCI_HTYPE_STMT, errhp, &mypard,
                          (ub4) counter);
/* Loop only if a descriptor was successfully retrieved for
   current position, starting at 1 */
while (parm_status == OCI_SUCCESS) {
    /* Retrieve the data type attribute */
    checkerr(errhp, OCIAttrGet((dvoid*) mypard, (ub4) OCI_DTYPE_PARAM,
                              (dvoid*) &dtype, (ub4 *) 0, (ub4) OCI_ATTR_DATA_TYPE,
                              (OCIError *) errhp ));
    /* Retrieve the column name attribute */
    checkerr(errhp, OCIAttrGet((dvoid*) mypard, (ub4) OCI_DTYPE_PARAM,
                              (dvoid**) &col_name, (ub4 *) &col_char_len, (ub4) OCI_ATTR_NAME,
                              (OCIError *) errhp ));
    /* Retrieve the length semantics for the column */
    checkerr(errhp, OCIAttrGet((dvoid*) mypard, (ub4) OCI_DTYPE_PARAM,
                              (dvoid*) &char_semantics, (ub4 *) 0, (ub4) OCI_ATTR_CHAR_USED,
                              (OCIError *) errhp ));
    if (char_semantics)
        /* Retrieve the column width in characters */
        checkerr(errhp, OCIAttrGet((dvoid*) mypard, (ub4) OCI_DTYPE_PARAM,
                                    (dvoid*) &col_width, (ub4 *) 0, (ub4) OCI_ATTR_CHAR_SIZE,
                                    (OCIError *) errhp ));
    else
        /* Retrieve the column width in bytes */
        checkerr(errhp, OCIAttrGet((dvoid*) mypard, (ub4) OCI_DTYPE_PARAM,
                                    (dvoid*) &col_width, (ub4 *) 0, (ub4) OCI_ATTR_DATA_SIZE,
                                    (OCIError *) errhp ));
    printf("column=%s datatype=%d, char semantics=%d, width=%d\n\n", col_name,
           dtype, char_semantics, col_width);
    fflush(stdout);
    /* increment counter and get next descriptor, if there is one */
    counter++;
}
```

```
        parm_status = OCIParmGet(stmthp, OCI_HTYPE_STMT, errhp, &mypard,  
                                (ub4) counter);  
    }  
    ...
```

LOB と FILE の操作

この章は、次の項目で構成されています。

- [LOB での OCI 関数の使用](#)
- [内部 LOB の作成と変更](#)
- [内部 LOB の作成と変更](#)
- [表内の FILE とオペレーティング・システム・ファイルの関連付け](#)
- [オブジェクトの LOB 属性](#)
- [LOB の配列インタフェース](#)
- [LOB および FILE の関数](#)
- [LOB 読み込みおよび書き込みコールバック](#)
- [一時 LOB のサポート](#)

LOB での OCI 関数の使用

Oracle OCI には、データベース内のラージ・オブジェクト（Large Object: LOB）で操作を実行するための一連の関数があります。内部 LOB（BLOB、CLOB、NCLOB）はデータベースの表領域に格納されており、領域の最適化とアクセスの効率化を実現します。これらの LOB は、データベース・サーバーのトランザクションを完全にサポートしています。外部 LOB（FILE）とは、データベース表領域の外のサーバーのオペレーティング・システム・ファイルに格納された大型のデータ・オブジェクトのことです。

また、OCI は一時 LOB もサポートしています。一時 LOB は、LOB データ上で操作するローカル変数のように使用することができます。

LOB および FILE の最大長は、4GB です。FILE 機能は読取り専用です。Oracle では、現在、バイナリ・ファイル（BFILE）のみをサポートしています。

関連項目：

- LOB 操作の使用方法を示したコード例については、Oracle のインストールに付属のデモ・プログラムを参照してください。追加情報については、[付録 B「OCI デモ・プログラム」](#)を参照してください。
- LOB と併用するための `dbms_lob` パッケージの詳細は、『Oracle9i PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』を参照してください。
- LOB および使用可能な LOB インタフェースの一般情報については、『Oracle9i アプリケーション開発者ガイド - ラージ・オブジェクト』を参照してください。
- 一時 LOB の詳細は、7-17 ページの「[一時 LOB のサポート](#)」を参照してください。

内部 LOB の作成と変更

内部 LOB を新規作成するには、`OCIDescriptorAlloc()` を使用して新規の LOB ロケータを初期化してから、`OCIAttrSet()` をコールして（`OCI_ATTR_LOBEMPTY` 属性を使用して）空に設定し、次に、そのロケータを `INSERT` 文でプレースホルダにバインドします。この処理により、LOB 列または属性がある表に空のロケータを挿入します。次に、`SELECT...FOR UPDATE` でこの行を選択し、ロケータを取得します。その後、OCI LOB 関数の 1 つを使用してロケータに書き込みます。

注意： LOB 列または属性を変更（書込み、コピー、切捨てなど）する場合は、必ずその LOB が含まれている行をロックする必要があります。そのためには、操作を実行する前に `SELECT...FOR UPDATE` 文を使用してロケータを選択します。

LOB 書込みコマンドを正常終了するには、トランザクションをオープンしておく必要があります。このため、データの書込み前にトランザクションをコミットする場合は、(SELECT...FOR UPDATE 文の再発行などにより) 行を再ロックする必要があります。これは、コミットによってトランザクションがクローズされるためです。

OCIAttrSet() ではなく、EMPTY_BLOB() および EMPTY_CLOB() を使用した内部 LOB 作成の詳細は、次の関連項目の 2 番目のマニュアルを参照してください。

関連項目：

- LOB ロケータのプレースホルダへのバインドおよびそれらを INSERT 文で使用する場合は、5-10 ページの「[LOB のバインド](#)」を参照してください。
- トリガー内からの LOB の読み込みと書き込みの詳細は、『Oracle9i アプリケーション開発者ガイド - ラージ・オブジェクト』を参照してください。

表内の FILE とオペレーティング・システム・ファイルの関連付け

INSERT 文で BFILENAME() 関数を使用すると、外部のサーバー側 (OS) ファイルと表内の BFILE 列 / 属性を関連付けることができます。UPDATE 文で BFILENAME() を使用して、BFILE の列または属性を別の OS ファイルに関連付けます。OCILobFileNameSet() を使用して、表内の FILE を OS ファイルに関連付けることもできます。通常、BFILENAME() は INSERT または UPDATE 内でバインド変数なしで使用され、OCILobFileNameSet() がバインド変数に使用されます。

関連項目： 詳細は、16-49 ページの [OCILobFileNameSet\(\)](#) を参照してください。BFILENAME() 関数の詳細は、『Oracle9i アプリケーション開発者ガイド - ラージ・オブジェクト』を参照してください。

オブジェクトの LOB 属性

OCI アプリケーションでは、OCIObjectNew() を使用して LOB 属性を持つ永続オブジェクトまたは一時オブジェクトを作成することができます。

オブジェクトの LOB 属性への書込み

OCI を使用して、LOB 属性を持つ新規の永続オブジェクトを作成したり、その LOB 属性への書込みができます。アプリケーションでは次のステップに従います。

1. OCIObjectNew() をコールして、LOB 属性を持つ永続オブジェクトを作成します。
2. オブジェクトを使用済みとマークします。

3. 表に行を挿入して、オブジェクトをフラッシュします。
4. オブジェクトをデータベースから取り出し、LOB に有効なロケータを取得して、オブジェクトの最新バージョンを再確保します（またはオブジェクトをリフレッシュします）。
5. オブジェクトの LOB ロケータを使用して `OCILobWrite()` をコールし、データを書き込みます。

関連項目： マーク、フラッシュ、リフレッシュなどのオブジェクト操作の詳細は、[第 10 章「OCI オブジェクト・リレーショナル・プログラミング」](#) およびそれ以降の章を参照してください。

LOB 属性を持つ一時オブジェクト

アプリケーションでは、`OCIObjectNew()` をコールして、内部 LOB (BLOB、CLOB、NCLOB) 属性を持つ一時オブジェクトを作成できます。ただし、一時 LOB は現在サポートされていないため、LOB 属性での操作（読み込みや書き込みなど）は実行できません。一時内部 LOB 型を作成するための `OCIObjectNew()` コールは失敗しませんが、アプリケーションでは、この一時 LOB で LOB 操作を使用することはできません。

ただし、アプリケーションでは、FILE 属性を持つ一時オブジェクトを作成し、FILE 属性を使用して、サーバーのファイル・システムに格納されているファイルからデータを読み込みます。アプリケーションでは、`OCIObjectNew()` をコールして一時 FILE を作成し、その FILE でサーバーのファイルから読み込むことができます。

LOB の配列インタフェース

LOB とともに、OCI の配列インタフェースを他のデータ型と同じように使用することができます。ただし、記述子を割り当てるには次のように行う必要があります。

```
/* First create an array of OCILocator pointers: */
OCILobLocator *lobp[10];

for (i=0; i < 10; i++)
{ OCIDescriptorAlloc (...,&lobp[i],...);

/* Then bind the descriptor as follows */
  OCIBindByPos(... &lobp[i], ...);
}
```

LOB および FILE の関数

表 7-1 の関数は、LOB および FILE で操作するために使用できます。

関連項目： 各関数に関する詳細は、16-22 ページの「[LOB 関数](#)」を参照してください。これらの LOB および FILE コールは、アプリケーションが Oracle 7 Server に接続している場合は無効です。

データへのオフセットを伴うすべての LOB 操作では、オフセットは 1 から開始されます。OCILOBCopy()、OCILOBErase()、OCILOBLoadFromFile()、OCILOBTrim() などの LOB 操作の場合、*amount* パラメータは、クライアント側のキャラクタ・セットにかかわらず、CLOB と NCLOB の文字数です。

これらの LOB 操作は、サーバー上の LOB データの量を参照します。クライアント側のキャラクタ・セットが可変幅の場合は、次の一般的な規則が LOB コールの *amount* パラメータと *offset* パラメータに適用されます。

- *amount* — *amount* パラメータがサーバー側の LOB を参照する場合、*amount* は文字数です。*amount* パラメータがクライアント側のバッファを参照する場合、*amount* はバイト数です。
- *offset* — クライアント側のキャラクタ・セットが可変幅かどうかにかかわらず、*offset* パラメータは常に CLOB および NCLOB の場合は文字数で、BLOB および BFILE の場合はバイト数です。

これらの一般的な規則の例外は、特定の LOB コールの説明で示してあります。

関連項目： FILE の詳細は、『Oracle9i アプリケーション開発者ガイド - ラージ・オブジェクト』の BFILE に関する説明を参照してください。

表 7-1 OCI LOB および FILE の関数

関数	制限	用途
OCILOBAppend()	内部 LOB のみ	ある内部 LOB のデータを別の内部 LOB に追加します。LOB の追加元および追加先は、すでに存在する必要があります。新規に書き込むデータが宛先 LOB の現行の長さよりも大きい場合、宛先 LOB は、そのデータにあわせて拡張されます。追加先 LOB を最大長 (4GB) を超えて拡張したり、NULL である LOB から追加しようとすると、エラーになります。

表 7-1 OCI LOB および FILE の関数（続き）

関数	制限	用途
<code>OCILobAssign()</code>		LOB/FILE ロケータを別のロケータに割り当てます。この関数は、一時 LOB には使用できません。 <code>OCILobLocatorAssign()</code> を使用してください。
<code>OCILobCharSetForm()</code>		CLOB/NCLOB のキャラクタ・セット・フォームを入手します。
<code>OCILobCharSetId()</code>		CLOB/NCLOB のキャラクタ・セット ID を入手します。
<code>OCILobClose()</code>		オープンされた LOB または BFILE をクローズします。
<code>OCILobCopy()</code>	内部 LOB のみ	この関数は、内部 LOB の一部を別の内部 LOB にコピーします。コピー元 LOB およびコピー先 LOB は、すでに存在する必要があります。データが宛先の開始位置にすでに存在する場合、ソース・データで上書きされます。宛先の開始位置がカレント設定値の終了地点を超える場合は、宛先の設定値の終了地点から新たにソースから書き込まれるデータの開始地点までの LOB に、0（ゼロ）バイト充填文字（BLOB）か空白（CLOB/NCLOB）が配置されます。新規に書き込むデータが宛先 LOB の現行の長さよりも大きい場合、宛先 LOB は、そのデータにあわせて拡張されます。コピー先 LOB を最大長（4GB）を超えて拡張すると、エラーになります。LOB のコピー操作は、同じ型の LOB で実行してください。たとえば、ある CLOB は別の CLOB にコピーでき、ある BLOB は別の BLOB にコピーできますが、CLOB は BLOB にコピーできず、その反対もできません。
<code>OCILobCreateTemporary()</code>		一時 LOB を作成します。
<code>OCILobDisableBuffering()</code>	内部 LOB のみ	指定の内部ロケータについて、LOB のバッファリングを使用禁止にします。
<code>OCILobEnableBuffering()</code>	内部 LOB のみ	指定の内部ロケータについて、LOB のバッファリングを使用可能にします。

表 7-1 OCI LOB および FILE の関数（続き）

関数	制限	用途
<code>OCILobErase()</code>	内部 LOB のみ	指定されたオフセットから始まる指定された部分を内部 LOB 値から消去します。実際に消去した文字数またはバイト数が戻ります。要求した文字数またはバイト数を消去する前に LOB データの終了位置に達すると、実際に消去した文字数またはバイト数と要求した文字数またはバイト数は異なる値になります。LOB が NULL の場合、このルーチンは、0（ゼロ）文字 / バイトを消去したことを表示します。
<code>OCILobFileClose()</code> 、 <code>OCILobFileCloseAll()</code>		以前にオープンした FILE またはオープンしているすべての FILE をクローズします。内部 LOB に対してこの関数をコールするとエラーになります。FILE が存在していてオープンしていないときは、エラーは戻りません。
<code>OCILobFileExists()</code>		FILE がサーバー上に存在しているかどうかをテストします。
<code>OCILobFileNameGet()</code>		FILE の名前とディレクトリ別名を入手します。
<code>OCILobFileIsOpen()</code>		FILE が入力ロケータでオープンされているかどうかをテストします。
<code>OCILobFileOpen()</code>		FILE をオープンします。FILE は、読取り専用アクセス用にオープンできます。この関数を内部 LOB に対してコールするとエラーになります。
<code>OCILobFileNameSet()</code>		FILE の名前とディレクトリ別名を設定します。
<code>OCILobFlushBuffer()</code>	内部 LOB のみ	LOB のバッファをフラッシュします。
<code>OCILobFreeTemporary()</code>		一時 LOB の値を解放します。
<code>OCILobGetChunkSize()</code>		利用可能な LOB のチャンク・サイズを入手します。

表 7-1 OCI LOB および FILE の関数（続き）

関数	制限	用途
<code>OCILobGetLength()</code>		この関数は、LOB または FILE の長さを入手します。LOB または FILE が NULL の場合、長さは未定義です。空の内部 LOB の長さは 0（ゼロ）です。クライアント側キャラクタ・セットが可変幅かどうかには関係なく、出力の長さは、CLOB および NCLOB の場合は文字数で、BLOB および BFILE の場合はバイト数です。
<code>OCILobIsEqual()</code>		2 つの LOB/FILE ロケータが等しいかどうかをテストします。2 つのロケータが等しいのは、両方のロケータが同じ LOB/FILE 値を参照している場合のみです。
<code>OCILobIsOpen()</code>		LOB がオープンされているかどうかテストします。
<code>OCILobIsTemporary()</code>		一時 LOB であるかどうかテストします。
<code>OCILobLoadFromFile()</code>		FILE からのデータを持つ LOB のすべてまたは一部を移行します。
<code>OCILobLocatorAssign()</code>		LOB/FILE ロケータを別の LOB/FILE ロケータに割り当てます。
<code>OCILobLocatorIsInit()</code>		LOB/FILE ロケータが初期化されているかどうかをテストします。
<code>OCILobOpen()</code>		LOB または BFILE をオープンします。

表 7-1 OCI LOB および FILE の関数（続き）

関数	制限	用途
<code>OCILobRead()</code>		この関数は、LOB または FILE 値の一部をバッファへ読み込みます。NULL 属性の LOB または FILE から読み込むとエラーになります。クライアント側 キャラクタ・セットが可変幅の場合は、CLOB および NCLOB の入力量は文字数、出力量はバイト数です。入力量は、サーバー側 CLOB/NCLOB から読み込まれた文字数を指します。出力量は、バッファ <i>bufp</i> に読み込まれたバイト数を示します。ポーリング・モードを使用している場合は、バッファ全体が埋められていない可能性があるため、各 <code>OCILobRead()</code> コール後に <i>amtp</i> パラメータの値を見て、バッファに読み込まれたバイト数を調べます。コールバックを使用する場合、 <i>len</i> パラメータはコールバックへの入力で、バッファ内で格納されたバイト数を示します。バッファ全体がデータで埋められていない可能性があるため、コールバック処理中に <i>len</i> パラメータをチェックしてください。
<code>OCILobTrim()</code>	内部 LOB のみ	この関数は、LOB の値を指定された長さに短く切り詰めて、LOB を切り捨てます。
<code>OCILobWrite()</code>	内部 LOB のみ	この関数は、バッファのデータを内部 LOB に書き込みます。データがすでに LOB の中に存在する場合は、バッファに格納されていたデータによって上書きされます。クライアント側 キャラクタ・セットが可変幅の場合は、CLOB および NCLOB の入力量はバイト数、出力量は文字数です。入力量は、LOB に書き込まれるデータのバイト数を指します。出力量は、サーバー側 CLOB/NCLOB に書き込まれる文字数を指します。
<code>OCILobWriteAppend()</code>		LOB の末尾からデータを書き込みます。

LOB の読み込み / 書き込みパフォーマンスを向上するための関数

OCILobGetChunkSize() の使用方法

OCILobGetChunkSize() コールを利用して、LOB の読み込みおよび書き込み操作のパフォーマンスを向上させることができます。OCILobGetChunkSize() は、利用可能なチャンク・サイズを BLOB に対してはバイト数で、CLOB および NCLOB に対しては文字数で戻します。サイズが利用可能なチャンク・サイズの倍数で、チャンクの境界から開始するデータを使用して読み込みまたは書き込みが行われると、パフォーマンスが向上します。LOB を含む表の作成時に、LOB 列に対してチャンク・サイズを指定できます。

OCILobGetChunkSize() のコールにより、LOB の利用可能なチャンク・サイズが戻されるため、アプリケーションは、同じチャンク上で実行される複数の LOB 書き込みコールを発行するかわりに、チャンク全体が書き込まれるまで、一連の書き込み操作をバッチ処理できます。

LOB の末尾までを読み込むには、4GB を使用して OCILobRead() をコールします。4GB を使用した OCILobRead() は LOB の末尾に到達するまで読み込みを行うため、最初に OCILobGetLength() をコールすることによるラウンドトリップを避けることができます。

注意： OCILobGetChunkSize() は、可変幅の文字を格納する LOB に対して LOB チャンクに相当する Unicode 文字の数を戻します。

OCILobWriteAppend() の使用方法

OCI では、LOB の末尾へのデータの書き込みをより効率的にするためのショートカットが提供されています。OCILobWriteAppend() を使用すると、アプリケーションは、最初に OCILobGetLength() をコールして OCILobWrite() の開始点を決定しなくても、データを LOB の末尾に追加できます。OCILobWriteAppend() で、両方のステップが行われます。

LOB バッファリング関数

Oracle OCI では、内部 LOB 値の小さな読み込みと書き込みのために、次のように、LOB のバッファリングを制御するコールを提供します。

- `OCILobEnableBuffering()`
- `OCILobDisableBuffering()`
- `OCILobFlushBuffer()`

これらの関数により、アプリケーションで内部 LOB (BLOB、CLOB、NCLOB) を使用して、クライアント側のバッファで LOB の小さな読み込みと書き込みをバッファできるため、パフォーマンスが向上します。これにより、ネットワーク・ラウンドトリップの回数と LOB のバージョンを削減でき、小さな読み込みと書き込みについては、LOB のパフォーマンスが向上します。

関連項目：

- LOB バッファリングの詳細は、『Oracle9i アプリケーション開発者ガイド - ラージ・オブジェクト』の LOB に関する章を参照してください。
- LOB バッファリングの使用方法を示したコード例については、Oracle のインストールに付属するデモ・プログラムを参照してください。付録 B「OCI デモ・プログラム」を参照してください。

LOB のオープンおよびクローズのための関数

OCI では、LOB を明示的にオープン (`OCILobOpen()`) およびクローズ (`OCILobClose()`) するための関数と、特定の LOB がすでにオープンされているかをテストするための関数 (`OCILobIsOpen()`) が提供されています。これらの関数を使用すると、アプリケーションで一連の LOB 操作の開始および終了をマークできるため、LOB のクローズ時に索引の更新など特定の処理が実行できます。

注意： 内部 LOB では、オープンされているかどうかの概念は、ロケータではなく LOB に関連しています。ロケータには、ロケータが参照する LOB がオープンされているかどうかの情報は格納されません。オープンされている同じ LOB を、複数のロケータが指し示すことが可能です。ただし、BFILE については、オープンは特定のロケータに関連付けられます。したがって、別のロケータを使用して、同じ BFILE に対して複数のオープンが実行されることがあります。

アプリケーションで、LOB 操作を `OCILobOpen()` コールと `OCILobClose()` コールのセットで囲んでいない場合は、LOB が変更されるたびに LOB が暗黙的にオープンおよびクローズされ、それによって LOB の変更に関連付けられているトリガーが起動します。

注意： LOB 操作がオープン・コールとクローズ・コールで囲まれていない場合は、LOB の変更時に、LOB の拡張可能な索引がすべて更新されます。このため、索引は常に有効で、任意のタイミングで使用できます。

OCILOBOpen() コールと OCILOBClose() コールで囲まれている LOB が変更された場合、個々の LOB の変更に対してトリガーは起動されません。トリガーは OCILOBClose() コールの後にのみ起動されるため、索引が更新されるのはクローズ・コールの後です。したがって、索引はオープン・コールからクローズ・コールまでの間は無効です。OCILOBIsOpen() は、内部および外部の LOB (BFILE) とともに使用できます。

トランザクションによってオープンされたすべての LOB をクローズする前にトランザクションをコミットすると、エラーが発生します。このエラーが戻されると、LOB はオープンとしてマークされなくなりますが、トランザクションは正常にコミットされます。したがって、トランザクションで LOB データと LOB 以外のデータに対して行われた変更はすべてコミットされます。ただし、ドメイン索引とファンクション索引は更新されません。このエラーが戻された場合は、LOB 列のファンクション索引とドメイン索引を再作成してください。

トランザクションがない場合にオープンしている LOB は、セッションの終了前にクローズする必要があります。セッションの終了時にオープンしている LOB がある場合、その LOB はオープンとしてマークされなくなり、ドメイン索引とファンクション索引は更新されません。このエラーが戻された場合は、LOB 列のファンクション索引とドメイン索引を再作成してください。

制限

LOB のオープンとクローズに関するメカニズムには、次の制限があります。

1. アプリケーションでは、トランザクションをコミットする前に、以前にオープンした LOB をすべてクローズする必要があります。これを行わないと、エラーが発生します。トランザクションがロールバックされた場合は、変更とともに、オープンしているすべての LOB が破棄され (LOB はクローズされません)、関連付けられているトリガーは起動しません。
2. 内部 LOB のオープン数に制限はありませんが、FILE のオープン数には制限があります。『Oracle9i データベース・リファレンス』の SESSION_MAX_OPEN_FILES パラメータを参照してください。すでにオープンしているロケータを別のロケータに割り当てても、新規 LOB のオープンとしてはカウントされないことに注意してください。
3. 同じトランザクション内で、別のロケータまたは同じロケータを使用して同じ内部 LOB を 2 度オープンまたはクローズすると、エラーになります。
4. オープンしていない LOB をクローズするとエラーになります。

注意： オープンしている LOB の値をクローズする必要があるトランザクションの定義は、次のいずれかです。

- SET TRANSACTION と COMMIT の間
 - DATA MODIFYING DML または SELECT ... FOR UPDATE と COMMIT の間
 - 自律型トランザクション・ブロック内
-
-

LOB のオープン / クローズの例

関連項目： OCILobOpen() コールおよび OCILobClose() コールの使用例については、[付録 B「OCI デモ・プログラム」](#) のオンライン・デモ・プログラムのリストを参照してください。

LOB 関数用のサーバー・ラウンドトリップ

関連項目： 個々の OCI LOB 関数に必要なサーバー・ラウンドトリップ回数については、[付録 C「OCI 関数のサーバー・ラウンドトリップ」](#) の表を参照してください。

LOB 読み込みおよび書き込みコールバック

OCI LOB の読み込みと書き込み関数により、書き込みデータを提供したり読み込みデータを処理するのに使用できるコールバック関数を定義します。これによって、クライアント・アプリケーションでは、データでオプション処理を実行できます。この使用例としては、データを書き込むための圧縮アルゴリズムと、読み込むための圧縮解凍アルゴリズムを実装するコールバックの使用があげられます。

注意： LOB 読み込み / 書き込みストリーム転送のコールバックにより、大量の LOB データの読み込み / 書き込みが高速になります。

次の項では、コールバックの使用方法を詳しく説明します。

ストリーム転送のコールバック・インタフェース

使用しているアプリケーションで、ユーザー定義の読み込みおよび書き込み用コールバック関数を使用して、データを LOB へ挿入したり LOB から取り出すことができます。データを LOB へストリーム転送したり、LOB からデータを取り出すためのポーリング・メソッドにかわる方法を提供します。ユーザー定義のコールバックには、次に説明する特定のプロトタイプがあります。これらの関数はプログラマーが実装し、OCILOBRead() コールおよび OCILOBWrite() コールを通じて OCI に登録します。コールバック関数は、必要に応じて OCI でコールします。

コールバックを使用した LOB の読み込み

ユーザー定義の読み込みコールバック関数は、OCILOBRead() 関数を通じて登録されます。コールバック関数には次のプロトタイプが必要です。

```
CallbackFunctionName ( dvoid *ctxp, CONST dvoid *bufp, ub4 buflen, ub1 piece);
```

最初のパラメータの *ctxp* はコールバックのコンテキストで、OCILOBRead() ファンクション・コールで OCI に渡されます。コールバック関数がコールされると、*ctxp* で提供した情報が戻されます（OCI が IN の途中でこの情報を使用することはありません）。*bufp* パラメータは、LOB データが戻される格納場所へのポインタで、*buflen* はこのバッファの長さを表します。これにより、指定したバッファに読み込まれたデータの量がわかります。

元の OCILOBRead() コールで提供したバッファの長さが、サーバーから戻されるデータをすべて格納するのに十分でない場合は、ユーザー定義コールバックがコールされます。この場合、*piece* パラメータにより、バッファに戻された情報が最初のピース、次のピースまたは最後ピースのうちどの場所であるかが示されます。

次に、読み込みコールバック関数を実装するための一般的なコード・フラグメントを示します。ここでは *lob1* を前もって選択された有効なロケータと想定します。*svchp* は有効なサービス・ハンドルで、*errhp* は有効なエラー・ハンドルです。

```
...
ub4   offset = 1;
ub4   loblen = 0;
ub1   bufp[MAXBUFLen];
ub4   amtp = 0;
sword retval;
amtp = 4294967295;          /* 4 gigabytes minus 1 */
if (retval = OCILOBRead(svchp, errhp, lob1, &amtp, offset, (dvoid *) bufp,
    (ub4) MAXBUFLen, (dvoid *) bufp, cbk_read_lob,
    (ub2) 0, (ub1) SQLCS_IMPLICIT))
{
    (void) printf("ERROR: OCILOBRead() LOB.\n");
    report_error();
}
...
sb4 cbk_read_lob(ctxp, bufxp, len, piece)
```

```

dvoid *ctxp;
CONST dvoid *bufxp;
ub4 len;
ub1 piece;
{
static ub4 piece_count = 0;
piece_count++;
switch (piece)
{
case OCI_LAST_PIECE:
/* process buffer bufxp */
--- buffer processing code goes here ---
(void) printf("callback read the %d th piece\n", piece_count);
piece_count = 0;
break;
case OCI_FIRST_PIECE:
case OCI_NEXT_PIECE:
/* process buffer bufxp */
--- buffer processing code goes here ---
(void) printf("callback read the %d th piece\n", piece_count);
break;
default:
(void) printf("callback read error: unkown piece = %d.\n", piece);
return OCI_ERROR;
}
return OCI_CONTINUE;
}

```

前述の例では、すべての LOB データが読み込まれるまで、ユーザー定義関数 `cbk_read_lob()` が繰り返しコールされます。

関連項目： ポーリングおよびコールバックを使用した `OCILobRead()` の使用例については、[付録 B「OCI デモ・プログラム」](#) のオンライン・デモ・プログラムのリストを参照してください。

コールバックを使用した LOB の書き込み

読み込みコールバックと同様に、ユーザー定義の書き込みコールバック関数が `OCILobWrite()` 関数を通じて登録されます。コールバック関数には次のプロトタイプが必要です。

```
CallbackFunctionName ( dvoid *ctxp, dvoid *bufp, ub4 *buf1, ub1 *piecep);
```

最初のパラメータの `ctxp` はコールバックのコンテキストで、`OCILobWrite()` ファンクション・コールで OCI に渡されます。OCI でコールバック関数がコールされると、`ctxp` で提供した情報が戻されます (OCI が IN の途中でこの情報を使用することはありません)。`bufp` パラメータは記憶領域へのポインタです。このポインタは、`OCILobWrite()` へのコールで指定されます。コールで指定するデータを `OCILobWrite()` に挿入した後も、書き込むデータがまだ存在する場合は、ユーザー定義コールバックがコールされます。コールバックでは、`bufp` によって示す記憶域に挿入するデータを指定し、またその長さを `buf1` に指定します。さらに、`piecep` パラメータを使用して、次のピース (`OCI_NEXT_PIECE`) か最後のピース (`OCI_LAST_PIECE`) か示す必要があります。アプリケーションで提供する記憶域ポインタに対する全責任はプログラマにあるため、割り当てられた格納領域のサイズ以上は書き込まないように注意してください。

次に、書き込みコールバック関数を実装するための一般的なコード・フラグメントを示します。

ここでは `lob1` を更新用にロックされた有効なロケータと想定します。`svchp` は有効なサービス・ハンドルで、`errhp` は有効なエラー・ハンドルです。

```
...
ub4   offset = 1;
ub1   bufp[MAXBUFLN];
ub4   amtp = MAXBUFLN * 20;
ub4   nbytes = MAXBUFLN;
/* Fill bufp with some data */
-- code to fill bufp with data goes here. nbytes should reflect the size and should
be less than or equal to MAXBUFLN --
if (retval = OCILobWrite(svchp, errhp, lob1, &amtp, offset, (dvoid*)
    bufp, (ub4)nbytes, OCI_FIRST_PIECE, (dvoid *)0, cbk_write_lob,
    (ub2) 0, (ub1) SQLCS_IMPLICIT))
{
    (void) printf("ERROR: OCILobWrite().\n");
    report_error();
    return;
}
...
sb4 cbk_write_lob(ctxp, bufxp, lenp, piecep)
dvoid *ctxp;
dvoid *bufxp;
ub4 *lenp;
ub1 *piecep;
{
```

```

/* Fill bufxp with data */
-- code to fill bufxp with data goes here. *lenp should reflect the size
and should be less than or equal to MAXBUFLLEN --
if (this is the last data buffer)
    *piecep = OCI_LAST_PIECE;
else
    *piecep = OCI_NEXT_PIECE;;
return OCI_CONTINUE;
}

```

前述の例では、*piecep* パラメータを使用して、アプリケーションが最後のピースを提供中であることを示すまで、ユーザー定義関数 `cbk_write_lob()` が繰り返しコールされます。

関連項目： ポーリングおよびコールバックを使用した `OCILobWrite()` の使用例については、[付録 B「OCI デモ・プログラム」](#) のオンライン・デモ・プログラムのリストを参照してください。

一時 LOB のサポート

OCI では、一時 LOB の作成および解放のための関数 `OCILobCreateTemporary()` および `OCILobFreeTemporary()` と、特定の LOB が一時 LOB であるかどうかを問い合わせる関数 `OCILobIsTemporary()` が提供されています。

一時 LOB は、データベース内に永続的に格納されることはありませんが、LOB データを操作するために、ローカル変数のように動作します。標準（永続）LOB 上で動作する OCI 関数は、一時 LOB 上でも使用することができます。

標準 LOB と同様に、すべての関数は一時 LOB のロケータ上で動作し、実際の LOB データにはロケータを介してアクセスされます。

一時 LOB ロケータは、次の型の SQL 文への引数として使用できます。

- UPDATE - 一時 LOB ロケータは、NULL かどうかのテスト時に WHERE 句内の値として使用するか、あるいは関数へのパラメータとして使用できます。一時 LOB ロケータは、SET 句内で使用することもできます。
- DELETE - 一時 LOB ロケータは、NULL かどうかのテスト時に WHERE 句内で使用するか、あるいは関数へのパラメータとして使用できます。
- SELECT - 一時 LOB ロケータは、NULL かどうかのテスト時に WHERE 句内で使用するか、あるいは関数へのパラメータとして使用できます。また、一時 LOB は、関数の戻り値を選択する場合、SELECT...INTO 文で戻される変数として使用することもできます。

- **注意：**永続ロケータを一時ロケータに選択した場合、一時ロケータは永続ロケータによって上書きされます。この場合、一時 LOB は暗黙的には解放されません。ユーザーは `SELECT...INTO` の前に一時 LOB を明示的に解放する必要があります。明示的に解放しなければ、一時 LOB は継続時間の終了時まで解放されません。同じ LOB を指し示している別の一時ロケータがある場合を除き、元のロケータは `SELECT...INTO` によって上書きされるため、その一時 LOB を指し示すロケータは存在しないことになります。

一時 LOB の作成および解放

`OCILobCreateTemporary()` 関数を使用して一時 LOB を作成します。この関数に渡されるパラメータには、LOB の継続時間を示す値が含まれています。デフォルトの継続時間は、カレント・セッションの長さです。継続時間の終了時に、すべての一時 LOB が削除されます。ユーザーは、`OCILobFreeTemporary()` 関数で一時 LOB を明示的に解放することによって、一時 LOB の領域を再生できます。一時 LOB は、作成されたときは空の状態です。

一時 LOB を作成するとき、その一時 LOB がサーバーのバッファ・キャッシュに読み込まれるかどうかも指定できます。

一時 LOB を永続的にするには、アプリケーションで `OCILobCopy()` を使用して、データを一時 LOB から永続 LOB にコピーできます。また、アプリケーションでは、一時 LOB を `INSERT` 文の `VALUES` 句で使用したり、一時 LOB を `UPDATE` 文内の割当てのソースとして使用したり、あるいは一時 LOB を通常の永続 LOB 属性に割り当てて、オブジェクトをフラッシュできます。

一時 LOB は、標準 LOB の変更に使用する関数と同じ関数を使用して変更できます。

一時 LOB の継続時間

OCI では、いくつかの一時 LOB 用の事前定義済み継続時間、およびアプリケーションでアプリケーション固有の継続時間を定義するために使用できる関数のセットが用意されています。事前定義済みの継続時間は次のとおりです。

1. コール (`OCI_DURATION_CALL`)、サーバー側のみ
2. セッション (`OCI_DURATION_SESSION`)

セッション継続時間は、セッションまたは接続が終了した時点で期限切れになります。コール継続時間は、現行の OCI コールが終了した時点で期限切れになります。

オブジェクト・モードで実行している場合は、アプリケーション固有の継続時間も定義できます。アプリケーション固有の継続時間はユーザー期間とも呼ばれ、`OCIDurationBegin()` 関数を使用して継続開始時間を指定し、`OCIDurationEnd()` 関数を使用して継続終了時間を指定することによって定義されます。

注意： ユーザー定義の継続時間は、アプリケーションがオブジェクト・モードで初期化されている場合にものみ利用できます。

アプリケーション固有の継続時間にはそれぞれ、`OCIDurationBegin()` で戻される継続時間識別子があり、継続時間の `OCIDurationEnd()` がコールされるまで一意であることが保証されます。アプリケーション固有の継続時間は、セッションの継続時間と同じに設定できますが、それ以上長くはできません。

継続時間の終了時には、その継続時間に関連付けられたすべての一時 LOB が解放されます。一時 LOB に関連付けられた記述子は、`OCIDescriptorFree()` コールを使用して明示的に解放する必要があります。

ユーザー定義の継続時間ではネストが可能です。つまり、ある継続時間を別のユーザー期間の子の継続時間として定義できます。親の継続時間が子の継続時間を持ち、その子の継続時間がそれぞれ順に子の継続時間を持つことが可能です。

注意： 継続時間が `OCIDurationBegin()` で開始されると、パラメータの 1 つは親の継続時間の識別子となります。親の継続時間が終了すると、子の継続時間もすべて終了します。詳細は、16-24 ページの [OCIDurationBegin\(\)](#) を参照してください。

一時 LOB の例

次のコードは、一時 LOB の使用例を示しています。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <oci.h>

/* Function Prototype */
static void checkerr (/*_ OCIError *errhp, sword status _*/);
sb4 select_and_createtemp (OCILobLocator *lob_loc,
                           OCIError      *errhp,
                           OCISvcCtx     *svchp,
                           OCISstmt     *stmthp,
                           OCIEnv       *envhp);

/* This function reads in a single video Frame from the Multimedia_tab table. Then
it creates a temporary lob. The temporary LOB which is created is read through the
CACHE, and is automatically cleaned up at the end of the user's session, if it is
not explicitly freed sooner. This function returns OCI_SUCCESS if it completes
successfully or OCI_ERROR if it fails. */

sb4 select_and_createtemp (OCILobLocator *lob_loc,
                           OCIError      *errhp,
                           OCISvcCtx     *svchp,
                           OCISstmt     *stmthp,
                           OCIEnv       *envhp)
```

```
{
    OCIDefine      *defnp1;
    OCIBind        *bndhp;
    text          *sqlstmt;
    int rowind =1;
    ub4 loblen = 0;
    OCILobLocator *tblob;
    printf ("in select_and_createtemp \n");
    if(OCIDescriptorAlloc((dvoid*)envhp, (dvoid **)&tblob,
                          (ub4)OCI_DTYPE_LOB, (size_t)0,
                          (dvoid**)0))
    {
        printf("failed in OCIDescriptor Alloc in select_and_createtemp \n");
        return OCI_ERROR;
    }
    /* arbitrarily select where Clip_ID =1 */
    sqlstmt = (text *)"SELECT Frame FROM Multimedia_tab WHERE Clip_ID = 1 FOR UPDATE";
    if (OCISTmtPrepare(stmthp, errhp, sqlstmt,
                      (ub4) strlen((char *)sqlstmt),
                      (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCISTmtPrepare() sqlstmt\n");
        return OCI_ERROR;
    }
    /* Define for BLOB */
    if (OCIDefineByPos(stmthp,
                      &defnp1,
                      errhp,
                      (ub4) 1,
                      (dvoid *) &lob_loc,
                      (sb4)0,
                      (ub2) SQLT_BLOB,
                      (dvoid *) 0,
                      (ub2 *) 0,
                      (ub2 *) 0,
                      (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: Select locator: OCIDefineByPos()\n");
        return OCI_ERROR;
    }
    /* Execute the select and fetch one row */
    if (OCISTmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                      (CONST OCISnapshot*) 0, (OCISnapshot*) 0,
                      (ub4) OCI_DEFAULT))
    {
        (void) printf("FAILED: OCISTmtExecute() sqlstmt\n");
        return OCI_ERROR;
    }
}
```

```

    }
    if(OCILobCreateTemporary(svchp,
                             errhp,
                             tblob,
                             (ub2)0,
                             SQLCS_IMPLICIT,
                             OCI_TEMP_BLOB,
                             OCI_ATTR_NOCACHE,
                             OCI_DURATION_SESSION))
    {
        (void) printf("FAILED: CreateTemporary() \n");
        return OCI_ERROR;
    }
    if (OCILobGetLength(svchp, errhp, lob_loc, &loblen) != OCI_SUCCESS)
    {
        printf("OCILobGetLength FAILED\n");
        return OCI_ERROR;
    }
    if (OCILobCopy(svchp, errhp, tblob, lob_loc, (ub4)loblen, (ub4) 1,
                  (ub4) 1))
    {
        printf("OCILobCopy FAILED \n");
    }
    if(OCILobFreeTemporary(svchp, errhp, tblob))
    {
        printf("FAILED: OCILobFreeTemporary call \n");
        return OCI_ERROR;
    }
    return OCI_SUCCESS;
}

int main(char *argv, int argc)
{
    /* OCI Handles */

    OCIEnv      *envhp;
    OCIServer    *srvhp;
    OCISvcCtx    *svchp;
    OCIError     *errhp;
    OCISession   *authp;
    OCISstmt     *stmthp;
    OCILobLocator *clob, *blob;
    OCILobLocator *lob_loc;
    int type =1;
    /* Initialize and Logon */
    (void) OCIInitialize((ub4) OCI_DEFAULT, (dvoid *)0,
                        (dvoid * (*)(dvoid *, size_t)) 0,
                        (dvoid * (*)(dvoid *, dvoid *, size_t))0,

```

```
(void (*)(dvoid *, dvoid *)) 0 );
(void) OCIEnvInit( (OCIEnv **) &envhp,
                  OCI_DEFAULT, (size_t) 0,
                  (dvoid **) 0 );
(void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, OCI_HTYPE_ERROR,
                      (size_t) 0, (dvoid **) 0);
/* server contexts */
(void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, OCI_HTYPE_SERVER,
                      (size_t) 0, (dvoid **) 0);
/* service context */
(void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, OCI_HTYPE_SVCCTX,
                      (size_t) 0, (dvoid **) 0);
/* attach to Oracle */
(void) OCIServerAttach( srvhp, errhp, (text *)"", strlen(""), 0);
/* set attribute server context in the service context */
(void) OCIAttrSet((dvoid *) svchp, OCI_HTYPE_SVCCTX,
                 (dvoid *)srvhp, (ub4) 0,
                 OCI_ATTR_SERVER, (OCIError *) errhp);
(void) OCIHandleAlloc((dvoid *) envhp,
                     (dvoid **)&authp, (ub4) OCI_HTYPE_SESSION,
                     (size_t) 0, (dvoid **) 0);
(void) OCIAttrSet((dvoid *) authp, (ub4) OCI_HTYPE_SESSION,
                 (dvoid *) "scott", (ub4) 5,
                 (ub4) OCI_ATTR_USERNAME, errhp);
(void) OCIAttrSet((dvoid *) authp, (ub4) OCI_HTYPE_SESSION,
                 (dvoid *) "tiger", (ub4) 5,
                 (ub4) OCI_ATTR_PASSWORD, errhp);
/* Begin a User Session */
checkerr(errhp, OCISessionBegin ( svchp, errhp, authp, OCI_CRED_RDBMS,
                                (ub4) OCI_DEFAULT));
(void) OCIAttrSet((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX,
                 (dvoid *) authp, (ub4) 0,
                 (ub4) OCI_ATTR_SESSION, errhp);
/* ----- Done login in ----- */
/* allocate a statement handle */
checkerr(errhp, OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &stmthp,
                                OCI_HTYPE_STMT, (size_t) 0, (dvoid **) 0));
checkerr(errhp, OCIDescriptorAlloc((dvoid *)envhp, (dvoid **) &lob_loc,
                                   (ub4) OCI_DTYPE_LOB,
                                   (size_t) 0, (dvoid **) 0));

/* Subroutine calls begin here */
printf("calling select_and_createtemp\n");
select_and_createtemp (lob_loc, errhp, svchp,stmthp,envhp);

return 0;
}
void checkerr(errhp, status)
```

```
OCIError *errhp;
sword status;
{
    text errbuf[512];
    sb4 errcode = 0;
    switch (status)
    {
        case OCI_SUCCESS:
            break;
        case OCI_SUCCESS_WITH_INFO:
            (void) printf("Error - OCI_SUCCESS_WITH_INFO\n");
            break;
        case OCI_NEED_DATA:
            (void) printf("Error - OCI_NEED_DATA\n");
            break;
        case OCI_NO_DATA:
            (void) printf("Error - OCI_NODATA\n");
            break;
        case OCI_ERROR:
            (void) OCIErrorGet((dvoid *)errhp, (ub4) 1, (text *) NULL, &errcode,
                               errbuf, (ub4) sizeof(errbuf), OCI_HTYPE_ERROR);
            (void) printf("Error - %.*s\n", 512, errbuf);
            break;
        case OCI_INVALID_HANDLE:
            (void) printf("Error - OCI_INVALID_HANDLE\n");
            break;
        case OCI_STILL_EXECUTING:
            (void) printf("Error - OCI_STILL_EXECUTE\n");
            break;
        case OCI_CONTINUE:
            (void) printf("Error - OCI_CONTINUE\n");
            break;
        default:
            break;
    }
}
```

スケーラブルなプラットフォームの管理

この章は、次の項目で構成されています。

- [OCIでのトランザクションのサポート](#)
- [パスワードおよびセッションの管理](#)
- [中間層アプリケーション](#)
- [外部で初期化されたコンテキスト](#)

OCI でのトランザクションのサポート

OCI には、ローカル・トランザクションとグローバル・トランザクションの両方での操作に対応した一連の API コールが用意されています。これらのコールにはオブジェクトのサポートが含まれているため、OCI アプリケーションがオブジェクト・モードで実行されている場合はコミット・コールおよびロールバック・コールでオブジェクト・キャッシュとトランザクションの状態が同期します。

トランザクション操作を実行する関数は、下記のとおりです。それぞれのコールには、適切なサーバー・コンテキストおよびユーザー・セッション・ハンドルで初期化されるサービス・コンテキスト・ハンドルが必要です。トランザクション・ハンドルは、サービス・コンテキストの 3 番目の要素で、トランザクションに関する特定の情報を格納します。SQL 文が準備されると、トランザクション・ハンドルは、特定のサービス・コンテキストと関連付けられます。その文が実行されると、その結果（問合せ、フェッチ、挿入）は、サービス・コンテキストと現在関連しているトランザクションの一部になります。

- `OCITransStart()` – トランザクションの開始をマークします。
- `OCITransDetach()` – トランザクションを連結解除します。
- `OCITransCommit()` – トランザクションをコミットします。
- `OCITransRollback()` – トランザクションをロールバックします。
- `OCITransPrepare()` – 分散処理環境でトランザクションをコミットする準備をします。
- `OCITransMultiPrepare()` – 1 回のコールで、複数のブランチのあるトランザクションを準備します。
- `OCITransForget()` – 発見的方法で完了したグローバル・トランザクションの記憶をサーバーから消します。

これらのコールをすべて使用するか、そのうちのいくつかのみを使用するかは、アプリケーションでのトランザクションの複雑度のレベルによって異なります。次の項で、詳細に説明します。

関連項目： これらのコールに関する特定の情報は、16-154 ページの「[トランザクション関数](#)」の関数の説明を参照してください。

トランザクションの複雑度のレベル

OCI では、複数のレベルのトランザクションの複雑度をサポートします。それぞれのレベルについては、次の各項で説明します。

- [単純なローカル・トランザクション](#)
- [シリアル化可能または読取り専用のローカル・トランザクション](#)
- [グローバル・トランザクション](#)

単純なローカル・トランザクション

ほとんどのアプリケーションで扱うのは、単純なローカル・トランザクションのみです。これらのアプリケーションでは、そのアプリケーションの中でデータベースを変更すると、暗黙的トランザクションが作成されます。アプリケーションなどで必要とされるトランザクション特有のコールには、次のようなコールがあります。

- `OCITransCommit()` — トランザクションをコミットします。
- `OCITransRollback()` — トランザクションをロールバックします。

1 つのトランザクションがコミットまたはロールバックするとすぐに、次のデータベース変更によって、そのアプリケーションに対する新規の暗黙的トランザクションが作成されます。

1 つのサービス・コンテキストでは、常に 1 つの暗黙的トランザクションのみをアクティブにできます。暗黙的なトランザクションの属性は、ユーザーには不透明です。

アプリケーションで複数のセッションを作成すると、それぞれの権限に対し、関連する暗黙的トランザクションを 1 つずつ保持することができます。

関連項目： 単純なローカル・トランザクションの使用方法を示すコード
例は、16-155 ページの `OCITransCommit()` の例を参照してください。

シリアル化可能または読取り専用のローカル・トランザクション

アプリケーションでシリアル化可能または読取り専用トランザクションが必要な場合は、単純なローカル・トランザクションを操作するアプリケーションで必要とされる OCI コールの他に、追加の OCI コールが必要です。シリアル化可能または読取り専用トランザクションを初期化するには、トランザクションを開始する `OCITransStart()` をアプリケーションでコールして、トランザクションを作成する必要があります。

`OCITransStart()` コールでは、`flags` パラメータで、`OCI_TRANS_SERIALIZABLE` または `OCI_TRANS_READONLY` のいずれか適切な値を指定します。フラグが指定されない場合、デフォルト値は、標準の読取り / 書き込みトランザクションの `OCI_TRANS_READWRITE` です。

`OCITransStart()` コールで読取り専用オプションを指定すると、`SET TRANSACTION READ ONLY` 文を実行するためのサーバー・ラウンドトリップは必要ありません。

グローバル・トランザクション

グローバル・トランザクションは、より高度なトランザクションを処理するアプリケーションでのみ必要です。

注意： この項は、分散トランザクションまたはグローバル・トランザクション環境以外で操作を行う場合はスキップしてください。

この項では、最初にグローバル・トランザクションの背景について説明します。次に、OCI コールを使用してグローバル・トランザクションを処理する場合の詳細を説明します。

トランザクション識別子 トランザクション処理（TP）モニターなどの 3 層アプリケーションでは、グローバル・トランザクションの作成および管理を行います。これらのアプリケーションは、グローバル・トランザクション識別子（XID）を提供します。これによって、サーバーにローカル・トランザクションが関連付けられます。

グローバル・トランザクションには、1 つ以上のブランチがあります。それぞれのブランチは、XID によって識別されます。XID は、グローバル・トランザクション識別子（gtrid）とブランチ修飾子（bqual）で構成されています。この構造は、標準の XA 仕様に基づくものです。

次の例では、XID が 1234 の構造です。

コンポーネント	値
gtrid	12
bqual	34
gtrid+bqual=XID	1234

OCI トランザクション・コールで使用されるトランザクション識別子は、OCIAttrSet() を使用して、トランザクション・ハンドルの OCI_ATTR_XID 属性に設定します。かわりに、OCI_ATTR_TRANS_NAME 属性で設定した名前でもトランザクションを識別することもできます。

トランザクション・ブランチ Oracle では、単一のグローバル・トランザクションで、一対のブランチが密結合している場合と疎結合している場合の両方をサポートしています。

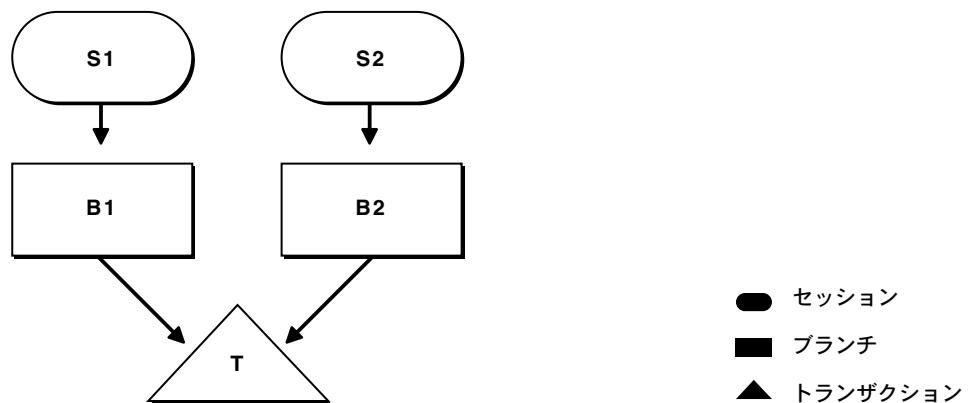
- 密結合ブランチとは、同じローカル・トランザクションを共有する個々のブランチです。この場合、*gtrid* は、単一のローカル・トランザクションを指し、複数のブランチが同じトランザクションを指し示します。トランザクションの所有者は、最初に作成されたブランチです。
- 疎結合ブランチとは、異なるローカル・トランザクションを使用する個々のブランチです。この場合は、*gtrid* と *bqual* がともに一意のローカル・トランザクションにマッピングされます。各ブランチは、異なるトランザクションを指し示します。

OCITransStart() の *flags* パラメータによって、アプリケーションから、OCI_TRANS_TIGHT または OCI_TRANS_LOOSE を渡して結合のタイプを指定できます。

セッションは OCISessionBegin() で作成され、ユーザー・セッションに対応します。

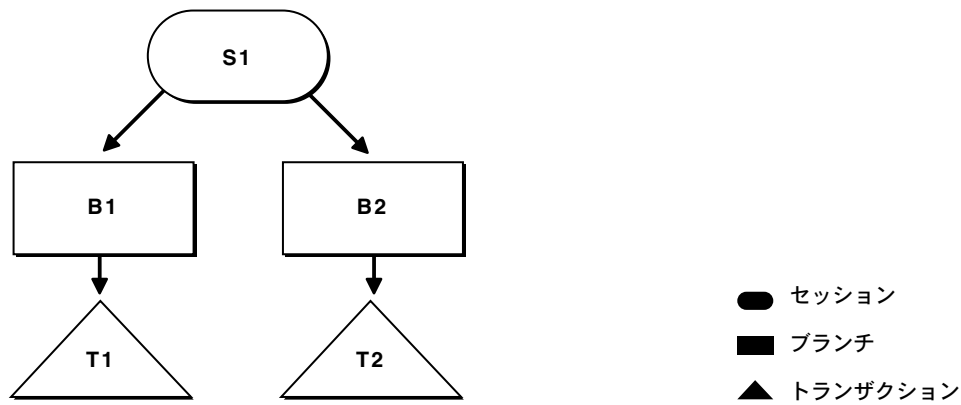
次の図は、あるアプリケーションの密結合ブランチを説明しています。この図では、S1 と S2 がセッション、B1 と B2 がブランチ、T がトランザクションです。最初の例では、2 つのブランチの *XID* が同じトランザクションで操作されているので、同じ *gtrid* を共有します。ただし、これらは別々のブランチなので、異なる *bqual* を持ちます。

図 8-1 複数の密結合ブランチ



また、単一セッションで異なる複数のブランチを操作することも可能です。この場合は、次の図に示すように、異なるグローバル・トランザクションであるため、*XID* の *gtrid* コンポーネントが異なります。

図 8-2 複数のブランチ操作のセッション



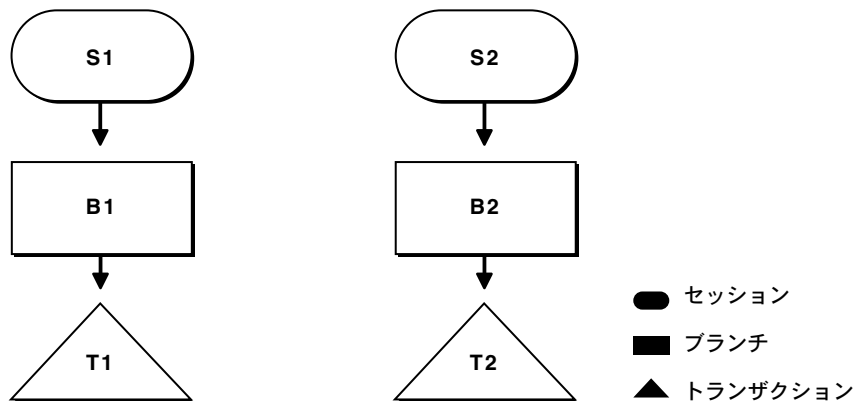
関連項目： この場合のコード例は、16-163 ページの `OCITransStart()` の例を参照してください。

同じトランザクションを共有する複数のブランチを単一セッションで操作することも可能ですが、これはあまり実用的ではありません。

関連項目： この場合のコード例は、16-163 ページの `OCITransStart()` の例を参照してください。

次の図は、疎結合ブランチの例です。

図 8-3 疎結合ブランチ



ブランチの状態 トランザクションのブランチには、アクティブ・ブランチと非アクティブ・ブランチの2通りの状態があります。

サーバー・プロセスがブランチに対する要求を実行しているとき、ブランチはアクティブ状態です。サーバー・プロセスがブランチ内の要求を実行していないとき、ブランチは非アクティブ状態です。この場合は、ブランチの親セッションは1つもなく、ブランチはサーバーの PMON プロセスに所有されます。

ブランチの連結解除と再開 OCI アプリケーションが `OCITransDetach()` コールを使用してブランチを連結解除すると、ブランチは非アクティブ状態になります。`flags` パラメータに `OCI_TRANS_RESUME` を設定した `OCITransStart()` のコールを使用してブランチを再開すると、ブランチを再びアクティブにできます。

アプリケーションで `OCITransDetach()` をコールし、ブランチを連結解除する場合は、そのブランチを作成した `OCITransStart()` コールの `timeout` パラメータで指定した値を使用します。`timeout` パラメータで指定した秒数が経過すると、PMON の子として休止しているトランザクションは削除されます。

アプリケーションでブランチを再開するには、`OCITransStart()` をコールします。このとき、トランザクション・ハンドルの属性としてブランチの `XID` を、`flags` パラメータに `OCI_TRANS_RESUME` を、また `timeout` パラメータには前回とは異なる値を指定します。このコールの `timeout` の値は、ブランチが他のプロセスで使用中のとき、そのブランチが使用可能になるまでセッションが待つ時間の長さを指定します。ブランチにアクセスしているプロセスがない場合は、すぐに再開されます。

注意： トランザクションは、そのトランザクションを連結解除したプロセス以外のプロセスからでも再開できます。ただし、トランザクションを再開できるのは、そのトランザクションを連結解除したプロセスと同じ権限を持つプロセスに限られます。

クライアント・データベース名の設定 サーバー・ハンドルには、関連する `OCI_ATTR_EXTERNAL_NAME` 属性および `OCI_ATTR_INTERNAL_NAME` 属性があります。これらの属性により、グローバル・トランザクションを実行するときに記録されるクライアント・データベース名を設定します。この名前は、障害のために準備状態で保留になっている可能性があるトランザクションを追跡するために、DBA で使用できます。

注意： OCI アプリケーションでは、グローバル・トランザクションにログオンして使用する前に、`OCIAttrSet()` を使用して、これらの属性を設定する必要があります。

1 フェーズ・コミットと 2 フェーズ・コミット グローバル・トランザクションは、1 フェーズまたは 2 フェーズでコミットされます。最も単純な状態は、単一のトランザクションが単一のデータベースに対して操作されている場合です。この場合は、アプリケーションで `OCITransCommit()` をコールすることによって、そのトランザクションの 1 フェーズ・コミットを実行できます。`OCITransCommit()` コールのデフォルト値は、1 フェーズ・コミットです。

アプリケーションから、複数のデータベースまたは複数の Oracle サーバーに対してトランザクションが処理されている場合は、さらに複雑な状態になります。この場合は、2 フェーズ・コミットが必要です。2 フェーズ・コミットは、次のステップで構成されています。

1. **準備** — アプリケーションから各トランザクションに対し、準備コール `OCITransPrepare()` を発行します。トランザクションでは、現行の作業をコミットできるか (`OCI_SUCCESS`)、できないか (`OCI_ERROR`) を示す値を戻します。
2. **コミット** — 各準備コールが `OCI_SUCCESS` の値を戻した場合は、アプリケーションから各トランザクションにコミット・コール `OCITransCommit()` を発行できます。適切な処理を行うためには、このコミット・コールの *flags* パラメータに `OCI_TRANS_TWOPHASE` を明示的に設定する必要があります。このコールのデフォルトは、1 フェーズ・コミットです。

注意： トランザクションが読取り専用であるため、コミットが不適切かつ不必要であることを示す必要がある場合、準備コールは `OCI_SUCCESS_WITH_INFO` を戻します。

追加のコール `OCITransForget()` は、偶発的に完了したトランザクションの情報をデータベースで消す必要があることを示します。このコールは、問題が起きて 2 フェーズ・コミットを異常終了する必要がある場合に使用します。サーバーで `OCITransForget()` コールを受け取ると、トランザクションに関する情報はすべて消去されます。

関連項目： 2 フェーズ・コミットの詳細は、『Oracle9i Heterogeneous Connectivity Administrator's Guide』を参照してください。

単一のメッセージによる複数のブランチの準備 同じ Oracle データベースに対し、複数のアプリケーションでグローバル・トランザクションの様々なブランチが使用される場合があります。そのようなトランザクションをコミットする前に、すべてのブランチを準備する必要があります。

大抵の場合、ブランチを使用しているアプリケーションにはそれぞれ固有のブランチを準備する役割があります。しかし、アーキテクチャによってはこの役割を外部のトランザクション・サービスに求めるものがあります。その場合、外部のトランザクション・サービスで、グローバル・トランザクションの各ブランチを準備してください。従来の `OCITransPrepare()` コールを使用すると、各ブランチを別々に準備する必要があるため、非常に手間がかかります。`OCITransMultiPrepare()` コールを使用することで、クライアントからサーバーに送信するメッセージの数を軽減できます。このコールによって、1 回

のラウンドトリップで同じグローバル・トランザクションに関連する複数のブランチを準備できます。

トランザクションの例

ここでは、トランザクション OCI コールの使用方法について説明します。

次の表では、一連の OCI コールおよびその他の処理を、その処理結果とともに示しています。簡潔な表にするため、これらのコールのパラメータをすべてリストするのではなく、コールの流れを中心に解説します。

OCI の処理列は、OCI アプリケーションの活動やコールの内容を示します。**XID** 列には、必要に応じて、トランザクション識別子がリストされます。**フラグ**列は、*flags* パラメータに渡された値を示します。**結果**列は、コールの結果を説明します。

更新成功、1 フェーズ・コミット

ステップ	OCI の処理	XID	フラグ	結果
1	OCITransStart	1234	OCI_TRANS_NEW	新規の読取り / 書込みトランザクションを開始する
2	SQL UPDATE			列を更新する
3	OCITransCommit			コミットが成功する

トランザクションの開始、連結解除、再開、準備、2 フェーズ・コミット

ステップ	OCI の処理	XID	フラグ	結果
1	OCITransStart	1234	OCI_TRANS_NEW	新規の読取り専用トランザクションを開始する
2	SQL UPDATE			列を更新する
3	OCITransDetach			トランザクションを連結解除する
4	OCITransStart	1234	OCI_TRANS_RESUME	トランザクションを再開する
5	SQL UPDATE			
6	OCITransPrepare			トランザクションで 2 フェーズ・コミットを準備する
7	OCITransCommit		OCI_TRANS_TWOPHASE	トランザクションをコミットする

注意：ステップ 4 では、同じ権限を持つ別のプロセスからトランザクションを再開することもできます。

読取り専用トランザクションの更新失敗

ステップ	OCI の処理	XID	フラグ	結果
1	OCITransStart	1234	OCI_TRANS_NEW OCI_TRANS_READONLY	新規の読取り専用トランザクションを開始する
2	SQL UPDATE			トランザクションが読取り専用なので、更新は失敗する
3	OCITransCommit			コミットによる影響なし

読取り専用トランザクションの開始、選択およびコミット

ステップ	OCI の処理	XID	フラグ	結果
1	OCITransStart	1234	OCI_TRANS_NEW OCI_TRANS_READONLY	新規の読取り専用トランザクションを開始する
2	SQL SELECT			データベースに問い合わせる
3	OCITransCommit			影響なし - トランザクションは読取り専用なので、何も変更なし

関連する初期化パラメータ

グローバル・トランザクション・ブランチおよび移行可能オープン接続の使用に関連する初期化パラメータには、次の 2 つがあります。

- TRANSACTIONS - このパラメータでは、システム全体のグローバル・トランザクション・ブランチの最大数を指定します。一方、1 つのグローバル・トランザクションのブランチ数は 8 です。
- OPEN_LINKS_PER_INSTANCE - このパラメータでは、移行可能オープン接続の最大数を指定します。移行可能オープン接続は、トランザクションをコミットした後に接続をキャッシュできるように、グローバル・トランザクションで使⽤します。この点で、OPEN_LINKS パラメータとは異なります。OPEN_LINKS パラメータで指定するのは、1 セッションからの接続数です (OPEN_LINKS パラメータは、グローバル・トランザクションには適用できません)。

パスワードおよびセッションの管理

OCI アプリケーションで、複数のユーザーの認証とメンテナンスができます。アプリケーションでユーザーのパスワードを更新できる OCI コールもあります。このコールが特に役立つのは、認証の処理でパスワード期限切れのメッセージが戻された場合です。

認証管理

`OCISessionBegin()` コールは、サービス・コンテキスト・ハンドルに設定されたサーバー・セットに対して、ユーザーを認証するときに使用します。サーバー・ハンドルに要求を行う前に、指定したサーバー・ハンドルに対して `OCISessionBegin()` をコールする必要があります。また、`OCISessionBegin()` は、`OCISessionBegin()` コールで使用するサービス・コンテキスト内で、サーバー・ハンドルによって指定された Oracle サーバーに接続するためのユーザー認証のみサポートします。つまり、`OCIServerAttach()` をコールしてサーバー・ハンドルを初期化した後、サーバー・ハンドルによって指定されたサーバーに対してユーザーを認証するために `OCISessionBegin()` をコールする必要があります。

指定のサーバー・ハンドルに対して、最初に `OCISessionBegin()` をコールしたとき、ユーザー・セッションは、移行可能モード (`OCI_MIGRATE`) では作成されません。サーバー・ハンドルに対して `OCISessionBegin()` をコールした後、アプリケーションでは `OCISessionBegin()` を再度コールし、別のユーザー・セッション・ハンドルを、別の（または同じ）資格証明と別の（または同じ）操作モードを使用して初期化できます。アプリケーションでユーザーを `OCI_MIGRATE` モードで認証する場合、サービス・ハンドルは、移行不可能なユーザー・ハンドルにすでに関連付けられている必要があります。このユーザー・ハンドルのユーザー ID は、移行可能なユーザー・セッションの所有者 ID になります。移行可能なすべてのセッションは、移行不可能な親セッションを持つ必要があります。

`OCI_MIGRATE` を指定しない場合、ユーザー・セッション・コンテキストは、`OCISessionBegin()` で使用したサーバー・ハンドルと同じサーバー・ハンドルでしか使用できません。`OCI_MIGRATE` モードを指定した場合、ユーザー認証は、異なるサーバー・ハンドルで設定されます。ただし、ユーザー・セッション・コンテキストは、同じデータベース・インスタンスを解決するサーバー・ハンドルでのみ使用できます。セキュリティ・チェックは、セッションの切替え中に行われます。

移行可能セッションを別のサーバー・ハンドルに切り替えることができるのは、セッションの所有者 ID が、現在その同じサーバーに接続されている移行不可能なセッションのユーザー ID と一致する場合のみです。

`OCI_SYSDBA`、`OCI_SYSOPER` および `OCI_PRELIM_AUTH` は、1 次ユーザー・セッション・コンテキストでのみ使用できます。

移行可能セッションは、環境ハンドルによって表された特定の環境内のサーバー・ハンドルに切替えまたは移行できます。また、移行可能セッションは、同じプロセスまたは別のモードの別のプロセス内にある、別の環境のサーバー・ハンドルに移行またはクローニングすることもできます。この移行またはクローニングを実行するには、次の手順を行う必要があります。

1. OCI_ATTR_MIGSESSION を使用してセッション・ハンドルからセッション ID を抽出します。これは、バイトの配列です。値は、コール元で変更しないでください。

関連項目： A-20 ページ「[OCI_ATTR_MIGSESSION](#)」

2. このセッション ID をなんらかの方法で他のプロセスに移送します。
3. 新しい環境内でセッション・ハンドルを作成し、OCI_ATTR_MIGSESSION を使用してセッション ID を設定します。
4. OCISessionBegin() を実行します。結果のセッション・ハンドルは、完全に認証されたセッション・ハンドルです。

OCISessionBegin() のコール用に資格証明を与えるために、2 通りの方法のうち 1 つがサポートされています。最初の方法は、OCISessionBegin() に渡されるユーザー・セッション・ハンドル内にデータベース認証用の有効なユーザー名とパスワードのペアを用意することです。この方法では、OCIAttrSet() を使用して、ユーザー・セッション・ハンドルに対して OCI_ATTR_USERNAME 属性および OCI_ATTR_PASSWORD 属性を設定します。これにより、OCI_CRED_RDBMS を指定して OCISessionBegin() がコールされます。

注意： ユーザー・セッション・ハンドルが OCISessionEnd() によって終了した場合、ユーザー名とパスワード属性は変更されるため、OCISessionBegin() への次回以降のコールでは再使用できません。次回の OCISessionBegin() コールの前に新しい値を再設定する必要があります。

もう 1 つの資格証明の方法は、外部資格証明です。この方法では、OCISessionBegin() をコールする前に、ユーザー・セッション・ハンドルについて属性を設定する必要はありません。資格証明型は OCI_CRED_EXT です。すでに OCI_ATTR_USERNAME および OCI_ATTR_PASSWORD に値が設定してある場合、OCI_CRED_EXT を使用すると、これらの値は無視されます。

パスワード管理

OCI では、OCI アプリケーションで OCIPasswordChange() コールを使用して、必要に応じてユーザーのデータベース・パスワードを変更できます。これは、OCISessionBegin() へのコールで、ユーザーのパスワードが期限切れであることを示すエラー・メッセージまたは警告が戻された場合に特に有効です。

アプリケーションでは、適切なフラグを設定すれば、OCIPasswordChange() を使用して、パスワードを変更するのと同様にユーザー認証コンテキストを設定できます。

OCIPasswordChange() が未初期化サービス・コンテキストとともにコールされる場合は、サービス・コンテキストが確立され、旧パスワードを使用してユーザーのアカウントを認証し、次に新しいパスワードに変更されます。OCI_AUTH フラグを設定している場合、ユー

ザー・セッションは初期化された状態のままになります。OCI_AUTH フラグを設定していない場合、ユーザー・セッションは消去されます。

OCIPasswordChange() に渡したサービス・コンテキストがすでに初期化されている場合、OCIPasswordChange() では、指定のアカウントが旧パスワードを使用して認証され、旧パスワードが新パスワードに変更されます。この場合、どのフラグを設定していても、ユーザー・セッションは初期化された状態のままになります。

セッション管理

ユーザー・セッションをいくつかのサーバー接続に渡って多重化することによりアクティブ・ユーザーのロード・バランシングを実行するアプリケーション（トランザクション・サーバーなど）では、これらの接続をサーバー・グループにグループ化する必要があります。Oracle はサーバー・グループを使用してこれらの接続を識別し、セッションを効率的で安全に管理できます。

サーバー・グループ名を指定するには、サーバー・コンテキストに対して OCI_ATTR_SERVER_GROUP 属性を定義する必要があります。次に例を示します。

```
OCIAttrSet ((dvoid *) srvhp, (ub4) OCI_HTYPE_SERVER, (dvoid *) group_name,
            (ub4) strlen ((CONST char *) group_name),
            (ub4) OCI_ATTR_SERVER_GROUP, errhp);
```

サーバー・グループ名は、最大 30 文字までの英数字文字列です。

OCI_ATTR_SERVER_GROUP 属性は、そのコンテキストを使用して移行不可能なセッションを作成する前に、サーバー・コンテキストで設定する必要があります。セッションの作成が成功し、サーバーへの接続が確立した後は、サーバー・グループ名は変更できません。

関連項目： A-15 ページ「OCI_ATTR_SERVER_GROUP」

あるサーバー・グループ内のサーバー上で作成されたすべての移行可能セッションは、同じサーバー・グループ内の他のサーバーにのみ移行が可能です。終了したサーバーは、サーバー・グループから削除されます。既存のサーバー・グループ内には、いつでも新しいサーバーを作成できます。

サーバー・グループの指定は省略可能です。サーバー・グループが指定されない場合、サーバーは DEFAULT というサーバー・グループ内に作成されます。

DEFAULT 以外のサーバー・グループ内の最初のサーバーに作成された最初の移行不可能なセッションの所有者には、そのサーバー・グループの所有権が確立されます。このサーバー・グループ内のすべてのサーバーについて、以降に作成されるすべての移行不可能なセッションは、そのサーバー・グループの所有者と同じユーザーによって作成される必要があります。

サーバー・グループ機能は、専用サーバーを使用する場合に便利です。この機能は、共有サーバーには効果がありません。共有サーバーの場合、すべての共有サーバーはサーバー・グループ DEFAULT に効果的に所属しています。

中間層アプリケーション

中間層アプリケーションでは、ブラウザ・クライアントからの要求が受信され、データベースに接続するかどうかが決まります。データベースでは、クライアントに戻す HTML ページが生成されます。アプリケーションには、1 つのデータベース・セッションに複数のユーザー・セッションを含めることが可能です。これら軽量セッションを使用すると、別のデータベース接続によるオーバーヘッドなしに各ユーザーが認証され、実ユーザーとしての識別情報が中間層を通して保たれます。

クライアントの認証が中間層で行われ、また、中間層の認証がデータベースで行われ、中間層がクライアントのかわりに稼働する権限を管理者から与えられているかぎり、クライアントのセキュリティに影響することなく、クライアントの識別情報をデータベース内で幅広くメンテナンスできます。

保護された 3 層アーキテクチャは、3 つのトラスト・ゾーンを囲むように設計されています。第 1 のクライアント・トラスト・ゾーンは、Web やアプリケーション・サーバーに接続している Web クライアントです。クライアントは、パスワード、暗号、トークンなどを使用して中間層で認証されます。この方法は、他のトラスト・ゾーンの確立に使用する方法とはまったく異なります。

第 2 のトラスト・ゾーンは、アプリケーション・サーバーのトラスト・リージョンです。データ・サーバーではアプリケーション・サーバーの識別情報を検証し、信頼できる場合、正しいクライアントの識別情報を渡します。第 3 のトラスト・ゾーンでは、データ・サーバーが認可サーバーと対話し、クライアントとアプリケーション・サーバーに割り当てられているロールを取得します。

アプリケーション・サーバーがサーバーに接続すると、1 次セッションが確立されます。アプリケーション・サーバーにより通常の方法でデータベースに対して認証が行われ、アプリケーション・サーバー・トラスト・ゾーンが作成されます。これで、アプリケーション・サーバーの識別情報はデータ・サーバーに対して予約済みになり、信頼されたことになります。

アプリケーションで、アプリケーション・サーバーに接続しているクライアントの識別情報を検証した場合は、第 1 のトラスト・ゾーンが作成されます。アプリケーション・サーバーでは、クライアントの要求に応じることができるよう、クライアントに対するセッション・ハンドルが必要です。中間層プロセスではセッション・ハンドルを割り当て、`OCIAttrSet()` を使用して次のようなクライアント属性を設定します。

クライアントのデータベース・ユーザー名を設定する `OCI_ATTR_USERNAME`

プロキシ要求を行う認証済みアプリケーションを示す `OCI_ATTR_PROXY_CREDENTIALS`

アプリケーション・サーバーがクライアントとして接続した後にアクティブ化するロールのリストを設定する場合は、`OCISessionBegin()` の前に `OCIAttrSet()` をコールできます。そのときに、属性 `OCI_ATTR_INITIAL_CLIENT_ROLES` およびロールのリストを含む文字列の配列を指定します。ロールの確立およびプロキシ機能の検証は、1 回のラウンドトリップで行われます。アプリケーション・サーバーがクライアントの代理として機能することを管理者が許可していない場合、あるいは特定のロールをアクティブ化することが許可されていない場合は、`OCISessionBegin()` コールが失敗します。

関連項目：『Oracle9i アプリケーション開発者ガイド - 基礎編』のセキュリティ・ポリシーの確立に関する章を参照してください。

中間層アプリケーションの属性

次の属性を使用すると、クライアントの外部名と初期権限を指定できます。アプリケーションでは、クライアントを識別または認証する代替方法として、新しいタイプの資格証明が使用されます。

OCI_CRED_PROXY

クライアントにかわってアプリケーション・サーバーでセッションを開始した場合、OCI_CRED_RDBMS（データベースのユーザー名とパスワードが必要です）や OCI_CRED_EXT（外部に資格証明が与えられます）ではなく、OCI_CRED_PROXY を資格証明のタイプとして使用します。

OCI_ATTR_PROXY_CREDENTIALS

この属性を使用して、クライアントの認証に使用するアプリケーション・サーバーの資格証明を指定します。アプリケーションのセッション・ハンドルを、次のように渡します。

```
OCIAttrSet(OCISession *session_handle,
            OCI_HTYPE_SESSION,
            OCISession *application_server_session_handle,
            (ub4) 0,
            OCI_ATTR_PROXY_CREDENTIALS,
            OCIError *error_handle);
```

OCI_ATTR_DISTINGUISHED_NAME

アプリケーションでは、データベースのユーザー名ではなく、X.509 証明書に含まれる識別名をクライアントのログイン名として使用できます。

クライアントの識別名を渡すには、次のように中間層サーバーで OCIAttrSet() をコールし、OCI_ATTR_DISTINGUISHED_NAME を渡します。

```
OCIAttrSet(OCISession *session_handle,
            OCI_HTYPE_SESSION,
            lxstp *distinguished_name,
            (ub4) 0,
            OCI_ATTR_DISTINGUISHED_NAME,
            OCIError *error_handle);
```

OCI_ATTR_CERTIFICATE

この方法は、前述の識別名による方法に似ています。ただし、この方法では、識別名ではなく X.509 証明書全体が、中間層サーバーによってデータベースに渡され、その中からデータベースにより識別名が取り出されます。

証明書全体を渡すには、次のように中間層サーバーで `OCIAttrSet()` をコールし、`OCI_ATTR_CERTIFICATE` を渡します。

```
OCIAttrSet(OCISession *session_handle,
            OCI_HTYPE_SESSION,
            ub1 *certificate,
            ub4 certificate_length,
            OCI_ATTR_CERTIFICATE,
            OCIError *error_handle);
```

証明書とともに証明タイプを渡す場合は、次のように中間層サーバーで `OCIAttrSet()` をコールし、`OCI_ATTR_CERTIFICATE_TYPE` を渡します。

```
OCIAttrSet(OCISession *session_handle,
            OCI_HTYPE_SESSION,
            ub1 *certificate_type,
            ub4 certificate_type_length,
            OCI_ATTR_CERTIFICATE_TYPE,
            OCIError *error_handle);
```

証明タイプが指定されていない場合、サーバーでは、X.509 のデフォルトの証明タイプが使用されます。

OCI_ATTR_INITIAL_CLIENT_ROLES

クライアントにかわってアプリケーション・サーバーが Oracle サーバーに接続する場合は、`OCI_ATTR_INITIAL_CLIENT_ROLES` 属性を使用して、クライアントが最初に所有する（1 つまたは複数の）ロールを指定します。一連のロールを使用可能にするために、属性、ヌル文字で終了する文字列の配列および配列内の文字列の数を指定して `OCIAttrSet()` 関数をコールします。

```
OCIAttrSet(OCISession *session_handle,
            OCI_HTYPE_SESSION,
            text ** role_array,
            ub4 number_of_strings,
            OCI_ATTR_INITIAL_CLIENT_ROLES,
            OCIError *error_handle);
```

OCI_ATTR_CLIENT_IDENTIFIER

グローバル・アプリケーション・コンテキストをサポートするために、クライアントは、セッション中にいつでもセッション・ハンドルにユーザー識別子を設定できます。これを設定するには、OCIAttrSet() のコールで OCI_ATTR_CLIENT_IDENTIFIER 属性を使用します。これにより、サーバーへの次の要求で情報が伝播され、サーバー・セッションに格納されます。識別子の最初の文字に ':' は使用できません。この文字を使用すると、予期しない動作が発生する場合があります。

使用例：

```
OCIAttrSet(sesssion,
           OCI_HTYPE_SESSION,
           (dvoid *) "appuser1",
           (ub4) strlen("appuser1"),
           OCI_ATTR_CLIENT_IDENTIFIER,
           OCIError *error_handle);
```

中間層の例

たとえば、次のようなコードが考えられます。

```
...
*OCIEnv *environment_handle;
OCIServer *data_server_handle;
OCIError *error_handle;
OCISvcCtx *application_server_service_handle;
text *client_roles[2];
OCISession *first_client_session_handle, second_client_session_handle;
...
/*
** General initialization and allocation of contexts.
*/

(void) OCIInitialize((ub4) OCI_DEFAULT,
                    (dvoid *) 0,
                    (dvoid * (*)(dvoid *, size_t)) 0,
                    (dvoid * (*)(dvoid *, dvoid *, size_t)) 0,
                    (void (*)(dvoid *, dvoid *)) 0 );
(void) OCIEnvInit( (OCIEnv **) &environment_handle, OCI_DEFAULT, (size_t) 0,
                  (dvoid **) 0 );
(void) OCIHandleAlloc( (dvoid *) environment_handle, (dvoid **) &error_handle,
                      OCI_HTYPE_ERROR, (size_t) 0, (dvoid **) 0);
/*
** Allocate and initialize the server and service contexts used by the
** application server.
*/
```

```
(void) OCIHandleAlloc( (dvoid *) environment_handle,
    (dvoid **)&data_server_handle, OCI_HTYPE_SERVER, (size_t) 0, (dvoid **) 0);
(void) OCIHandleAlloc( (dvoid *) environment_handle, (dvoid **)
    &application_server_service_handle, OCI_HTYPE_SVCCTX, (size_t) 0,
    (dvoid **) 0);
(void) OCIAttrSet((dvoid *) application_server_service_handle,
    OCI_HTYPE_SVCCTX, (dvoid *) data_server_handle, (ub4) 0, OCI_ATTR_SERVER,
    error_handle);
/*
** Authenticate the application server. In this case, external authentication is
** being used.
*/

(void) OCIHandleAlloc((dvoid *) environment_handle,
    (dvoid **)&application_server_session_handle, (ub4) OCI_HTYPE_SESSION,
    (size_t) 0, (dvoid **) 0);
checkerr(error_handle, OCISessionBegin(application_server_service_handle,
    error_handle, application_server_session_handle, OCI_CRED_EXT,
    OCI_DEFAULT));
/*
** Authenticate the first client ** Note that no password is specified by the
** application server for the client as it is trusted.
*/

(void) OCIHandleAlloc((dvoid *) environment_handle,
    (dvoid **)&first_client_session_handle, (ub4) OCI_HTYPE_SESSION,
    (size_t) 0, (dvoid **) 0);
(void) OCIAttrSet((dvoid *) first_client_session_handle,
    (ub4) OCI_HTYPE_SESSION, (dvoid *) "jeff", (ub4) strlen("jeff"),
    OCI_ATTR_USERNAME, error_handle);
/*
** In place of specifying a password, pass the session handle of the application
** server instead.
*/

(void) OCIAttrSet((dvoid *) first_client_session_handle,
    (ub4) OCI_HTYPE_SESSION, (dvoid *) application_server_session_handle,
    (ub4) 0, OCI_ATTR_PROXY_CREDENTIALS, error_handle);
(void) OCIAttrSet((dvoid *) first_client_session_handle,
    (ub4) OCI_HTYPE_SESSION, (dvoid *) "jeff@VeryBigBank.com",
    (ub4) strlen("jeff@VeryBigBank.com"), OCI_ATTR_EXTERNAL_NAME,
    error_handle);
/*
** Establish the roles that the application server can use as the client.
*/
```



```

client_roles[0] = (text *) "TELLER"; client_roles[1] = (text *) "SUPERVISOR";
(void) OCIAttrSet((dvoid *) first_client_session_handle,
    OCI_ATTR_INITIAL_CLIENT_ROLES, error_handle);
checkerr(error_handle, OCISessionBegin(application_server_service_handle,
    error_handle, first_client_session_handle, OCI_CRED_PROXY, OCI_DEFAULT));
/*
** To start a session as another client, the application server would do the
** following. It should be
** noted this code is unchanged from the current way of doing session switching.
*/

(void) OCIHandleAlloc((dvoid *) environment_handle,
    (dvoid **)&second_client_session_handle, (ub4) OCI_HTYPE_SESSION,
    (size_t) 0, (dvoid **) 0);
(void) OCIAttrSet((dvoid *) second_client_session_handle,
    (ub4) OCI_HTYPE_SESSION, (dvoid *) "mutt", (ub4) strlen("mutt"),
    OCI_ATTR_USERNAME, error_handle);
(void) OCIAttrSet((dvoid *) second_client_session_handle,
    (ub4) OCI_HTYPE_SESSION, (dvoid *) application_server_session_handle,
    (ub4) 0, OCI_ATTR_PROXY_CREDENTIALS, error_handle);
(void) OCIAttrSet((dvoid *) second_client_session_handle,
    (ub4) OCI_HTYPE_SESSION, (dvoid *) "mutt@VeryBigBank.com",
    (ub4) strlen("mutt@VeryBigBank.com"), OCI_ATTR_EXTERNAL_NAME,
    error_handle);
/*
** Note that the application server has not specified any initial roles to have
** as the second client.
*/

checkerr(error_handle, OCISessionBegin(application_server_service_handle,
    error_handle, second_client_session_handle, OCI_CRED_PROXY, OCI_DEFAULT));
/*
** To switch to the first user, the application server would apply the session
** handle obtained by the first
** OCISessionBegin() call. This is the same as is currently done.
*/

(void) OCIAttrSet((dvoid *) application_server_service_handle,
    (ub4) OCI_HTYPE_SVCCTX, (dvoid *) first_client_session_handle,
    (ub4) 0, (ub4) OCI_ATTR_SESSION, error_handle);
/*
** After doing some operations, the application server might want to switch to
** the second client. That
** would be done by the following call:
*/

```

```
(void) OCIAttrSet((dvoid *)application_server_service_handle,
    (ub4) OCI_HTYPE_SVCCTX,
    (dvoid *)second_client_session_handle, (ub4)0, (ub4)OCI_ATTR_SESSION,
    error_handle);

/*
** and then do operations as that client
*/
...
```

認証の属性

中間層では、データベース・サーバーに対して、認証自体を行うかわりにクライアントのパスワードを検証することでクライアントを認証するように要求できます。

関連項目：『Oracle9i アプリケーション開発者ガイド - 基礎編』のセキュリティ・ポリシーの確立に関する章を参照してください。

本質的にクライアント / データ・サーバー接続と同じように見えますが、同じではありません。クライアントでは、データベース操作を実行するために、クライアントのシステムに Oracle ソフトウェアをインストールする必要はありません。

アプリケーション・サーバーでは、クライアントのパスワードを渡すため、既存の OCI_ATTR_PASSWORD 属性を使用して OCIAttrSet() に次のデータを提供します。

```
OCIAttrSet(OCISession *session_handle,
    OCI_HTYPE_SESSION,
    lxstp *password,
    (ub4) 0,
    OCI_ATTR_PASSWORD,
    OCIError *error_handle);
```

外部で初期化されたコンテキスト

外部で初期化されたコンテキストとは、属性を OCI から初期化できるアプリケーション・コンテキストです。SQL 文の CREATE CONTEXT で INITIALIZED EXTERNALLY オプションを使用して、サーバーにコンテキスト・ネームスペースを作成します。

これで、OCIAttrSet() と OCISessionBegin() を使用してセッションを確立するときに、OCI インタフェースを初期化できます。セッションの確立後、CREATE CONTEXT 文で指定された PL/SQL パッケージのみを使用して、ネームスペース内部に属性を書き込むコマンドを発行できます。

デフォルト値と他のセッション属性は、OCISessionBegin() コールを使用して送信できるため、サーバー・ラウンドトリップ回数が削減されます。アプリケーション・コンテキストの定義と詳細は、次を参照してください。

関連項目：

- 『Oracle9i アプリケーション開発者ガイド - 基礎編』のセキュリティの確立に関する章を参照してください。
- 『Oracle9i SQL リファレンス』の CREATE CONTEXT 文および SYS_CONTEXT 関数を参照してください。

外部で初期化されたコンテキストの OCI 属性

クライアント・アプリケーションを開発する場合、認証を行う前に、OCI 関数に次の属性を使用して、明示的にアプリケーション・コンテキストをセッション・ハンドルに設定できます。

OCI_ATTR_APPCTX_SIZE

この属性を OCIAttrSet() コールで使用して、コンテキストの配列サイズを任意の数のコンテキスト属性で初期化します。(引数 *size* は **ub4** データ型です。)

```
OCIAttrSet(session, OCI_HTYPE_SESSION,
            (dvoid *)&size, (ub4)0, OCI_ATTR_APPCTX_SIZE, error_handle);
```

OCI_ATTR_APPCTX_LIST

この属性を OCIAttrGet() コールで使用して、セッションに対するアプリケーション・コンテキスト・リスト記述子のハンドルを入手します。(ctxl_desc パラメータのデータ型は、**OCIParm ***であることが必要です。)

```
OCIAttrGet(session, OCI_HTYPE_SESSION,
            (dvoid *)&ctxl_desc, (ub4)0, OCI_ATTR_APPCTX_LIST, error_handle);
```

次に、OCIParmGet() のコールでそのアプリケーション・コンテキスト・リスト記述子を使用し、i 番目のアプリケーション・コンテキストの個々の記述子を取得します。

```
OCIParmGet(ctxl_desc, OCI_DTYPE_PARAM, error_handle, (dvoid **)&ctx_desc, i);
```

ctx_desc のデータ型は **OCIParm ***です。

外部で初期化されたコンテキストの設定に使用するセッション・ハンドル属性

次の 3 つの属性を使用して、アプリケーション・コンテキストの適切な値を設定します。

- OCI_ATTR_APPCTX_NAME を使用してコンテキストのネームスペースを設定します。これは、有効な SQL 識別子であることが必要です。
- OCI_ATTR_APPCTX_ATTR を使用して指定されたコンテキストに属性名を設定します。属性名は、最大 30 バイトの大 / 小文字区別がない文字列です。

- OCI_ATTR_APPCTX_VALUE を使用して、指定されたコンテキストに属性の値を設定します。

各ネームスペースに複数の属性を保持でき、各属性には 1 つの値を設定します。次に、値を設定するために使用できる 3 つのコールを示します。

```
OCIAttrSet(ctx_desc, OCI_DTYPE_PARAM,
            (dvoid *)ctx_name, sizeof(ctx_name), OCI_ATTR_APPCTX_NAME, error_handle);
```

```
OCIAttrSet(ctx_desc, OCI_DTYPE_PARAM,
            (dvoid *)attr_name, sizeof(attr_name), OCI_ATTR_APPCTX_ATTR, error_handle);
```

```
OCIAttrSet(ctx_desc, OCI_DTYPE_PARAM,
            (dvoid *)value, sizeof(value), OCI_ATTR_APPCTX_VALUE, error_handle);
```

アプリケーション・コンテキスト操作は VARCHAR2 データ型に基づいているため、キャラクタ・タイプのみサポートされていることに注意してください。

関連項目： この項で説明した属性の詳細は、A-17 ページの「[ユーザー・セッション・ハンドル属性](#)」を参照してください。

外部で初期化されたコンテキストによる OCISessionBegin() の使用

OCISessionBegin() をコールすると、セッション・ハンドルに設定されたコンテキストがサーバーにプッシュされます。このコール後は、サーバー・セッションに伝播されるコンテキストはありません。

次のコード例で、このコールと属性の使用方法を示します。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <oci.h>

static text *username = (text *) "SCOTT";
static text *password = (text *) "TIGER";

static OCIEnv *envhp;
static OCIError *errhp;

int main(/* _ int argc, char *argv[] _*/);

static sword status;
```

```

int main(argc, argv)
int argc;
char *argv[];
{

    OCISession *authp = (OCISession *) 0;
    OCIServer *srvhp;
    OCISvcCtx *svchp;
    OCIDefine *defnp = (OCIDefine *) 0;
    dvoid *parmdp;
    ub4 ctxsize;
    OCIParam *ctxldesc;
    OCIParam *ctxedesc;

    (void) OCIInitialize((ub4) OCI_DEFAULT, (dvoid *) 0,
                        (dvoid * (*)(dvoid *, size_t)) 0,
                        (dvoid * (*)(dvoid *, dvoid *, size_t)) 0,
                        (void (*)(dvoid *, dvoid *)) 0 );

    (void) OCIEnvInit( (OCIEnv **) &envhp, OCI_DEFAULT, (size_t) 0,
                      (dvoid **) 0 );

    (void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, OCI_HTYPE_ERROR,
                          (size_t) 0, (dvoid **) 0 );

    /* server contexts */
    (void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, OCI_HTYPE_SERVER,
                          (size_t) 0, (dvoid **) 0 );

    (void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, OCI_HTYPE_SVCCTX,
                          (size_t) 0, (dvoid **) 0 );

    (void) OCIServerAttach( srvhp, errhp, (text *) "", strlen(""), 0 );

    /* set attribute server context in the service context */
    (void) OCIAttrSet( (dvoid *) svchp, OCI_HTYPE_SVCCTX, (dvoid *) srvhp,
                      (ub4) 0, OCI_ATTR_SERVER, (OCIError *) errhp);

    (void) OCIHandleAlloc((dvoid *) envhp, (dvoid **) &authp,
                          (ub4) OCI_HTYPE_SESSION, (size_t) 0, (dvoid **) 0);

    /*****

    /* set app ctx size to 2 because we want to set up 2 application contexts */

    ctxsize = 2;

```

```
/* set up app ctx buffer */
(void) OCIAAttrSet((dvoid *) authp, (ub4) OCI_HTYPE_SESSION,
                  (dvoid *) &ctxsize, (ub4) 0,
                  (ub4) OCI_ATTR_APPCTX_SIZE, errhp);

/* retrieve the list descriptor */
(void) OCIAAttrGet((dvoid *) authp, (ub4) OCI_HTYPE_SESSION,
                  (dvoid *) &ctxldesc, 0, OCI_ATTR_APPCTX_LIST, errhp );

/* retrieve the 1st ctx element descriptor */
(void) OCIPParamGet(ctxldesc, OCI_DTYPE_PARAM, errhp, (dvoid**)&ctxedesc, 1);

(void) OCIAAttrSet((dvoid *) ctxedesc, (ub4) OCI_DTYPE_PARAM,
                  (dvoid *) "HR", (ub4) strlen((char *) "HR"),
                  (ub4) OCI_ATTR_APPCTX_NAME, errhp);

(void) OCIAAttrSet((dvoid *) ctxedesc, (ub4) OCI_DTYPE_PARAM,
                  (dvoid *) "ATTR1", (ub4) strlen((char *) "ATTR1"),
                  (ub4) OCI_ATTR_APPCTX_ATTR, errhp);

(void) OCIAAttrSet((dvoid *) ctxedesc, (ub4) OCI_DTYPE_PARAM,
                  (dvoid *) "VALUE1", (ub4) strlen((char *) "VALUE1"),
                  (ub4) OCI_ATTR_APPCTX_VALUE, errhp);

/* set second context */
(void) OCIPParamGet(ctxldesc, OCI_DTYPE_PARAM, errhp, (dvoid**)&ctxedesc, 2);

(void) OCIAAttrSet((dvoid *) ctxedesc, (ub4) OCI_DTYPE_PARAM,
                  (dvoid *) "HR", (ub4) strlen((char *) "HR"),
                  (ub4) OCI_ATTR_APPCTX_NAME, errhp);

(void) OCIAAttrSet((dvoid *) ctxedesc, (ub4) OCI_DTYPE_PARAM,
                  (dvoid *) "ATTR2", (ub4) strlen((char *) "ATTR2"),
                  (ub4) OCI_ATTR_APPCTX_ATTR, errhp);

(void) OCIAAttrSet((dvoid *) ctxedesc, (ub4) OCI_DTYPE_PARAM,
                  (dvoid *) "VALUE2", (ub4) strlen((char *) "VALUE2"),
                  (ub4) OCI_ATTR_APPCTX_VALUE, errhp);

/*****
(void) OCIAAttrSet((dvoid *) authp, (ub4) OCI_HTYPE_SESSION,
                  (dvoid *) username, (ub4) strlen((char *) username),
                  (ub4) OCI_ATTR_USERNAME, errhp);
```

```
(void) OCIAAttrSet((dvoid *) authp, (ub4) OCI_HTYPE_SESSION,  
                  (dvoid *) password, (ub4) strlen((char *)password),  
                  (ub4) OCI_ATTR_PASSWORD, errhp);  
  
OCISessionBegin ( svchp, errhp, authp, OCI_CRED_RDBMS, (ub4) OCI_DEFAULT);  
  
}  
...
```

OCI プログラミングの高度なトピック

この章では、次の高度なプログラミング・トピックについて説明します。

- スレッド・セーフティ
- OCIThread パッケージ
- 接続プーリング
- セッション・プーリング
- 文キャッシュ
- ユーザー定義コールバック関数
- アプリケーション・フェイルオーバー・コールバック
- OCI およびアドバンスト・キューイング
- パブリッシュ・サブスクライブの通知

スレッド・セーフティ

Oracle データベース・サーバーおよび OCI ライブラリのスレッド・セーフティ機能により、開発者は OCI をマルチスレッド環境で使用できます。スレッド・セーフティ機能を使用すると、1 つのスレッドから他のスレッドへの副作用なしに、OCI のコードを、OCI コールを実行するユーザー・プログラムの複数のスレッドから再入可能にできます。

注意： スレッド・セーフティは、どのプラットフォームでも利用できるとはかぎりません。詳細は、使用しているシステム固有の Oracle マニュアルで確認してください。

次の項以降では、OCI を使用してマルチスレッド・アプリケーションを開発する方法について説明します。

OCI スレッド・セーフティの利点

Oracle Call Interface にスレッド・セーフティを実装すると、次の利点があります。

- 複数のスレッドで OCI コールを実行した場合も、単一のスレッドで連続するコールを実行した場合と同じ結果になります。
- 複数のスレッドで OCI コールを実行した場合は、スレッド間で副作用がありません。
- マルチスレッド・プログラムを作成しない場合でも、スレッド・セーフの OCI コールを使用してもパフォーマンスが低下しません。
- 複数のスレッドを使用すると、プログラムのパフォーマンスが向上します。パフォーマンスが向上するのは、別々のプロセッサでスレッドを同時実行するマルチプロセッサ・システム、および低速操作と高速操作の間でオーバーラップが発生するシングル・プロセッサ・システムの場合です。

スレッド・セーフティと 3 層アーキテクチャ

クライアント / サーバー・アプリケーションでは、クライアントをマルチスレッド・プログラムにできます。クライアント / サーバー以外のマルチスレッド・アプリケーションの典型は、3 層アーキテクチャ（クライアント / エージェント / サーバー・アーキテクチャとも呼ぶ）のアプリケーションです。このアーキテクチャでは、クライアントが関係するのは表示サービスのみです。エージェント（すなわちアプリケーション・サーバー）は、クライアント・アプリケーションのアプリケーション・ロジックを処理します。一般的には、この関係は多対 1 の関係で、複数のクライアントが同一のアプリケーション・サーバーを共有します。

この場合のサーバー層は、Oracle データベースです。このアプリケーション・サーバー（エージェント）は、それぞれのスレッドがクライアント・アプリケーションに対してサービスを提供することから、マルチスレッド・アプリケーション・サーバーとして最適です。

Oracle 環境では、このアプリケーション・サーバーは OCI プログラムまたはプリコンパイラ・プログラムです。

マルチスレッド開発の基本概念

スレッドは大きなプロセス内に存在する軽量のプロセスです。複数のスレッドで同一コードおよび同一データ・セグメントを共有しますが、プログラム・カウンタ、マシン・レジスタおよびスタックはスレッドごとにあります。グローバル変数および静的変数は、すべてのスレッドに共通です。そのため、1つのアプリケーション内で、複数のスレッドによるこれらの変数へのアクセスを管理する相互排他メカニズムが必要です。

スレッドは、一度作成されると他のスレッドとは非同期的に動作します。このため、順序に気にせずに共通のデータ要素にアクセスし、OCI コールを実行できます。このようにデータ要素に共有アクセスを行うため、複数のスレッドによってアクセスされるデータの整合性を維持するメカニズムが必要です。

データ・アクセスを管理するメカニズムでは、*mutex*（相互排他ロック）のフォームを使用します。これにより、アプリケーションで共有リソースにアクセスする複数のスレッド間で競合が発生しません。OCI では、*mutex* はそれぞれの環境ハンドルに基づくものとみなされます。

スレッド・セーフティの実装

スレッド・セーフティを利用するには、アプリケーションをスレッド・セーフなプラットフォーム上で実行する必要があります。次に、`OCIEnvCreate()` のオープン・コールの *mode* パラメータに対して `OCI_THREADED` を指定することにより、アプリケーションがマルチスレッド・モードで実行されていることを OCI レイヤーに通知する必要があります。

`OCIEnvCreate()` が `OCI_THREADED` を指定してコールされると、`OCIEnvCreate()` の後続のコールにもすべて `OCI_THREADED` を指定する必要があります。

注意： スレッド・セーフでないプラットフォームで実行するアプリケーションの場合は、`OCIInitialize()` にも、`OCIEnvCreate()` にも `OCI_THREADED` の値を渡さないでください。

シングル・スレッド・アプリケーションでは、プラットフォームがスレッド・セーフであるかどうかにかかわらず、`OCIInitialize()` または `OCIEnvCreate()` に `OCI_DEFAULT` の値を渡す必要があります。シングル・スレッド・アプリケーションを `OCI_THREADED` モードで実行すると、パフォーマンスが低下する可能性があります。

マルチスレッド・アプリケーションがスレッド・セーフなプラットフォーム上で動作している場合は、OCI ライブラリで、アプリケーションの *mutex* 化が環境ハンドル単位で管理されます。必要な場合は、アプリケーションでこの機能を上書きし、独自の *mutex* 化スキームを採用することもできます。これは、`OCI_NO_MUTEX` の値を `OCIEnvCreate()` コールに指定すると実行されます。

想定されるのは、次の3つのシナリオです。これらのシナリオは、各環境ハンドルに存在する接続の数、および各接続で作成されるスレッドの数によって変わります。

1. アプリケーションに複数の環境ハンドルがあっても、それぞれの環境ハンドルにスレッドが1つのみの場合（各環境ハンドルに1つのセッションが存在する場合）は、**mutex**化する必要はありません。
2. OCI_THREADED モードで実行しているアプリケーションに1つ以上の環境ハンドルがあり、それぞれの環境ハンドルに複数の接続がある場合は、次の方法の中から選択できます。
 - OCIEnvCreate() の *mode* に対して OCI_NO_MUTEX の値を渡します。この場合は、アプリケーション側で、同一環境ハンドルで実行する OCI コールを **mutex** する必要があります。この方法には、アプリケーションの設計に基づいて **mutex** 化スキームを最適化できるという利点があります。この方法では、1つの環境ハンドル接続で必ず一度に1つの OCI コールを処理するようにプログラミングする必要があります。
 - OCIEnvCreate() に OCI_DEFAULT の値を渡します。この場合は、OCI ライブラリが、環境ハンドルの各 OCI コールに対して自動的に **mutex** を取得します。

注意： OCI コールの処理の大部分はサーバー上で行われます。そのため、OCI コールを使用している2つのスレッドの接続先が同じ場合は、サーバーで一方のスレッドの処理が終了するまでもう一方はブロックされます。

リリース 7.x とそれより後のリリースの OCI コールの混在

アプリケーションでリリース 7.x とそれより後のリリースの OCI コールが混在し、後のリリースの適切なコールを使用してアプリケーションがスレッド・セーフとして初期化されている場合、スレッド・セーフティを達成するために opinit() をコールする必要はありません。アプリケーションでは、後続のリリース 7.x ファンクション・コールでリリース 7.x の動作を実現できます。

マルチスレッドの例

マルチスレッド・アプリケーションの例は、demo ディレクトリの cdemothr.c を参照してください。

OCIThread パッケージ

OCIThread パッケージには、Oracle のユーザーが通常使用するスレッドの基本形が多数用意されています。様々なプラットフォームに固有のスレッド機能に対して、移植可能なインタフェースも備えています。固有のスレッド機能を持たないプラットフォームでは、スレッドが実装されません。

OCIThread では、マルチスレッド機能の移植可能な実装が提供されません。固有のマルチスレッド機能に対する移植可能なカバーのセットとしてのみ機能します。したがって、マルチスレッドのための固有のサポートが用意されていないプラットフォームで対応できる実装は、OCIThread の一部に限られます。このため、OCIThread のすべての機能を使用する製品は、プラットフォームによっては移植されません。すべてのプラットフォームに移植する必要のある製品の場合は、OCIThread の機能を選択する必要があります。この問題は、このドキュメントの後の項で詳しく説明します。

OCIThread API は、主に 3 つの部分で構成されます。ここでは、各部分について簡単に説明します。それぞれの部分の詳細は、これ以降の項で説明します。

関連項目：

- 重要な追加情報については、9-11 ページの「[OCIThread パッケージの使用方法](#)」を参照してください。
- 構文、パラメータ・リスト、その他のコメントなど、OCIThread 関数の詳細は、16-126 ページの「[スレッド管理関数](#)」を参照してください。
- [初期化および終了](#)
これらのコールでは、OCIThread の初期化と終了を行います。OCIThread の初期化処理により、OCI 環境またはユーザー・セッション・ハンドルのメンバーである OCIThread コンテキストが初期化されます。このコンテキストは、他の OCIThread コールに必要です。
- [非アクティブなスレッドの基本形](#)
非アクティブなスレッドの基本形には、相互排他（mutex）ロック、スレッド ID およびスレッド固有のデータ・キーを操作する基本形が含まれます。

これらの基本形が非アクティブと呼ばれる理由は、仕様上は複数のスレッドの存在が可能ですが、複数のスレッドを必要としないためです。つまり、これらの基本形は、シングル・スレッド環境とマルチスレッド環境の両方の仕様に応じて実装できます。

このため、これらの基本形のみを使用する OCIThread クライアントでは、複数のスレッドを使用しなくても正常に機能します。つまり、コードを分岐せずにシングル・スレッド環境を稼働することができます。

■ アクティブなスレッドの基本形

アクティブなスレッドの基本形には、スレッドの作成、終了およびその他の操作を行う基本形が含まれます。

これらの基本形がアクティブと呼ばれる理由は、マルチスレッド環境以外では使用できないためです。これらの基本形の仕様では、複数のスレッドが使用可能であることが明示的に要求されます。マルチスレッド環境であるかどうかを実行時に判断する必要がある場合は、OCIThread のアクティブ基本形をコールする前に、OCIThreadIsMulti () をコールします。

初期化および終了

この項で説明する型および関数は、OCIThread パッケージの初期化および終了に関係しています。OCIThread の機能を使用する場合は、あらかじめ適切に初期化する必要があります。OCIThread のプロセス初期化関数である OCIThreadProcessInit () は、次の点に注意してコールする必要があります。

初期化関数と終了関数の外見上の動作は、OCIThread をシングル・スレッド環境で使した場合もマルチスレッド環境で使した場合も同じになります。

OCIThread コンテキスト

OCIThread 関数をコールするときは、多くの場合、OCI 環境またはユーザー・セッション・ハンドルをパラメータとして指定します。OCIThread コンテキストは、OCI 環境またはユーザー・セッション・ハンドルの一部なので、OCIThreadInit () をコールして初期化する必要があります。OCIThread コンテキストの終了は、OCIThreadTerm () をコールして行います。

注意： OCIThread コンテキストは、不透明なデータ構造になっています。コンテキストの内容を検査する必要はありません。

スレッドの初期化および終了を実装するには、次の関数を使用します。各関数の詳細は、16-126 ページの「スレッド管理関数」を参照してください。

関数	用途
OCIThreadProcessInit ()	OCIThread プロセスの初期化を実行します。
OCIThreadInit ()	OCIThread コンテキストを初期化します。

関数	用途
OCIThreadTerm()	OCIThread レイヤーを終了してコンテキストのメモリーを解放します。
OCIThreadIsMulti()	アプリケーションがマルチスレッド環境とシングル・スレッド環境のどちらで動作しているかを、コール元に通知します。

非アクティブなスレッドの基本形

非アクティブなスレッドの基本形は、`mutex`、スレッド ID、およびスレッド固有のデータを扱います。これらの基本形の仕様では、複数のスレッドが存在する必要がないため、マルチスレッドとシングル・スレッドの両方のプラットフォームで使用できます。

OCIThreadMutex

`OCIThreadMutex` 型は、相互排他ロック（`mutex`）を指定するために使用します。`mutex` は、次のいずれかの目的で使用します。

- 特定のデータ・セットに、同時に複数のスレッドからアクセスされないようにするため
- コードの特定の重要なセクションを、同時に複数のスレッドが実行しないようにするため

`mutex` ポインタは、クライアント構造またはスタンドアロン変数の一部として宣言できます。これらのポインタは、使用する前に `OCIThreadMutexInit()` により初期化する必要があります。不要になった場合は、`OCIThreadMutexDestroy()` を使用して破棄する必要があります。`mutex` ポインタは、破棄した後には使用しないでください。

`OCIThreadMutexAcquire()` を使用して、スレッドから `mutex` を取得できます。この場合、同時に複数のスレッドによって特定の `mutex` が保持されることはありません。`mutex` を保持しているスレッドは、`OCIThreadMutexRelease()` をコールすると解放できます。

OCIThreadKey

`OCIThreadKey` 型は、スレッド固有の値を持ったプロセス全体の変数とみなすことができます。つまり、プロセス内のすべてのスレッドで任意の特定のキーを使用できます。ただし、各スレッドでは、他のスレッドから独立してそのキーの検査または修正を行うことができます。スレッドがキーを検査したときに検出する値は、そのスレッドがそのキーに対して最後に設定した値と常に同じです。他のスレッドによって設定されたキーの値は検出されません。

キーによって保持される値の型は、`dvoid *` 汎用ポインタです。

キーは、`OCIThreadKeyInit()` を使用して作成できます。キーが作成されると、すべてのスレッドに対して `NULL` に初期化されます。

スレッドからは、`OCIThreadKeySet()` を使用してキーの値を設定できます。また、`OCIThreadKeyGet()` を使用してキーの値を取得できます。

OCIThread キー関数によって、スレッド固有のデータの保存および取出しを行うことができます。クライアントでスレッド・プールを管理し、スレッドごとに異なるタスクを割り当てる場合は、OCIThread キー関数を使用してタスクに関連付けられたデータを保存するのが適切でない場合があります。

失敗する場合のシナリオを次に示します。スレッドが割り当てられて、タスクの初期化が実行されます。初期化中に、タスクでは OCIThread キー関数を使用して、関連するデータをスレッドに格納します。

初期化後、そのスレッドはスレッド・プールに戻されます。次に、スレッド・プール・マネージャでは、そのタスクで一定の操作を実行するために、別のスレッドを割り当てます。また、そのタスクでは初期化時に格納したデータを取り出す必要があります。このタスクは別のスレッド内で実行しているので、そのデータを取り出せません。スレッド・プールを使用するアプリケーションでは、OCIThread キー関数を使用する場合、この問題を考慮し、注意する必要があります。

OCIThreadKeyDestFunc

OCIThreadKeyDestFunc は、キーのデストラクタ・ルーチンへのポインタの型です。キーの作成時に、キーをデストラクタ・ルーチンに関連付けることができます (**OCIThreadKeyInit()** を参照)。

キーのデストラクタ・ルーチンは、キーの値が **NULL** でないスレッドが終了するときに常にコールされます。

デストラクタ・ルーチンの戻り値はなく、パラメータを 1 つ指定します。パラメータは、スレッドの終了時にキーに対して設定された値です。

デストラクタ・ルーチンは、スレッドの終了後およびプロセスの終了前に、そのスレッドの値に対して常にコールされます。デストラクタ・ルーチンがコールされる正確なタイミングは不明です。このため、プロセス内のコードでは、デストラクタ・ルーチンの実行後の状態を想定していません。特に、デストラクタは、終了したスレッドが戻るときの結合コール前に実行されるとはかぎりません。

OCIThreadId

OCIThreadId は、スレッドの識別に使用する型です。どのような場合にも、複数のスレッドの **OCIThreadId** が同じになることはありません。ただし、**OCIThreadId** の値はリサイクルができます。つまり、スレッドが終了した後で、終了したスレッドと同じ **OCIThreadId** が付いたスレッドが作成されます。特に、スレッド ID により、スレッド T はプロセス内で一意に識別できる必要があります。T がスレッド U と常に同時に実行されるプロセスの場合は、すべてのスレッド U 内でスレッド ID が一貫しており、有効である必要があります。スレッド T のスレッド ID は、スレッド T 内で取出し可能であることが必要です。この ID の取出しは、**OCIThreadIdGet()** を使用して行います。

OCIThreadId 型は、**NULL** スレッド ID の概念をサポートしています。**NULL** スレッド ID は、実際のスレッドの ID と同じになることはありません。

非アクティブなスレッドの関数

次の関数は、mutex、スレッド・キーおよびスレッド ID の操作に使用します。

関連項目： 各関数の詳細は、16-126 ページの「[スレッド管理関数](#)」を参照してください。

関数	用途
OCIThreadMutexInit()	mutex の割当ておよび初期化を行います。
OCIThreadMutexDestroy()	mutex の破棄および割当て解除を行います。
OCIThreadMutexAcquire()	コールが行われたスレッドに対して mutex を取得します。
OCIThreadMutexRelease()	mutex を解放します。
OCIThreadKeyInit()	キーの割当ておよび初期化を行います。
OCIThreadKeyDestroy()	キーの破棄および割当て解除を行います。
OCIThreadKeyGet()	コール側スレッドのキーの現在の値を取得します。
OCIThreadKeySet()	コール側スレッドのキー値を設定します。
OCIThreadIdInit()	スレッド ID の割当ておよび初期化を行います。
OCIThreadIdDestroy()	スレッド ID の破棄および割当て解除を行います。
OCIThreadIdSet()	スレッド ID の設定を変更します。
OCIThreadIdSetNull()	スレッド ID を NULL にします。
OCIThreadIdGet()	コール側スレッドのスレッド ID を取り出します。
OCIThreadIdSame()	2 つのスレッド ID が同じスレッドを表すかどうかを調べます。
OCIThreadIdNull()	スレッド ID が NULL かどうかを調べます。

アクティブなスレッドの基本形

アクティブなスレッドの基本形では、実際のスレッドの操作を行います。これらの基本形の仕様では、多くの場合、複数のスレッドの使用が要求されます。このため、OCIThread が有効な場合にのみ正しく動作します。OCIThread が無効な場合は、常にエラーが戻されます。ただし、OCIThreadHandleGet () は例外で、シングルスレッド環境でコールできますが、この場合は効果がありません。

アクティブ基本形は、マルチスレッド環境で実行されているコード以外からはコールしないでください。OCIThreadIsMulti () をコールして、環境がマルチスレッドかシングルスレッドかどうかを調べることができます。

OCIThreadHandle

アクティブな基本形の OCIThreadJoin () および OCIThreadClose () のスレッドを操作するには、**OCIThreadHandle** 型を使用します。OCIThreadCreate () によってオープンされたスレッド・ハンドルは、対応する OCIThreadClose () コールでクローズする必要があります。OCIThreadClose () のコール後は、スレッド・ハンドルは無効になります。

OCIThread を使用するときのスレッド ID とスレッド・ハンドル間の区別は、Windows NT および Windows 2000 上のスレッド ID とスレッド・ハンドル間の区別と似ています。多くのプラットフォーム上では、基礎となるシステム固有の型は同じです。

アクティブなスレッドの関数

次の関数は、アクティブなスレッドの実装に使用します。

関連項目： 関数の詳細は、16-126 ページの「[スレッド管理関数](#)」を参照してください。

関数	用途
OCIThreadHndInit ()	スレッド・ハンドルの割当ておよび初期化を行います。
OCIThreadHndDestroy ()	スレッド・ハンドルの破棄および割当て解除を行います。
OCIThreadCreate ()	新しいスレッドを作成します。
OCIThreadJoin ()	コール側スレッドを別のスレッドと結合します。
OCIThreadClose ()	スレッド・ハンドルをクローズします。
OCIThreadHandleGet ()	スレッド・ハンドルを取り出します。

OCIThread パッケージの使用法

この項では、OCIThread の使用方法に関連する、より重要な詳細項目をいくつか説明します。

プロセスの初期化

OCIThread では、OCIThread がマルチスレッド・アプリケーション内で使用されているときにのみ、プロセス初期化関数（OCIThreadProcessInit()）のコールが必要です。シングル・スレッド・アプリケーションで OCIThreadProcessInit() をコールしなくても、エラーになりません。

OCIThread の初期化

OCIThreadInit() をコールするたびに、常に同じ OCIThread コンテキストが戻されます。

OCIThreadInit() への各コールに対応して、OCIThreadTerm() へのコールを行う必要があります。

アクティブなスレッドの基本形と非アクティブなスレッドの基本形

アクティブな基本形を使用せずに作成された OCIThread クライアント・コードは、シングル・スレッドとマルチスレッドの両方のプラットフォームで、変更せずにコンパイルして使用できます。

アクティブな基本形を使用して作成された OCIThread クライアント・コードは、マルチスレッド・プラットフォームでのみ正しく動作します。同じアプリケーションをシングル・スレッド・プラットフォームで実行するように修正するには、コードを分岐する必要があります。ソース・ファイルのアプリケーションを分岐するか、あるいは OCIThreadIsMulti() コールを使用して実行時に分岐します。

OCIThread の使用例

次のコード例は、OCIThread の使用方法を示します。

関連項目： すべてのデモ・プログラムのリストについては、[付録 B「OCI デモ・プログラム」](#) を参照してください。

```
static OCIEnv *envhp;  
static OCIError *errhp;  
void parent(argc, argv)  
sb4 argc;  
text **argv;  
{  
    OCIThreadId *tidArr[5];  
    OCIThreadHandle *tHndArr[5];  
    ub4 i;  
    OCIThreadKey *key;  
    (void) OCIInitialize((ub4) OCI_DEFAULT, (dvoid *)0,
```

```
(dvoid * (*)(dvoid *, size_t)) 0,
(dvoid * (*)(dvoid *, dvoid *, size_t))0,
(void (*)(dvoid *, dvoid *)) 0 );
(void) OCIEnvInit( (OCIEnv **) &envhp, OCI_DEFAULT, (size_t) 0,
(dvoid **) 0 );
(void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp,
OCI_HTYPE_ERROR, (size_t) 0, (dvoid **) 0);
OCIThreadProcessInit();
OCIThreadInit(envhp, errhp);
OCIThreadKeyInit(envhp, errhp, &key, (OCIThreadKeyDestFunc) NULL);
for (i=0; i<5; i++)
{
    OCIThreadIdInit(envhp, errhp, &(tidArr[i]));
    OCIThreadHndInit(envhp, errhp, &(tHndArr[i]));
}
for (i=0; i<5; i++)
    OCIThreadCreate(envhp, errhp, child, (dvoid *)key,
tidArr[i], tHndArr[i]);
for (i=0; i<5; i++)
{
    OCIThreadJoin(envhp, errhp, tHndArr[i]);
    OCIThreadClose(envhp, errhp, tHndArr[i]);
}
for (i=0; i<5; i++)
{
    OCIThreadIdDestroy(envhp, errhp, &(tidArr[i]));
    OCIThreadHndDestroy(envhp, errhp, &(tHndArr[i]));
}
OCIThreadKeyDestroy(envhp, errhp, &key);
OCIThreadTerm(envhp, errhp);
}
void child(arg)
dvoid *arg;
{
    OCIThreadKey *key = (OCIThreadKey *)arg;
    OCIThreadId *tid;
    dvoid *keyval;
    OCIThreadIdInit(envhp, errhp, &tid);
    OCIThreadIdGet(envhp, errhp, tid);
    if (OCIThreadKeySet(envhp, errhp, key, (dvoid *)tid) != OCI_SUCCESS)
        printf("Could not set value for key\n");
    if (OCIThreadKeyGet(envhp, errhp, key, &keyval) !=OCI_SUCCESS)
        printf("Could not retrieve value for key\n");
    if (keyval != (dvoid *)tid)
        printf("Incorrect value from key after setting it\n");
    /* we must destroy thread id */
    OCIThreadIdDestroy(envhp, errhp, &tid);
}
```

接続プーリング

接続プーリングとは、ロードを均衡化するために、再使用可能な物理接続のグループ（プール）を複数のセッションで使用する事です。プールの管理は、アプリケーションではなく OCI で行います。Web アプリケーション・サーバーおよび電子メール・サーバーの中間層アプリケーションでも接続プーリングを使用できます。

この機能の使用例は、バックエンドの Oracle データベースに接続している Web アプリケーション・サーバーにあります。Web アプリケーション・サーバーで、データベース・サーバーから同時に複数のデータ要求を受けたとします。Web アプリケーション・サーバーは通常、データベースとの接続を明示的に管理する必要があります。接続プーリング機能を使用すると、このタスクを OCI で実行できます。アプリケーションでは、初期化を行うとき、各環境にプール（またはプール・セット）を作成できます。

OCI 接続プーリングの概念

Oracle は、データベース・セッションと接続のファイングレイン管理など、いくつかのトランザクション・モニター機能を備えています。この機能は、接続（サーバー・ハンドル）からデータベース・セッション（ユーザー・ハンドル）の概念を分離することで実行されます。セッション切替えやセッション移行で OCI コールを使用すると、アプリケーション・サーバーまたはトランザクション・モニターでは、少数の物理接続で複数のセッションを多重化できます。したがって、接続およびバックエンドの Oracle サーバー・プロセスをプールすることで、高度な拡張性が実現します。

接続プーリングは、物理接続プールの管理をエンド・ユーザーから見えなくすることで、セッションと接続の分離をさらに簡素化します。エンド・ユーザーが行うのは、必要なデータベース・セッションの作成のみです。接続プール自体は通常、物理接続の共有プールで構成され、同じ数の専用サーバー・プロセスを含むバックエンド・サーバー・プールに変換されます。

物理接続の数は、アプリケーションで使用するデータベース・セッションの数より少なくなります。物理接続とバックエンド・サーバー・プロセスの数も、接続プーリングを使用することで少なくなります。これによって、多重化できるデータベース・セッションの数が多くなります。

共有サーバーとの類似点と相違点

中間層での接続プーリングは、共有サーバーがバックエンドで提供する機能に似ています。セッション多重化ロジックを中間層で管理することで、接続プーリングは、専用サーバー・インスタンスを共有サーバー・インスタンスと同じように動作させます。

したがって、専用サーバー・プロセスへの着信接続要求が含まれた専用サーバー・プロセスのプーリングは、中間層の接続プールによって制御されます。接続プーリングと共有サーバーの主な相違点は、共有サーバーの場合、クライアントからの接続は、通常データベース・インスタンスのディスパッチャへの接続である点です。ディスパッチャには、クライアント要求を適切な共有サーバーに送信します。一方、接続プールからの物理接続は、中間層からバックエンドのサーバー・プールにある専用サーバー・プロセスに直接接続されます。

接続プーリングは、中間層自体がマルチスレッドである場合にのみ有効です。各スレッドでは、データベースに対するセッションをメンテナンスできます。データベースへの実際の接続は接続プールでメンテナンスされ、その接続（専用データベース・サーバー・プロセスのプールを含む）は中間層のすべてのスレッドの間で共有されます。

ステートレス・セッションとステートフル・セッション

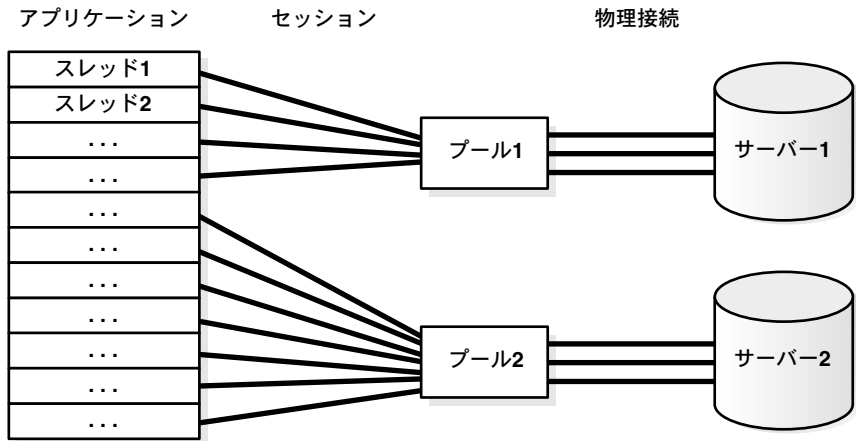
接続プーリングによって、ステートレス接続とステートフル・セッションが提供されます。ステートレス・セッションを操作する必要がある場合は、9-23 ページの「[セッション・プーリング](#)」を参照してください。

複数の接続プール

複数の接続プールの拡張概念は、異なるデータベース接続に適用できます。複数の接続プールは、異なる優先順位がユーザーに割り当てられている場合にも使用できます。接続プーリングを使用すると、異なるレベルのサービス保証を実装できます。

次の図は、これまでに説明した接続プーリングを表しています。

図 9-1 OCI 接続プーリング



接続プーリングの OCI コール

アプリケーションで接続プーリングを使用する手順は、次のとおりです。

プール・ハンドルの割当て

接続プーリングでは、`OCIHandleAlloc()` によってプール・ハンドル `OCI_HTYPE_CPOOL` を割り当てる必要があります。指定された環境ハンドルに対して複数のプールを作成できます。

単一の接続プーリングの割当て例を次に示します。

```
OCIPool *poolhp;
OCIHandleAlloc((dvoid *) envhp, (dvoid **) &poolhp, OCI_HTYPE_CPOOL,
               (size_t) 0, (dvoid **) 0));
```

接続プールの作成

`OCIConnectionPoolCreate()` 関数は、接続プール・ハンドルを初期化します。次の IN パラメータがあります。

- `connMin` – プールの作成時にオープンする最小接続数です。
- `connIncr` – すべての接続がビジーのときにコールで接続が必要な場合に、オープンする接続の増分数です。この増分は、オープンしている接続の合計数がそのプールでオープン可能な最大接続数より少ない場合にのみ使用します。
- `connMax` – プールでオープンできる最大接続数です。プールで最大数の接続がオープンするとすべての接続がビジーになるため、コールに接続が必要な場合は、取得するまで待機することになります。ただし、プールに `OCI_ATTR_CONN_NOWAIT` 属性が設定されている場合は、エラーが戻されます。
- `poolUsername` および `poolPasswd` – ユーザー・セッションがプールの接続間で透過的に移行できます。
- さらに、`OCI_ATTR_CONN_TIMEOUT` 属性によって、プールの接続のタイムアウトを設定できます。この時間を越えた接続アイドルは定期的に終了し、オープン接続を最適な数に維持します。この属性が設定されていない場合は、接続がタイムアウトになることはありません。

前述の属性は、すべて動的に構成できます。したがって、アプリケーションは、現行ロード（オープン接続数とビジーの接続数）の読み込みおよび属性の適切なチューニングを柔軟に行うことができます。

プール属性（`connMax`、`connMin`、`connIncr`）を動的に変更する場合は、`mode` パラメータに `OCI_CPOOL_REINITIALIZE` を設定した `OCIConnectionPoolCreate()` をコールする必要があります。

OUT パラメータの `poolName` と `poolNameLen` には、後続の `OCI_SERVER_ATTACH()` コールと `OCI_LOGON2()` コールで使用する値が、データベース名とデータベース名の長さの引数のかわりに格納されます。

アプリケーションで作成できるプールの数に制限はありません。中間層アプリケーションでは、この機能によって複数のプールを作成して同一サーバーまたは異なるサーバーに接続し、アプリケーション固有のニーズにあわせてロードを均衡化できます。

このコールのコード例を次に示します。

```
OCIConnectionPoolCreate((OCIEnv *)envhp,
                        (OCIError *)errhp, (OCIPool *)poolhp,
                        &poolName, &poolNameLen,
                        (text *)database, strlen(database),
                        (ub4) conMin, (ub4) conMax, (ub4) conIncr,
                        (text *)pooluser, strlen(pooluser),
                        (text *)poolpasswd, strlen(poolpasswd),
                        OCI_DEFAULT));
```

データベースへのログイン

接続プーリング・モードでデータベースにログインするために、3つのインタフェースが使用できます。それぞれのインタフェースについて、次に説明します。アプリケーションでは、次のいずれかのインタフェースを使用して、スレッドごとにデータベースにログインする必要があります。

(1) OCI_LOGON2()

最も単純なインタフェースです。このインタフェースは、接続プールを使用して単純な接続を行う場合に使用します。この場合、セッション・ハンドルの属性を変更する必要はありません。このインタフェースは、データベースへのプロキシ接続を行う場合にも使用できます。

次に、`OCI_LOGON2()` の使用例を示します。

```
for (i = 0; i < MAXTHREADS; ++i)
{
    OCI_LOGON2(envhp, errhp, &svchp[i], "scott", 5, "tiger", 5, poolName,
               poolNameLen, OCI_LOGON2_CPOOL);
}
```

このインタフェースを使用してプロキシ接続を行うには、パスワード・パラメータを `NULL` に設定してください。

(2) OCI_SESSION_GET()

このインタフェースの使用をお勧めします。このインタフェースを使用すると、ユーザーは、証明書、識別名などの外部認証方式も使用できます。`OCI_SESSION_GET()` は、セッションを取得する場合の推奨する統一ファンクション・コールです。

次に、OCI_SessionGet() の使用例を示します。

```
for (i = 0; i < MAXTHREADS; ++i)
{
    OCI_SessionGet(envhp, errhp, &svchp, authp,
                  (OraText *) poolName,
                  strlen(poolName), NULL, 0, NULL, NULL, NULL,
                  OCI_SESSGET_CPOOL)
}
```

(3) OCI_ServerAttach() および OCI_SessionBegin()

アプリケーションで、ユーザー・セッション・ハンドルおよびサーバー・ハンドルに特別な属性を設定する必要がある場合は、別のインタフェースを使用できます。この場合、アプリケーションでは次の処理が必要です。

すべてのハンドル（接続プール・ハンドル、サーバー・ハンドル、セッション・ハンドルおよびサービス・コンテキスト・ハンドル）を割り当てます。

- 接続プールを作成します。
- モードを OCI_CPOOL に設定して OCI_ServerAttach() をコールします。
- モードを OCI_DEFAULT または OCI_MIGRATE に設定して OCI_SessionBegin() をコールします。

いずれの場合も、OCI_MIGRATE フラグは内部的に設定されます。資格証明は、OCI_CRED_RDBMS または OCI_CRED_PROXY に設定できます。資格証明を OCI_CRED_PROXY に設定すると、セッション・ハンドルの設定ではユーザー名のみ必要です。（1 次セッションを明示的に作成したり、OCI_ATTR_MIGSESSION を設定する必要はありません。）

接続プーリングでの SGA 制限

OCI_CPOOL モード（接続プーリング）では、バックエンド・データベースのセッション・メモリー（UGA）は SGA から取得されます。SGA で格納できるセッション・メモリー以上のメモリー量をアプリケーションで使用する場合は、バックエンド・データベースで SGA をチューニングして SGA を拡張する必要性が生じる場合があります。バックエンド・データベースに対するメモリー・チューニングの要求は、インスタンスが専用モードである点を除けば、共有サーバーがバックエンドの場合に LARGE POOL を構成する動作に似ています。

関連項目： 詳細は、『Oracle9i データベース・パフォーマンス・チューニング・ガイドおよびリファレンス』で共有サーバーの構成の項を参照してください。

さらに SGA 制限が発生する場合は、次の点を考慮してください。

- セッションごとにオープンする文を少なくして、セッション・メモリーの使用量を減らします。
- 中間層にセッションをプールするなどの方法で、バックエンドのセッション数を減らします。
- 接続プーリングをオフにします。

アプリケーションでは、バックエンドで専用データベース・リンクを接続プーリングとともに使用しないでください。

バックエンドが専用サーバーの場合、効率的な接続プーリングは実現しません。これは、専用データベース・リンクを使用するセッションは物理接続に連結され、同じ接続が他のセッションでは使用できなくなるためです。アプリケーションで専用データベース・リンクを使用し、セッション間でバックエンド処理を効率的に共有できない場合は、共有データベース・リンクの使用を検討してください。

関連項目： 分散データベースの詳細は、『Oracle9i データベース管理者ガイド』で共有データベース・リンクの項を参照してください。

データベースからのログオフ

接続プーリング・モードでデータベースからログオフするために、ログイン・コールに対応する3つのインタフェースがあります。

(1) OCILogoff()

OCILogon2() を使用して接続を行った場合は、OCILogoff() を使用してログオフする必要があります。

(2) OCISessionRelease()

OCISessionGet() をコールして接続を行った場合は、OCISessionRelease() をコールしてログオフする必要があります。

(3) OCISessionEnd() および OCIServerDetach()

OCIServerAttach() および OCISessionBegin() をコールして接続を行いセッションを開始した場合は、OCISessionEnd() をコールしてセッションを終了し、OCIServerDetach() をコールして接続を解放する必要があります。

接続プールの破棄

[OCIConnectionPoolDestroy\(\)](#) を使用して、接続プールを破棄します。

プール・ハンドルの解放

プール・ハンドルは、OCIHandleFree() を使用して解放します。

次のコード・フラグメントは、前述した 3 つの処理を示しています。

```
for (i = 0; i < MAXTHREADS; ++i)
{
    checkerr(errhp, OCILogoff((dvoid *) svchp[i], errhp));
}
checkerr(errhp, OCIConnectionPoolDestroy(poolhp, errhp, OCI_DEFAULT));
checkerr(errhp, OCIHandleFree((dvoid *)poolhp, OCI_HTYPE_CPOOL));
```

関連項目：

- 接続プーリング属性の詳細は、A-21 ページの「[接続プール・ハンドル属性](#)」を参照してください。
- これらの関数の詳細は、[OCIConnectionPoolCreate\(\)](#)、[OCILogon2\(\)](#) および [OCIConnectionPoolDestroy\(\)](#) を参照してください。

スクロール・カーソルのパフォーマンスの向上

OCI のクライアント側プリフェッチ・バッファを使用すると、応答時間が短縮します。スクロール・カーソルに対して `OCIStmtExecute()` をコールした後、`OCI_FETCH_LAST` を使用して `OCIStmtFetch2()` をコールし、結果セットのサイズを取得できます。次に、`OCI_ATTR_PREFETCH_ROWS` をそのサイズの約 20% に設定し、結果セットで大量のメモリーを使用している場合は `OCI_ATTR_PREFETCH_MEMORY` を設定します。

接続プーリングの例

接続プーリングの例は、次を参照してください。

関連項目： demo ディレクトリの `cdemocp.c` および `cdemocpproxy.c`

接続プーリングの別のコード例を次に示します。

```
#include <oci.h>

#define MAXTHREAD 10

static OCIError *errhp;
static OCIEnv *envhp;
static OCICPool *poolhp;

static int employeeNum[MAXTHREAD];

static OraText *poolName;
static sb4 poolNameLen;
```

```

static text *database = (text *)"";
static text *username =(text *)"SCOTT";
static text *password =(text *)"TIGER";
static text *appusername =(text *)"APPUSER";
static text *apppassword =(text *)"APPPASSWD";

static ub4 conMin = 2;
static ub4 conMax = 5;
static ub4 conIncr = 1;

static void checkerr (OCIError *errhp, sword status);
static void threadFunction (dvoid *arg);

int main (void)
{
    int i = 0;

    OCIEncCreate (&envhp, OCI_THREADED, (dvoid *)0, (dvoid * (*)()) 0,
        (dvoid * (*)()) 0, (dvoid (*)()) 0, 0, (dvoid *)0);

    (void) OCIHandleAlloc((dvoid *) envhp, (dvoid **) &errhp, OCI_HTYPE_ERROR,
        (size_t) 0, (dvoid **) 0);

    (void) OCIHandleAlloc((dvoid *) envhp, (dvoid **) &poolhp, OCI_HTYPE_CPOOL,
        (size_t) 0, (dvoid **) 0);

    /* CREATE THE CONNECTION POOL */
    checkerr (errhp, OCIConnectionPoolCreate(envhp,
        errhp,poolhp, &poolName, &poolNameLen,
        database,strlen(database),
        conMin, conMax, conIncr,
        appusername,strlen(appusername),
        apppassword,strlen(apppassword),OCI_DEFAULT));

    /* Multiple threads using the connection pool */
    {
        OCIThreadId      *thrid[MAXTHREAD];
        OCIThreadHandle *thrhpp[MAXTHREAD];

        OCIThreadProcessInit ();
        checkerr (errhp, OCIThreadInit (envhp, errhp));
        for (i = 0; i < MAXTHREAD; ++i)
        {
            checkerr (errhp, OCIThreadIdInit (envhp, errhp, &thrid[i]));
            checkerr (errhp, OCIThreadHndInit (envhp, errhp, &thrhpp[i]));
        }
    }
}

```

```

for (i = 0; i < MAXTHREAD; ++i)
{
    employeeNum[i]=i;
    checkerr (errhp, OCIThreadCreate (envhp, errhp, threadFunction,
        (dvoid *) &employeeNum[i], thrid[i], thrhp[i]));
}
for (i = 0; i < MAXTHREAD; ++i)
{
    checkerr (errhp, OCIThreadJoin (envhp, errhp, thrhp[i]));
    checkerr (errhp, OCIThreadClose (envhp, errhp, thrhp[i]));
    checkerr (errhp, OCIThreadIdDestroy (envhp, errhp, &(thrid[i])));
    checkerr (errhp, OCIThreadHndDestroy (envhp, errhp, &(thrhp[i])));
}
checkerr (errhp, OCIThreadTerm (envhp, errhp));
} /* ALL THE THREADS ARE COMPLETE */

checkerr(errhp, OCIConnectionPoolDestroy(poolhp, errhp, OCI_DEFAULT));
checkerr(errhp, OCIHandleFree((dvoid *)poolhp, OCI_HTYPE_CPOOL));
checkerr(errhp, OCIHandleFree((dvoid *)errhp, OCI_HTYPE_ERROR));
} /* end of main () */

static void threadFunction (dvoid *arg)
{
    int empno = *(int *)arg;
    OCISvcCtx *svchp = (OCISvcCtx *) arg;
    text insertst1[256];
    OCISmt *stmthp = (OCISmt *)0;

    checkerr(errhp, OCILogon2(envhp, errhp, &svchp,
        (CONST OraText *)username, strlen(username),
        (CONST OraText *)password, strlen(password),
        (CONST OraText *)poolName, poolNameLen,
        OCI_CPOOL));

    sprintf(insertst1, "INSERT INTO emp(empno, ename, job, sal, deptno) values\
        (%d, 'abc', 'MANAGER', 122, 20)", empno);

    OCIHandleAlloc(envhp, (dvoid **)&stmthp, OCI_HTYPE_STMT, (size_t)0,
        (dvoid **)0);

    checkerr(errhp, OCISmtPrepare (stmthp, errhp, (text *)insertst1,
        (ub4)strlen(insertst1), OCI_NTV_SYNTAX, OCI_DEFAULT));

    checkerr(errhp, OCISmtExecute (svchp, stmthp, errhp, (ub4)1, (ub4)0,
        (OCISnapshot *)0, (OCISnapshot *)0, OCI_DEFAULT ));
}

```

```

        checkerr(errhp, OCITransCommit(svchp,errhp,(ub4)0));

        checkerr(errhp, OCIHandleFree((dvoid *) stmthp, OCI_HTYPE_STMT));
        checkerr(errhp, OCILogout((dvoid *) svchp, errhp));
    } /* end of threadFunction (dvoid *) */

void checkerr(errhp, status)
OCIError *errhp;
sword status;
{
    text errbuf[512];
    sb4 errcode = 0;

    switch (status)
    {
        case OCI_SUCCESS:
            break;
        case OCI_SUCCESS_WITH_INFO:
            (void) printf("Error - OCI_SUCCESS_WITH_INFO\n");
            break;
        case OCI_NEED_DATA:
            (void) printf("Error - OCI_NEED_DATA\n");
            break;
        case OCI_NO_DATA:
            (void) printf("Error - OCI_NODATA\n");
            break;
        case OCI_ERROR:
            (void) OCIErrorGet((dvoid *)errhp, (ub4) 1, (text *) NULL, &errcode,
                               errbuf, (ub4) sizeof(errbuf), OCI_HTYPE_ERROR);
            (void) printf("Error - %.*s\n", 512, errbuf);
            break;
        case OCI_INVALID_HANDLE:
            (void) printf("Error - OCI_INVALID_HANDLE\n");
            break;
        case OCI_STILL_EXECUTING:
            (void) printf("Error - OCI_STILL_EXECUTE\n");
            break;
        case OCI_CONTINUE:
            (void) printf("Error - OCI_CONTINUE\n");
            break;
        default:
            break;
    }
}

```

セッション・プーリング

セッション・プーリングとは、アプリケーションがデータベースに対するステートレス・セッションのグループを作成してメンテナンスすることを意味します。セッションは、要求に応じて Thin クライアントに渡されます。使用可能なセッションがない場合は、新しいセッションが作成されます。セッションでのクライアントの処理が完了すると、クライアントはプールに対してセッションを解放します。したがって、プール内のセッション数は動的に増加する場合があります。

プール内の一部のセッションに、特定のプロパティを使用してタグを付けることができます。たとえば、ユーザーは、デフォルト・セッションを要求して特定の属性を設定し、そのセッションにラベルを付けて（タグを付けて）プールに戻すことができます。そのユーザーまたは他のユーザーは、同じ属性を持つセッションを要求できるため、同じタグが付いたセッションを要求できます。プール内に、同じタグが付いた複数のセッションが存在する場合があります。セッションのタグは、変更またはリセットすることができます。

関連項目： タグの使用方法については、9-24 ページの「[セッション・プーリングでのタグの使用](#)」を参照してください。

このインタフェースを使用して、プロキシ・セッションを作成してメンテナンスすることもできます。

使用可能なセッションがなく、プールの最大サイズに達した場合のアプリケーションの動作は、指定した属性によって決まります。新しいセッションが作成されるか、エラーが戻されるか、またはスレッドがブロックしてセッションが使用可能になるまで待機する場合があります。

このタイプのプーリングの主な利点は、パフォーマンスにあります。データベースへの接続（特にデータベースがリモートの場合）は、時間がかかるアクティビティです。したがって、クライアントでは、時間をかけてサーバーに接続し、資格証明を認証して有効なセッションを受け取るかわりに、プールからセッションを取り出すことができます。

OCI セッション・プーリングの機能

セッション・プーリングの機能は、次のとおりです。

- ステートレス・セッションのプールを透過的に作成、メンテナンスおよび管理します。
- アプリケーションでプールを作成し、プール内のセッションの最小数、増分数および最大数を指定するインタフェースを提供します。
- ユーザーがデフォルト・セッションまたはタグ付けされたセッションをプールとの間で取得および解放するためのインタフェースを提供します。タグ付けされたセッションとは、クライアント定義の特定のプロパティが指定されたセッションです。
- アプリケーションで、セッションの最小数と最大数を動的に変更できます。

- 長時間アイドル状態のセッションをクローズし、必要なときにセッションを作成することによって、常に最適な数のセッションがオープンするようにメンテナンスするメカニズムを提供します。
- セッション・プーリングで認証を行うことができます。

同種セッション・プールおよび異種セッション・プール

セッション・プールは、同種または異種のいずれかになります。同種セッション・プーリングとは、プール内のセッションが同じ認証方法（同じユーザー名、パスワードおよび権限）を使用することを意味します。異種セッション・プーリングとは、セキュリティ属性と権限がセッションごとに異なる場合があるため、認証情報を指定する必要があることを意味します。

セッション・プーリングでのタグの使用

ユーザーは、タグを使用して、プール内のセッションをカスタマイズできます。クライアントは、デフォルト・セッションまたはタグが付いていないセッションをプールから取得して、特定の属性（NLS 設定など）をセッションに設定し、`OCISessionRelease()` コールでそのセッションをプールに戻すと、セッションに適切なラベル（タグ）を付けることができます。

そのユーザーまたは他のユーザーは、同じ属性が指定されたセッションを使用するために、同じタグが付いたセッションを要求できます。これを行うには、`OCISessionGet()` コールで同じタグを指定します。

関連項目： セッションのタグ付けの詳細は、15-35 ページの「[OCISessionGet\(\)](#)」を参照してください。

セッション・プーリング用のハンドル

セッション・プーリング用に、次の 2 タイプのハンドルが追加されています。

OCISPool

セッション・プール・ハンドルです。このハンドルは、`OCIHandleAlloc()` を使用して割り当てます。このハンドルは、`OCISessionPoolCreate()` および `OCISessionPoolDestroy()` に渡す必要があります。属性の型は `OCI_HTYPE_SPOOL` です。

`OCIHandleAlloc()` コールの例を次に示します。

```
OCISPool *spoolhp;  
OCIHandleAlloc((dvoid *) envhp, (dvoid **) &spoolhp, OCI_HTYPE_SPOOL,  
               (size_t) 0, (dvoid **) 0));
```

環境ハンドルに対して、複数のセッション・プールを作成できます。

OCIAuthInfo

認証情報ハンドルです。このハンドルは、`OCIHandleAlloc()` を使用して割り当てます。このハンドルは、`OCISessionGet()` に渡されます。このハンドルでは、ユーザー・セッション・ハンドルでサポートされるすべての属性をサポートします。詳細は、「ユーザー・セッション・ハンドル属性」を参照してください。認証情報ハンドルの属性の型は `OCI_HTYPE_AUTHINFO` です。

`OCIHandleAlloc()` コールの例を次に示します。

```
OCIAuthInfo *authp;
OCIHandleAlloc((dvoid *) envhp, (dvoid **) &authp, OCI_HTYPE_AUTHINFO,
               (size_t) 0, (dvoid **) 0);
```

関連項目：

- 認証情報ハンドルに属する属性については、A-17 ページの「[ユーザー・セッション・ハンドル属性](#)」を参照してください。
- セッション・プーリング属性の詳細は、A-23 ページの「[セッション・プール・ハンドル属性](#)」を参照してください。
- セッション・プーリングで使用する関数の詳細は、15-4 ページの「[接続関数、認証関数および初期化関数](#)」を参照してください。
- このコールで利用できるセッション・ハンドル属性の詳細は、15-35 ページの「[OCISessionGet\(\)](#)」を参照してください。

OCI セッション・プーリングの使用

ユーザー名とパスワードを使用する単純なセッション・プーリングのアプリケーションは、次のステップで記述します。

- **OCISPool** ハンドルに対して `OCIHandleAlloc()` を使用して、セッション・プール・ハンドルを割り当てます。環境ハンドルに対して複数のセッション・プールを作成できます。
- `mode` を `OCI_DEFAULT`（新規セッション・プールの場合）に設定した `OCISessionPoolCreate()` を使用して、セッション・プールを作成します。その他のモードについては、関数の説明を参照してください。
- スレッドごとにループを行います。次の処理を実行する関数を使用してスレッドを作成します。
- `OCIHandleAlloc()` を使用して、**OCIAuthInfo** 型の認証情報ハンドルを割り当てます。
- `OCIAttrSet()` を使用して、認証情報ハンドルにユーザー名とパスワードを設定します。
- `mode` を `OCI_SESSGET_SPOOL` に設定した `OCISessionGet()` を使用して、プールされたセッションを取得します。

- トランザクションを実行します。
- トランザクションをコミットまたはロールバックします。
- `OCISessionRelease()` を使用して、セッションを解放（ログオフ）します。
- `OCIHandleFree()` を使用して、認証情報ハンドルを解放します。
- 各スレッドのループを終了します。
- `OCISessionPoolDestroy()` を使用して、セッション・プールを破棄します。

セッション・プーリングの OCI コール

ここでは、セッション・プーリングの OCI コールの使用方法について説明します。

プール・ハンドルの割当て

セッション・プーリングでは、`OCIHandleAlloc()` をコールしてプール・ハンドル `OCI_HTYPE_SPOOL` を割り当てる必要があります。

指定された環境ハンドルに対して複数のプールを作成できます。単一のセッション・プーリングの割当て例を次に示します。

```
OCISPool *poolhp;  
OCIHandleAlloc((dvoid *) envhp, (dvoid **) &poolhp, OCI_HTYPE_SPOOL, (size_t) 0,  
               (dvoid **) 0);
```

プール・セッションの作成

`OCISessionPoolCreate()` 関数を使用して、セッション・プールを作成できます。このコールの使用例を次に示します。

```
OCISessionPoolCreate(envhp,  
                     errhp, poolhp, &poolName, &poolNameLen,  
                     database, (sb4)strlen((const signed char *)database),  
                     conMin, conMax, conIncr,  
                     appusername, (sb4)strlen((const signed char *)appusername),  
                     apppassword, (sb4)strlen((const signed char *)apppassword),  
                     OCI_DEFAULT);
```

データベースへのログイン

セッション・プーリング・モードでデータベースにログインするために、次の2つのインタフェースが使用できます。

(1) OCILogon2()

最も単純なインタフェースです。ただし、ユーザーは、セッションのタグ付けを行うことができません。

セッション・プーリング・モードで OCILogon2() を使用してデータベースにログインするコード例を、次に示します。

```
for (i = 0; i < MAXTHREADS; ++i)
{
    OCILogon2(envhp, errhp, &svchp[i], "scott", 5, "tiger", 5, poolName,
              poolNameLen, OCI_LOGON2_SPOOL);
}
```

(2) OCISessionGet()

このインタフェースの使用をお勧めします。このインタフェースを使用すると、ユーザーは、プール内のセッションにラベル付け（タグ付け）ができるため、特定のセッションを容易に取得できます。

次に、OCISessionGet() の使用例を示します。

```
OCISessionGet(envhp, errhp, &svchp, authInfop,
              (OraText *)database, strlen(database), tag,
              strlen(tag), &retTag, &retTagLen, &found,
              OCI_SESSGET_SPOOL)
```

データベースからのログオフ

セッション・プーリング・モードでデータベースからログオフするために、前述のログイン・コールに対応する2つのインタフェースがあります。

(1) OCILogoff()

OCILogon2() を使用して接続を行った場合は、OCILogoff() を使用してログオフする必要があります。

(2) OCISessionRelease()

OCISessionGet() をコールして接続を行った場合は、OCISessionRelease() をコールしてログオフする必要があります。

セッション・プールの破棄

セッション・プールを破棄するには、`OCISessionPoolDestroy()` をコールする必要があります。この関数のコール例を次に示します。

```
OCISessionPoolDestroy(poolhp, errhp, OCI_DEFAULT);
```

プール・ハンドルの解放

セッション・プール・ハンドルを解放するには、`OCIHandleFree()` をコールする必要があります。この関数のコール例を次に示します。

```
OCIHandleFree((dvoid *)poolhp, OCI_HTYPE_SPOOL);
```

OCI セッション・プーリングの例

セッション・プーリングのコード例は、次を参照してください。

関連項目： demo ディレクトリの `cdemosp.c`

文キャッシュ

文キャッシュは、リリース 2 (9.2) から導入され、セッションごとの文のキャッシュを提供および管理する機能です。文キャッシュによって、サーバーではカーソルが使用できる状態になり、文を再解析する必要はありません。文キャッシュは、接続プーリングおよびセッション・プーリングとともに使用でき、パフォーマンスおよび拡張性が向上します。また、セッション・プーリングを使用しない場合でも使用でき、TAF とともに機能します。文キャッシュを実装する OCI コールは次のとおりです。

- `OCIStmtPrepare2()`
- `OCIStmtRelease()`

セッション・プーリングを使用しない文キャッシュ

ユーザーは、OCI の通常の手順でログインします。セッションを取得するコールでは、セッションに対して文キャッシュが使用可能かどうかを指定するモードがあります。当初文キャッシュは空です。開発者は、文のテキストを使用して、キャッシュ内の文を検索します。文が存在すると、API は、すでにプリコンパイルされた SQL 文のハンドルを戻します。文が存在しない場合は、新規にプリコンパイルされた SQL 文のハンドルを戻します。

アプリケーション開発者は、文がキャッシュに戻される前に、文のバインドと定義を実行し、その後に文の実行とフェッチを実行できます。文ハンドルが見つからない場合、開発者は、追加のステップとして、ハンドルに別の属性を設定する必要があります。

`OCIStmtPrepare2()` には、キャッシュ内に文が見つからない場合に、開発者がプリコンパイルされた SQL 文を使用するのか、または NULL の文ハンドルを使用するのかを指定するモードがあります。

次にコード例を示します。

```
OCISessionBegin( userhp, ... OCI_STMT_CACHE) ;
OCIAttrset(svchp, userhp, ...); /* Set the user handle in the service context */
OCIStmtPrepare2(svchp, &stmthp, stmttext, key, ...);
OCIBindByPos(stmthp, ...);
OCIDefineByPos(stmthp, ...);
OCIStmtExecute(svchp, stmthp, ...);
OCIStmtFetch(svchp, ...);
OCIStmtRelease(stmthp, ...);
...
```

セッション・プーリングを使用する文キャッシュ

前述の文キャッシュの場合と概念は同じですが、文キャッシュはセッション・レイヤーではなくセッション・プール・レイヤーで使用可能になります。セッション・プールで文キャッシュが使用可能な場合は、そのプール内のすべてのセッションにあるすべての文がキャッシュされます。使用不可の場合は、すべてのセッションにあるすべての文がキャッシュされません。

文キャッシュの規則

次に、いくつかの注意事項を示します。

- OCISStmtPrepare() ではなく、OCIStmtPrepare2() 関数を使用します。
OCIStmtPrepare() を使用する場合は、複数のサービス・コンテキスト間で1つの文ハンドルを使用しないことをお勧めします。この関数を使用した場合、OCIStmtPrepare2() を使用して文を取得するとエラーが発生します。文ハンドルを新規のサービス・コンテキストに移行するとき、実際には古いセッションに関連するカーソルがクローズされるため、文ハンドルの共有は取得されません。OCI では、文ハンドルが移行されると、古いセッションに関連するすべてのバッファを解放するため、クライアント側での共有も取得されません。
- セッションごとに1つのサービス・コンテキストを保持し、そのサービス・コンテキストに対してのみ文ハンドルを使用することをお勧めします。デフォルトはこの使用モデルに設定され、このモデルの使用をお勧めします。
- セッションで文キャッシュが行われない場合でも OCIStmtPrepare2() コールで文ハンドルが割り当てられるため、OCIStmtPrepare2() のみを使用するアプリケーションでは、その文ハンドルに対して OCIHandleAlloc() をコールしないでください。
- OCIStmtPrepare2() のコール後、ユーザーが文ハンドルでの処理を完了した後に OCIStmtRelease() をコールする必要があります。文キャッシュを使用する場合、このコールによって文はキャッシュから解放されます。文キャッシュを使用しない場合、文は割当て解除されます。OCIHandleFree() をコールしてメモリーを解放しないでください。

- OCISTMTCACHE_SEARCH_ONLY モードを指定して OCISmtPrepare2() をコールし、NULL の文が戻された（文が見つからなかった）場合、後続の OCISmtRelease() コールは必要ないため、実行しないでください。
- OCISmtPrepare() を使用してプリコンパイルされた文に対して、OCISmtRelease() をコールしないでください。
- 文キャッシュには最大サイズ（文の数）があり、サービス・コンテキストの OCI_ATTR_STMTCACHESIZE 属性で変更できます。デフォルト値は 20 です。
- 文の解放時に、文にタグを付けることができます。これによって、次回に同じタグの文を要求できます。タグは、キャッシュを検索するために使用します。タグなしの文（タグが NULL）は、タグ付きの文の特殊なケースです。2 つの文がタグのみ異なる場合、または、1 つの文はタグ付きでもう 1 つの文はタグなしの場合、その 2 つの文は異なる文とみなされます。

関連項目： 文キャッシュに関する関数については、次を参照してください。

- 16-4 ページ「[文関数](#)」
- A-12 ページ「[OCI_ATTR_STMTCACHESIZE](#)」

文キャッシュのコード例

文キャッシュのコード例は、次を参照してください。

関連項目： demo ディレクトリの ocisc.c

ユーザー定義コールバック関数

Oracle Call Interface には、OCI コールの他にユーザー固有のコードを実行する機能があります。この機能は次の用途に使用できます。

- ユーザーがアプリケーションをチューニングできるように、トレースやパフォーマンス測定のためのコードを追加します。
- 特定の OCI コールに対して、前処理コードまたは後処理コードを実行します。
- Oracle データベース固有の OCI インタフェースを使用して、OCI 付きの他のデータ・ソースにアクセスし、OCI コールが Oracle 以外のデータ・ソースへのユーザー・コールバックを使用するようにします。

OCI コールの実行前後にユーザー・コードをコールできる機能のサポートによって、OCI コールバック機能が增強されました。OCI コードのかわりに、ユーザー定義コードを実行できる機能も用意されています。

アプリケーションのソース・コードを修正することなく、ユーザー・コールバック・コードを動的に登録することもできます。動的な登録は、OCIEnvCreate() コール時に環境ハン

ドルを初期化した後、ユーザーが作成した 5 つ以下の動的リンク・ライブラリをロードすることによって実装されます。ユーザーが作成したライブラリ（Windows NT 上の Dynamic Link Library（DLL）または Solaris オペレーティング環境上の共有ライブラリなど）では、選択した OCI コールに対するユーザー・コールバックをアプリケーションに透過的に登録します。

サンプル・アプリケーション

OCI ユーザー・コールバック機能を示したすべてのデモ・プログラムのリストについては、[付録 B「OCI デモ・プログラム」](#)を参照してください。

ユーザー・コールバックの登録

アプリケーションから、OCIUserCallbackRegister() 関数を使用してユーザー・コールバック・ライブラリを登録できます。コールバックは、環境ハンドルのコンテキスト内に登録されます。アプリケーションでは、OCIUserCallbackGet() 関数を使用してハンドルに登録されたコールバックについての情報を取り出すことができます。

関連項目： これらの関数およびパラメータの詳細は、[OCIUserCallbackGet\(\)](#) および [OCIUserCallbackRegister\(\)](#) の説明を参照してください。

ユーザー定義コールバックは、OCI コールおよび環境ハンドルに対して登録されたサブルーチンです。ユーザー定義コールバックには、最初のコールバック、置換コールバックまたは最後のコールバックを指定できます。

- 最初のコールバックの場合は、プログラムから OCI 関数をコールするときにコールされます。
- 置換コールバックは、最初のコールバック実行の後に実行されます。置換コールバックから OCI_CONTINUE の値が戻されると、後続の置換コールバックまたは通常の OCI コードが実行されます。置換コールバックから OCI_CONTINUE 以外の値が戻される場合は、後続の置換コールバックおよび OCI コードは実行されません。
- 置換コールバックから OCI_CONTINUE 以外の値が戻されるか、または OCI 関数が正常に実行されると、プログラムの制御は最後のコールバックに移ります（最後のコールバックが登録されている場合）。

置換コールバックまたは最後のコールバックから OCI_CONTINUE 以外が戻された場合は、コールバックからのリターン・コードは、対応付けられた OCI コールから戻されます。

ユーザー・コールバックでは、無効なハンドルまたは無効なコンテキストが渡された場合に、OCI_INVALID_HANDLE を戻すことがあります。

注意： 任意のコールバックから OCI_CONTINUE 以外の値が戻された場合は、リターン・コードが後続のコールバックに渡されます。置換コールバックまたは最後のコールバックから OCI_CONTINUE 以外のリターン・コードが戻された場合は、最後の（OCI_CONTINUE ではない）リターン・コードが OCI コールから戻されます。

OCIUserCallbackRegister

ユーザー・コールバックの登録には、`OCIUserCallbackRegister()` コールを使用します。

関連項目： このコールの構文については、16-186 ページの `OCIUserCallbackRegister()` を参照してください。

現段階では、`OCIUserCallbackRegister()` は環境ハンドル上でしか登録できません。`typedef OCIUserCallback` のユーザー・コールバック関数は、OCI 関数コード *fcode* によって識別される OCI コールのコンテキストとともに登録します。コールバックの型は、最初のコールバック、置換または最後のコールバックのいずれの場合も *when* パラメータによって指定します。

たとえば、最初のコールバック関数 `stmtprep_entry_dyncbk_fn` とそのコンテキスト `dynamic_context` は、次のパラメータを使用して `OCIUserCallbackRegister()` 関数をコールすることにより、`OCIStmtPrepare()` コールの環境ハンドル *hndlp* に対して登録されます。

```
OCIUserCallbackRegister( hndlp,
                        OCI_HTYPE_ENV,
                        errh,
                        stmtprep_entry_dyncbk_fn,
                        dynamic_context,
                        OCI_FNCODE_STMTPREPARE,
                        OCI_UCBTYPE_ENTRY
                        (OCIUcb*) NULL);
```


ユーザー・コールバック関数

ユーザー・コールバック関数は、次の構文に従う必要があります。

```
typedef sword (*OCIUserCallback)
(
    dvoid *ctxp,          /* context for the user callback */
    dvoid *hndlp,         /* handle for the callback, env handle for now */
    ub4 type,             /* type of handle, OCI_HTYPE_ENV for this release */
    ub4 fcode,            /* function code of the OCI call */
    ub1 when,             /* type of the callback, entry or exit */
    sword returnCode,     /* OCI return code */
    ub4 *errnop,          /* Oracle error number */
    va_list arglist);    /* parameters of the oci call */
```

コールバックは、OCIUserCallbackRegister() コールで説明したパラメータの他に、リターン・コード `errnop`、およびプロトタイプ定義で宣言された元の OCI のすべてのパラメータを使用してコールします。

最初のコールバックでは、リターン・コードは常に `OCI_SUCCESS` として渡され、`*errnop` は常に 0 として渡されます。`errnop` は IN/OUT パラメータであるため、`*errnop` は `errnop` の内容を指します。

コールバックで OCI リターン・コードを変更しない場合は、コールバックから `OCI_CONTINUE` を戻す必要があります。`*errnop` で戻された値は無視されます。一方、コールバックから `OCI_CONTINUE` 以外のリターン・コードが戻された場合は、最後に戻されたリターン・コードがそのコールのリターン・コードになります。このとき、戻された `*errnop` の値はエラー・ハンドル内に設定されます。`OCIHandleAlloc()` などの特定の OCI コールにはエラー・ハンドルがないため、エラー情報が環境ハンドルに戻された場合は、環境ハンドル内に設定されます。

置換コールバックの場合は、`returnCode` は前のコールバックまたは OCI コールから戻される `OCI_CONTINUE` 以外のリターン・コードであり、`*errnop` はエラー・ハンドル内に戻されるエラー番号の値です。これによって、後続のコールバックでは、必要に応じてリターン・コードやエラー情報を変更できます。

置換コールバックから `OCI_CONTINUE` 以外のリターン・コードが戻されると、後続の置換コールバックおよび OCI コードは無視され、最後のコールバックの処理に移るという点で、置換コールバックの処理は異なります。

置換コールバックから `OCI_CONTINUE` が戻されて OCI コードの処理が可能になった場合は、最初のコールバックからのリターン・コードが無視されることに注意してください。

OCI コールの元のパラメータは、すべて変数パラメータとしてコールバックに渡されます。コールバックでは、それらを `va_arg` を使用して取り出す必要があります。コールバックのデモ・プログラムには例が提供されています。

関連項目： 付録 B 「OCI デモ・プログラム」を参照してください。

コールバックの登録を解除するために、NULL 値を登録することができます。つまり、OCIUserCallbackRegister() コールのコールバック (OCIUserCallback) の値が NULL の場合は、ユーザー・コールバックの登録は解除されます。

スレッドセーフ・モードを使用している場合、OCI プログラムでは、ユーザー・コールバックをコールする前にすべての mutex を取得します。

UserCallback の制御フロー

次の擬似コードは、標準的な OCI コールの処理の概要を示しています。

```
OCIxyzCall()
{
    Acquire mutexes on handles;
    retCode = OCI_SUCCESS;
    errno = 0;
    for all ENTRY callbacks do
    {

        EntryretCode = (*entryCallback)(..., retcode, &errno, ...);
        if (retCode != OCI_CONTINUE)
        {
            set errno in error handle or environment handle;
            retCode = EntryretCode;
        }
    }
    for all REPLACEMENT callbacks do
    {
        retCode = (*replacementCallback) (... , retcode, &errno, ...);
        if (retCode != OCI_CONTINUE)
        {
            set errno in error handle or environment handle
            goto executeEXITCallback;
        }
    }

    retCode = return code for xyzCall; /* normal processing of OCI call */

    errno = error number from error handle or env handle;

executeExitCallback:
    for all EXIT callbacks do
    {
        exitRetCode = (*exitCallback)(..., retCode, &errno,...);
        if (exitRetCode != OCI_CONTINUE)
        {
            set errno in error handle or environment handle;
            retCode = exitRetCode;
        }
    }
}
```

```

    }
}
release mutexes;
return retCode
}

```

OCIErrorGet() に対する UserCallback

OCI コード全体がコールバックに置き換わる場合、コールバックでは、通常、コール・コンテキストに独自のエラー情報をメンテナンスし、この情報を使用して、OCIErrorGet() のコールの置換コールバックの *bufp* パラメータおよび *errnop* パラメータにエラー情報を戻します。

一方、コールバックが OCI コードの一部に置き換わるか、なんらかの後処理のみを行う場合は、最後のコールバックを使用して、エラー・テキストや OCIErrorGet() の *errnop* パラメータを、独自のエラー・メッセージやエラー番号に修正できます。最後のコールバックに渡される **errnop* は、エラー・ハンドルまたは環境ハンドル内のエラー番号です。

最初のコールバックのエラー

最初のコールバックから OCI コールのコール元にエラーを戻す場合は、置換コールバックまたは最後のコールバックを登録する必要があります。これは、OCI コードが実行される場合、最初のコールバックのエラー・コードが無視されるためです。したがって、最初のコールバックから独自のコンテキストによって、置換コールバックまたは最後のコールバックにエラーが渡される必要があります。

動的なコールバック登録

ユーザー・コールバックは、OCI 動作の監視または他のデータ・ソースへアクセスするために使用されることが多いので、コールバックの登録は、割込みが発生しないよう透過的に行われることが望まれます。これは、OCI の初期化時に、ユーザーが作成した動的リンク・ライブラリをロードすることで可能になります。これらの動的リンク・ライブラリは、パッケージと呼ばれます。ユーザーが作成したパッケージは、選択した OCI コール用のユーザー・コールバックを登録します。これらのコールバックでは、実行時に制御を受け取ったときに、必要に応じてユーザー・コールバックの追加登録または登録解除を行うことができます。

OCI デモ・プログラムとともに、パッケージを作成するための Make ファイル (Solaris オペレーティング環境の *ociucb.mk*) が用意されています。このパッケージの名前と位置は、オペレーティング・システムによって異なります。パッケージのソース・コードには、OCI の初期化時および環境作成時にコールされる特別なコールバックのコードを含める必要があります。

パッケージのロードを制御するには、オペレーティング・システムの環境変数 *ORA_OCI_UCBPKG* を設定します。この変数によって、汎用的な方法でパッケージに名前が付けられます。パッケージは、*\$ORACLE_HOME/lib* ディレクトリに格納してください。

複数のパッケージのロード

ORA_OCI_UCBPKG 変数には、パッケージ名をセミコロンで区切ったリストを含めることができます。パッケージは、リストに指定されている順序でロードされます。

たとえば、以前はパッケージを次のように指定しました。

```
setenv ORA_OCI_UCBPKG mypkg
```

現在でも前述の指定はできますが、さらに複数のパッケージを次のように指定できます。

```
setenv ORA_OCI_UCBPKG "mypkg;yourpkg;oraclepkg;sunpkg;msoftpkg"
```

これらのパッケージはすべて順番にロードされます。つまり、最初に mypkg がロードされ、最後に msoftpkg がロードされます。

最大 5 パッケージまで指定できます。

注意： サンプルの Make ファイル ociucb.mk によって、Solaris オペレーティング環境システムでは ociucb.so.1.0、Windows NT システムでは ociucb.dll が作成されます。ociucb パッケージをロードするには、環境変数 ORA_OCI_UCBPKG に ociucb を設定する必要があります。パッケージ名が .so で終わっている場合、Solaris オペレーティング環境では OCIInitialize() がエラーになります。パッケージ名は、.so.1.0 で終わる必要があります。

DLL の作成の詳細は、プラットフォームのデモ・ディレクトリにある Make ファイルをお読みください。ユーザー定義のコールバックの詳細は、プラットフォームのマニュアルでアプリケーションのコンパイルとリンクの項を参照してください。

パッケージ・フォーマット

以前は、パッケージで OCIEnvCallback() 関数のソース・コードを指定する必要がありました。現在、OCIEnvCallback() 関数は使用されていません。かわりに、パッケージのソース・コードに 2 つの関数を用意する必要があります。第 1 の関数の名前は *packagename* にして、*Init* という接尾辞を付けます。たとえば、パッケージの名前が *foo* である場合、ソース・ファイル (必ずしも *foo.c* である必要はありません) には、OCISharedLibInit() 関数のコールとともに *fooInit*() 関数を含めてください。次に例を示します。

```
sword fooInit(metaCtx, libCtx, argfmt, argc, argv)
    dvoid *      metaCtx;          /* The metacontext */
    dvoid *      libCtx;           /* The context for this package. */
    ub4          argfmt;           /* package argument format */
    sword        argc;             /* package arg count*/
    dvoid *      argv[];           /* package arguments */
{
```

```

        return (OCISharedLibInit(metaCtx, libCtx, argfmt, argc, argv,
                                fooEnvCallback));
    }

```

この場合、OCISharedLibInit() 関数の最後のパラメータである fooEnvCallback() は、第 2 の関数の名前になります。第 2 の関数の名前は通例 *packagename* にして、*EnvCallback* という接尾辞を付けます。

この関数は、OCIEnvCallback() にかわるものです。すべての動的なユーザー・コールバックは、この関数に登録する必要があります。関数は、次のように指定される **OCIEnvCallbackType** 型にしてください。

```

typedef sword (*OCIEnvCallbackType)(OCIEnv *env, ub4 mode,
                                    size_t xtramem_sz, dvoid *usrmemp,
                                    OCIUcb *ucbDesc);

```

環境ハンドルが作成されると、このコールバック関数が最後にコールされます。env パラメータは、新しく作成された環境ハンドルです。

mode、xtramem_sz および usrmemp は、OCIEnvCreate() コールに渡されるパラメータです。最後のパラメータ ucbDesc は、パッケージに渡される記述子です。後述するように、パッケージではこの記述子を使用して、ユーザー・コールバックを登録します。

サンプルの ociucb.c ファイルがデモ・ディレクトリにあります。Solaris オペレーティング環境では、パッケージ作成用の Make ファイル ociucb.mk も、デモ・ディレクトリに用意されています。これはプラットフォームによって異なりますので、注意してください。さらにデモ・ディレクトリには、ユーザー・コールバックの実例を示すデモ・プログラムがすべて (cdemouc.c、cdemoucbl.c) 揃っています。

ユーザー・コールバックの連鎖

ユーザー・コールバックは、アプリケーション自体に静的に登録することも、実行時に DLL 内に動的に登録することもできます。動的な登録の目的の動作を維持するには、以前に登録されていたコールバックを上書きした後に、上書きしたコールバックを新しく登録したコールバック内でコールするという、アプリケーションのメカニズムが必要です。この結果、ユーザー・コールバックが連鎖することがあります。

連鎖した場合のために、OCI コールに登録されている関数およびコンテキストを確認するための OCIUserCallbackGet() 関数が提供されています。

関連項目： このコールの構文については、16-184 ページの [OCIUserCallbackGet\(\)](#) を参照してください。

OCI を介した他のデータ・ソースへのアクセス

Oracle はアクセス頻度の高いデータベースであるため、アプリケーションから Oracle 以外のデータにアクセスする場合は、ユーザー・コールバックを使用して OCI インタフェース経由でアクセスします。これによって、OCI に記述されたアプリケーションでは、パフォーマンスを低下させずに Oracle データにアクセスすることができます。Oracle 以外のデータ・ソースにアクセスするには、ユーザー・コールバック内に、Oracle 以外のデータにアクセスするドライバを記述します。OCI では豊富なインタフェースを提供しているため、通常は、OCI コールからほとんどのデータ・ソースに直接マッピングできます。この方法は、ODBC などの他の中間層のアプリケーションを記述する方法より優れています。後者の場合、すべてのデータ・ソースのパフォーマンスが低下します。OCI を使用して Oracle データ・ソースに通常どおりアクセスした場合は、パフォーマンスが低下することはありませんが、Oracle 以外のデータ・ソースにアクセスした場合は、ODBC 経路と同じパフォーマンスになります。

コールバック関数の制限

OCIEnvCallback() などのコールバック関数の使用方法には、いくつかの制約があります。

- コールバックからは、OCIUserCallbackRegister()、OCIUserCallbackGet()、OCIHandleAlloc() および OCIHandleFree() 以外の OCI 関数をコールできません。これらの関数であっても、ユーザー・コールバックでコールされている場合、コールバックは再帰を避けるためにコールされません。たとえば、OCIHandleFree() が OCILogoff() のコールバックでコールされている場合は、OCILogoff() のコールバックの実行中に OCIHandleFree() のコールバックを行うことはできません。
- コールバックでは、環境ハンドルまたはエラー・ハンドルなどの OCI データ構造を変更できません。
- OCIUserCallbackRegister() コール自体または次のいずれかのコールに対して、コールバックは登録できません。
 - OCIUserCallbackGet()
 - OCIEnvCreate()
 - OCIInitialize()
 - OCIEnvInit()

OCI コールバックの例

たとえば、OCIStmtPrepare コールに対してそれぞれ最初のコールバック、置換コールバックおよび最後のコールバックを登録している 5 つのパッケージがあるとします。すなわち、ORA_OCI_UCBPKG 変数が次のように設定されています。

```
setenv ORA_OCI_UCBPKG "pkg1;pkg2;pkg3;pkg4;pkg5"
```

各パッケージ `pkgN` (`N` は 1 ～ 5) では、`pkgNInit()` 関数および `PkgNEnvCallback()` 関数を次のように指定します。

```
pkgNInit(metaCtx, libCtx, argfmt, argc, argv)
{
    return OCISharedLibInit(metaCtx, libCtx, argfmt, argc, argv, pkgNEnvCallback);
}
```

`pkgNEnvCallback()` 関数は、次のように最初のコールバック、置換コールバックおよび最後のコールバックを登録します。

```
pkgNEnvCallback(env, mode, xtramemsz, usrmemp, ucbDesc)
{
    OCIHandleAlloc((dvoid *)env, (dvoid **)&errh, OCI_HTYPE_ERROR, (size_t) 0,
        (dvoid **)NULL);

    OCIUserCallbackRegister(env, OCI_HTYPE_ENV, errh, pkgN_entry_callback_fn,
        pkgNctx, OCI_FNCODE_STMTPREPARE, OCI_UCBTYPE_ENTRY, ucbDesc);

    OCIUserCallbackRegister(env, OCI_HTYPE_ENV, errh, pkgN_replace_callback_fn,
        pkgNctx, OCI_FNCODE_STMTPREPARE, OCI_UCBTYPE_REPLACE, ucbDesc);

    OCIUserCallbackRegister(env, OCI_HTYPE_ENV, errh, pkgN_exit_callback_fn,
        pkgNctx, OCI_FNCODE_STMTPREPARE, OCI_UCBTYPE_EXIT, ucbDesc);

    return OCI_CONTINUE;
}
```

最後に、アプリケーションのソース・コードで、ユーザー・コールバックを次のように `NULL ucbDesc` で登録できます。

```
OCIUserCallbackRegister(env, OCI_HTYPE_ENV, errh, static_entry_callback_fn,
    pkgNctx, OCI_FNCODE_STMTPREPARE, OCI_UCBTYPE_ENTRY, (OCIUcb *)NULL);

OCIUserCallbackRegister(env, OCI_HTYPE_ENV, errh, static_replace_callback_fn,
    pkgNctx, OCI_FNCODE_STMTPREPARE, OCI_UCBTYPE_REPLACE, (OCIUcb *)NULL);

OCIUserCallbackRegister(env, OCI_HTYPE_ENV, errh, static_exit_callback_fn,
    pkgNctx, OCI_FNCODE_STMTPREPARE, OCI_UCBTYPE_EXIT, (OCIUcb *)NULL);
```

OCIStmtPrepare() コールが実行される場合、コールバックは次の順にコールされます。

```
static_entry_callback_fn()
pkg1_entry_callback_fn()
pkg2_entry_callback_fn()
pkg3_entry_callback_fn()
pkg4_entry_callback_fn()
pkg5_entry_callback_fn()

static_replace_callback_fn()
pkg1_replace_callback_fn()
  pkg2_replace_callback_fn()
    pkg3_replace_callback_fn()
      pkg4_replace_callback_fn()
        pkg5_replace_callback_fn()

OCI code for OCIStmtPrepare call

pkg5_exit_callback_fn()
pkg4_exit_callback_fn()
pkg3_exit_callback_fn()
pkg2_exit_callback_fn()
pkg1_exit_callback_fn()

static_exit_callback_fn()
```

注意： 最後のコールバックは、最初のコールバックおよび置換コールバックとは逆の順にコールされます。

最初のコールバックと最後のコールバックからは任意のリターン・コードが戻され、続いて次のコールバックが処理されます。ただし、置換コールバックから OCI_CONTINUE 以外のリターン・コードが戻された場合は、連鎖の次のコールバック（または、最後の置換コールバックの場合は OCI コード）が無視され、最後のコールバックの処理に移ります。たとえば、pkg3_replace_callback_fn() から OCI_SUCCESS が戻された場合、pkg4_replace_callback_fn()、pkg5_replace_callback_fn() および OCIStmtPrepare コールの OCI 処理は無視されます。かわりに、pkg5_exit_callback_fn() が次に実行されます。

外部プロシージャからの OCI コールバック

外部プロシージャからコールバックとして使用できるいくつかの OCI 関数があります。

関連項目： これらの関数については、第 19 章「OCI カートリッジ関数」のリストを参照してください。PL/SQL コードからコールできる C サブルーチンの作成、使用可能な OCI コールのリスト、およびコード例の詳細は、『Oracle9i アプリケーション開発者ガイド - 基礎編』を参照してください。

アプリケーション・フェイルオーバー・コールバック

アプリケーション・フェイルオーバー・コールバックは、あるデータベースのインスタンスで障害が発生し、別のインスタンスへフェイルオーバーする場合に使用できます。フェイルオーバー中に遅延が発生する可能性があるため、アプリケーション開発者は、フェイルオーバーが進行中であることをユーザーに知らせ、ユーザーの待機を要求した方がよい場合があります。さらに、最初のインスタンスのセッションで、いくつかの ALTER SESSION コマンドを受け取る可能性もあります。これらが、2 番目のインスタンスで自動的に再実行されることはありません。したがって、開発者は、これらの ALTER SESSION コマンドを 2 番目のインスタンスで再実行する可能性があります。

関連項目： アプリケーション・フェイルオーバーの詳細は、『Oracle9i Real Application Clusters 概要』および『Oracle9i Net Services リファレンス・ガイド』を参照してください。

フェイルオーバー・コールバックの概要

前述の問題を処理するために、アプリケーションの開発者は、フェイルオーバー・コールバック関数を登録できます。フェイルオーバーが発生した場合、コールバック関数は、ユーザー・セッションを再確立する過程で数回呼び出されます。

コールバック関数が最初にコールされるのは、Oracle がインスタンスの接続中断を最初に検出したときです。このコールバックは、アプリケーションがユーザーに対して遅延の発生を通知することを目的にしています。フェイルオーバーが正常終了すると、コールバック関数の 2 番目のコールは、接続が再確立して使用可能になったときに発生します。このとき、クライアントで ALTER SESSION コマンドが再実行され、ユーザーにフェイルオーバーが発生したことを通知する場合があります。フェイルオーバーが異常終了した場合、コールバックは、フェイルオーバーが発生しなかったことをアプリケーションに通知するためにコールされます。さらに、1 次ハンドル以外のユーザー・ハンドルが新しい接続で再認証されるたびに、コールバックがコールされます。各ユーザー・ハンドルはサーバー側セッションを表すため、クライアントでは、そのセッションのために ALTER SESSION コマンドを再実行する場合があります。

最初のフェイルオーバーが、成功しないことがあります。OCI では、失敗した後にフェイルオーバーを再試行するメカニズムを提供しています。

関連項目： この場合の詳細は、9-45 ページの「[OCI_FO_ERROR の処理](#)」を参照してください。

フェイルオーバー・コールバック構造およびパラメータ

ユーザー定義のアプリケーション・フェイルオーバー・コールバック関数の基本的な構造は次のとおりです。

```
sb4 appfocallback_fn ( dvoid      * svchp,
                      dvoid      * envhp,
                      dvoid      * fo_ctx,
                      ub4        fo_type,
                      ub4        fo_event );
```

次のパラメータについては、9-43 ページの「[フェイルオーバー・コールバックの例](#)」で例を参照してください。

svchp

最初のパラメータの *svchp* は、サービス・コンテキスト・ハンドルです。このパラメータは **dvoid *** 型です。

envhp

2 番目のパラメータの *envhp* は、OCI 環境ハンドルです。このパラメータは **dvoid *** 型です。

fo_ctx

3 番目のパラメータの *fo_ctx* は、クライアント・コンテキストです。このパラメータは、クライアントが指定するメモリーへのポインタです。この領域には、クライアントが必要な状態またはコンテキストを保管できます。このパラメータは **dvoid *** 型です。

fo_type

4 番目のパラメータの *fo_type* は、フェイルオーバーの種類です。このパラメータにより、クライアントが要求したフェイルオーバーの種類をコールバックが判断できます。通常値は次のとおりです。

- OCI_FO_SESSION — ユーザーがセッションのフェイルオーバーのみを要求したことを示します。
- OCI_FO_SELECT — ユーザーが選択フェイルオーバーも要求したことを示します。

fo_event

最後のパラメータは、フェイルオーバーのイベントです。このパラメータは、コールバックがコールされた理由を示します。次の値をとることができます。

- OCI_FO_BEGIN – フェイルオーバーが中断した接続を検出し、フェイルオーバーを開始することを示します。
- OCI_FO_END – フェイルオーバーが正常終了したことを示します。
- OCI_FO_ABORT – フェイルオーバーが失敗し、再試行できないことを示します。
- OCI_FO_ERROR – 同様にフェイルオーバーが失敗したことを示しますが、アプリケーションでエラーを処理してフェイルオーバーを再試行する機会が与えられます。

関連項目： この値の詳細は、9-45 ページの「[OCI_FO_ERROR の処理](#)」を参照してください。

- OCI_FO_REAUTH – ユーザー・ハンドルが再認証されたことを示します。どのユーザー・ハンドルかを調べるために、アプリケーションで、最初のパラメータであるサーバー・コンテキスト・ハンドルの OCI_ATTR_SESSION 属性をチェックする必要があります。

フェイルオーバー・コールバックの登録

フェイルオーバー・コールバックを使用するには、それをサーバー・コンテキスト・ハンドルに登録する必要があります。この登録は、コールバック定義構造体を作成し、サーバー・ハンドルの OCI_ATTR_FOCBK 属性にこの構造体を設定することにより行います。

コールバック定義構造体は、**OCIFocbkStruct** 型にしてください。これにはフィールドが 2 つあります。1 つは *callback_function* で、コールする関数のアドレスが含まれ、もう 1 つは *fo_ctx* で、クライアント・コンテキストのアドレスが含まれます。

コールバック登録の例は、次の項にある例の一部に含まれています。

フェイルオーバー・コールバックの例

次のコード例は、単純なユーザー定義コールバック関数の定義と登録を示しています。

パート 1、フェイルオーバー・コールバックの定義

```
sb4 callback_fn(svchp, envhp, fo_ctx, fo_type, fo_event )
dvoid * svchp;
dvoid * envhp;
dvoid *fo_ctx;
ub4 fo_type;
ub4 fo_event;
{
switch (fo_event)
{
case OCI_FO_BEGIN:
{
printf(" Failing Over ... Please stand by \n");
printf(" Failover type was found to be %s \n",
      ((fo_type==OCI_FO_SESSION) ? "SESSION"
      : (fo_type==OCI_FO_SELECT) ? "SELECT"
      : "UNKNOWN!"));
printf(" Failover Context is :%s\n",
      (fo_ctx?(char *)fo_ctx:"NULL POINTER!"));
break;
}
case OCI_FO_ABORT:
{
printf(" Failover aborted. Failover will not take place.\n");
break;
}
case OCI_FO_END:
{
printf(" Failover ended ...resuming services\n");
break;
}
case OCI_FO_REAUTH:
{
printf(" Failed over user. Resuming services\n");
break;
}
default:
{
printf("Bad Failover Event: %d.\n", fo_event);
break;
}
}
return 0;
}
```

パート2、フェイルオーバー・コールバックの登録

```
int register_callback(svrh, errh)
dvoid *svrh; /* the server handle */
OCLError *errh; /* the error handle */
{
    OCIFocbkStruct failover; /* failover callback structure */
    /* allocate memory for context */
    if (!(failover.fo_ctx = (dvoid *)malloc(strlen("my context."))))
        return(1);
    /* initialize the context. */
    strcpy((char *)failover.context_function, "my context.");
    failover.callback_function = &callback_fn;
    /* do the registration */
    if (OCIAttrSet(svrh, (ub4) OCI_HTYPE_SERVER,
                  (dvoid *) &failover, (ub4) 0,
                  (ub4) OCI_ATTR_FOCBK, errh) != OCI_SUCCESS)
        return(2);
    /* successful conclusion */
    return (0);
}
```

OCI_FO_ERROR の処理

フェイルオーバーは、成功しないことがあります。失敗した場合は、コールバック関数には、`fo_event` パラメータに `OCI_FO_ABORT` または `OCI_FO_ERROR` の値が戻されます。`OCI_FO_ABORT` の場合は、フェイルオーバーが失敗し、フェイルオーバーを続行できないことを示します。`OCI_FO_ERROR` の場合は、コールバック関数にエラーの対策を指示します。たとえば、コールバックから、一定時間待機してから、OCI ライブラリにフェイルオーバーの再試行を指示することができます。

注意： この機能が利用できるのは、Oracle サーバーに対して実行されている、リリース 8.0.5 以上の OCI ライブラリにリンクしているアプリケーションのみです。

次の時系列でイベントが発生すると仮定します。

時間	イベント
T0	データベースに障害が発生します（障害は T5 まで続きます）。
T1	ユーザーの操作によってフェイルオーバーが動作します。
T2	ユーザーが再接続を試行します。試行が失敗します。

時間	イベント
T3	OCI_FO_ERROR によってフェイルオーバー・コールバックがコールされます。
T4	フェイルオーバー・コールバックは事前に定義されたスリープ状態になります。
T5	データベースがリストアされます。
T6	フェイルオーバー・コールバックによって新しいフェイルオーバーが動作し、成功します。
T7	ユーザーは再接続に成功します。

コールバック関数から OCI_FO_RETRY の値が戻されると、新しいフェイルオーバーが動作します。

次のコードは、前述のシナリオと類似したフェイルオーバー手法の実装に使用するコールバック関数の例です。この場合、フェイルオーバー・コールバックはループ状態になります。その間いったんスリープ状態になってから、再試行が成功するまでフェイルオーバーを再試行します。

```
/*-----*/
/* the user defined failover callback */
/*-----*/
sb4 callback_fn(svchp, envhp, fo_ctx, fo_type, fo_event )
dvoid * svchp;
dvoid * envhp;
dvoid *fo_ctx;
ub4 fo_type;
ub4 fo_event;
{
    OCIError *errhp;
    OCIHandleAlloc(envhp, (dvoid **)&errhp, (ub4) OCI_HTYPE_ERROR,
        (size_t) 0, (dvoid **) 0);
    switch (fo_event)
    {
    case OCI_FO_BEGIN:
    {
        printf(" Failing Over ... Please stand by \n");
        printf(" Failover type was found to be %s \n",
            ((fo_type==OCI_FO_NONE) ? "NONE"
             : (fo_type==OCI_FO_SESSION) ? "SESSION"
             : (fo_type==OCI_FO_SELECT) ? "SELECT"
             : (fo_type==OCI_FO_TXNAL) ? "TRANSACTION"
             : "UNKNOWN!"));
        printf(" Failover Context is :%s\n",
            (fo_ctx?(char *)fo_ctx:"NULL POINTER!"));
```

```

        break;
    }
    case OCI_FO_ABORT:
    {
        printf(" Failover aborted. Failover will not take place.\n");
        break;
    }
    case OCI_FO_END:
    {
        printf("\n Failover ended ...resuming services\n");
        break;
    }
    case OCI_FO_REAUTH:
    {
        printf(" Failed over user. Resuming services\n");
        break;
    }
    case OCI_FO_ERROR:
    {
        /* all invocations of this can only generate one line. The newline
         * will be put at fo_end time.
         */
        printf(" Failover error gotten. Sleeping...");
        sleep(3);
        printf("Retrying. ");
        return (OCI_FO_RETRY);
        break;
    }
    default:
    {
        printf("Bad Failover Event: %d.\n", fo_event);
        break;
    }
}
return 0;
}

```

次のコードは、このフェイルオーバー・コールバック関数を含んだプログラムの例です。

```

executing select...
7369 SMITH CLERK
7499 ALLEN SALESMAN
Failing Over ... Please stand by
Failover type was found to be SELECT
Failover Context is :My context.
Failover error gotten. Sleeping...Retrying. Failover error gotten.
Sleeping...Retrying. Failover error gotten. Sleeping...Retrying. Failover error

```

```
gotten. Sleeping...Retrying. Failover error gotten. Sleeping...Retrying. Failover
error gotten. Sleeping...Retrying. Failover error gotten. Sleeping...Retrying.
Failover error gotten. Sleeping...Retrying. Failover error gotten.
Sleeping...Retrying. Failover error gotten. Sleeping...Retrying.
Failover ended ...resuming services
7521    WARD    SALESMAN
7566    JONES    MANAGER
7654    MARTIN   SALESMAN
7698    BLAKE    MANAGER
7782    CLARK    MANAGER
7788    SCOTT    ANALYST
7839    KING     PRESIDENT
7844    TURNER   SALESMAN
7876    ADAMS    CLERK
7900    JAMES    CLERK
7902    FORD     ANALYST
```

OCI およびアドバンスト・キューイング

OCI には、Oracle アドバンスト・キューイング (AQ) 機能に対応したインタフェースが用意されています。Oracle AQ は、Oracle サーバーと統合されたメッセージ・キューイングです。Oracle AQ のメッセージ・キューイングでは、キューイング・システムをデータベースと統合し、メッセージ対応データベースを作成する機能を提供しています。Oracle AQ では統合された解決策が提供されるため、アプリケーション開発者がメッセージ・インフラストラクチャを構築する必要がなくなり、特定のビジネス・ロジックに専念できるようになりました。

注意： アドバンスト・キューイングを使用するには、Enterprise Edition が必要です。

関連項目： AQ の概念、機能および例などの詳細は、次を参照してください。

- 『Oracle9i アプリケーション開発者ガイド - アドバンスト・キューイング』
- 『Oracle9i XML Developer's Kit ガイド - XDK』
- 『Oracle8i Integration Server 概要』
- AQ を OCI とともに使用する方法を示すコード例は、16-88 ページの [OCIAQEnq\(\)](#) の説明を参照してください。

OCI アドバンスト・キューイング関数

OCI ライブラリには、アドバンスト・キューイングに関して次の関数が含まれています。

- OCIAQEnq()
- OCIAQDeq()
- OCIAQListen()

関連項目： 16-85 ページの「[アドバンスト・キューイング関数およびパブリッシュ・サブスクライブ関数](#)」に、これらの関数およびパラメータについての完全な説明が記載されています。

OCI アドバンスト・キューイングの説明

次の記述子が OCI AQ 操作に使用されます。

- OCIAQEnqOptions
- OCIAQDeqOptions
- OCIAQMsgProperties
- OCIAQAgent

標準 OCIDescriptorAlloc() コールを使用して、サービス・ハンドルに関するこれらの記述子を割り当てられます。次のコードでこの例を示します。

```
OCIDescriptorAlloc(svch, &enqueue_options, OCI_DTYPE_AQENQ_OPTIONS, 0, 0 );
OCIDescriptorAlloc(svch, &dequeue_options, OCI_DTYPE_AQDEQ_OPTIONS, 0, 0 );
OCIDescriptorAlloc(svch, &message_properties, OCI_DTYPE_AQMSG_PROPERTIES, 0, 0 );
OCIDescriptorAlloc(svch, &agent, OCI_DTYPE_AQAGENT, 0, 0 );
```

各記述子には、設定または読み込み（あるいはその両方）ができる、様々な属性が含まれます。

関連項目： これらの属性の詳細は、A-43 ページの「[アドバンスト・キューイング記述子の属性](#)」を参照してください。

OCI のアドバンスト・キューイングと PL/SQL

次の表は、OCI AQ 関数および記述子の関数、パラメータ、オプションと、dbms_aq パッケージの PL/SQL AQ 関数との比較を表します。

PL/SQL ファンクション	OCI 関数
DBMS_AQ.ENQUEUE	OCIAQEnq()
DBMS_AQ.DEQUEUE	OCIAQDeq()

PL/SQL ファンクション	OCI 関数
DBMS_AQ.LISTEN	OCIAQListen()

DBMS_AQ.ENQUEUE パラメータ	OCIAQEnq() パラメータ
queue_name	queue_name
enqueue_options	enqueue_options
message_properties	message_properties
payload	payload
msgid	msgid
注意: OCIAQEnq() では、さらに <i>svch</i> 、 <i>errh</i> 、 <i>payload_tdo</i> 、 <i>payload_ind</i> および <i>flags</i> の各パラメータが必要です。	

DBMS_AQ.DEQUEUE パラメータ	OCIAQDeq() パラメータ
queue_name	queue_name
dequeue_options	dequeue_options
message_properties	message_properties
payload	payload
msgid	msgid
注意: OCIAQDeq() ではさらに、 <i>svch</i> 、 <i>errh</i> 、 <i>queue_name</i> 、 <i>dequeue_options</i> 、 <i>message_properties</i> 、 <i>payload_tdo</i> 、 <i>payload</i> 、 <i>payload_ind</i> および <i>flags</i> の各パラメータが必要です。	

DBMS_AQ.LISTEN パラメータ	OCIAQListen() パラメータ
agent_list	agent_list
wait	wait
agent	agent
注意: OCIAQListen() ではさらに、 <i>svchp</i> 、 <i>errhp</i> 、 <i>agent_list</i> 、 <i>num_agents</i> 、 <i>wait</i> 、 <i>agent</i> および <i>flags</i> の各パラメータが必要です。	

PL/SQL Agent パラメータ	OCIAQAgent 属性
name	OCI_ATTR_AGENT_NAME
address	OCI_ATTR_AGENT_ADDRESS
protocol	OCI_ATTR_AGENT_PROTOCOL

PL/SQL メッセージ・プロパティ	OCIAQMsgProperties 属性
priority	OCI_ATTR_PRIORITY
delay	OCI_ATTR_DELAY
expiration	OCI_ATTR_EXPIRATION
correlation	OCI_ATTR_CORRELATION
attempts	OCI_ATTR_ATTEMPTS
recipient_list	OCI_ATTR_RECIPIENT_LIST
exception_queue	OCI_ATTR_EXCEPTION_QUEUE
enqueue_time	OCI_ATTR_ENQ_TIME
state	OCI_ATTR_MSG_STATE
sender_id	OCI_ATTR_SENDER_ID
original_msgid	OCI_ATTR_ORIGINAL_MSGID

PL/SQL エンキュー・オプション	OCIAQEnqOptions 属性
visibility	OCI_ATTR_VISIBILITY
relative_msgid	OCI_ATTR_RELATIVE_MSGID
sequence_deviation	OCI_ATTR_SEQUENCE_DEVIATION

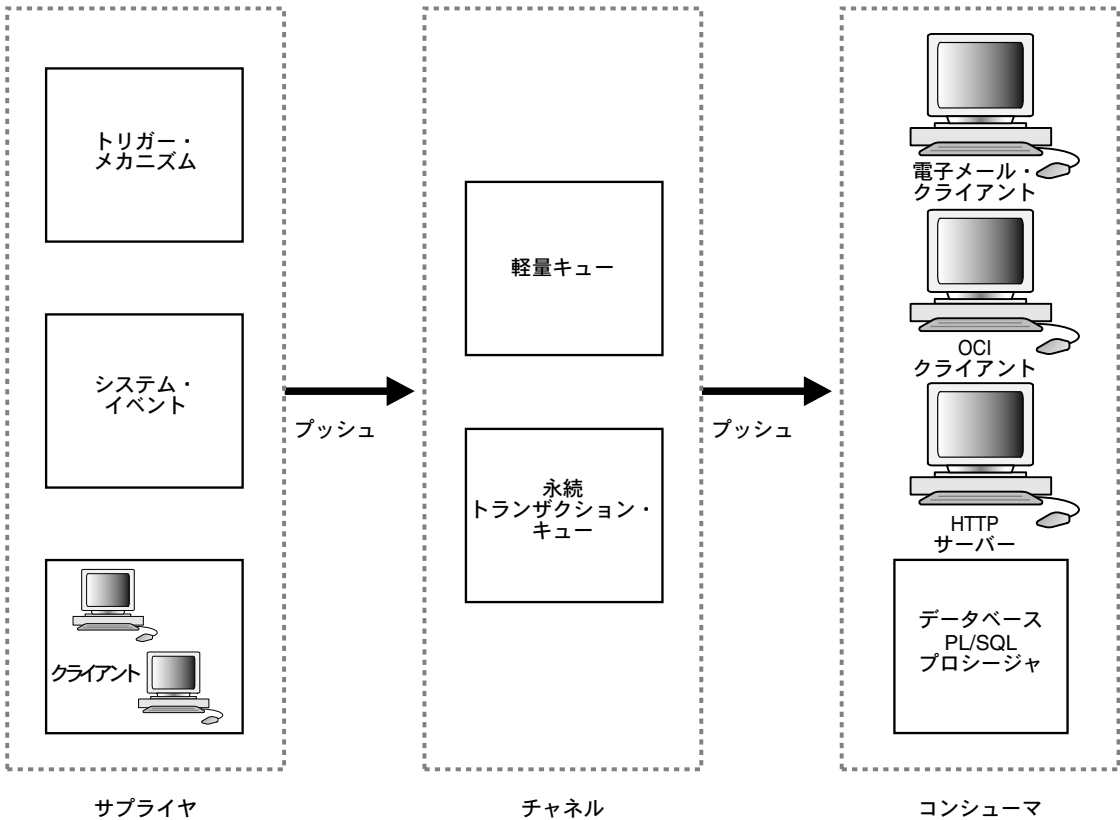
PL/SQL デキュー・オプション	OCIAQDeqOptions 属性
consumer_name	OCI_ATTR_CONSUMER_NAME
dequeue_mode	OCI_ATTR_DEQ_MODE
navigation	OCI_ATTR_NAVIGATION
visibility	OCI_ATTR_VISIBILITY
wait	OCI_ATTR_WAIT

PL/SQL デキュー・オプション	OCIAQDeqOptions 属性
msgid	OCI_ATTR_DEQ_MSGID
correlation	OCI_ATTR_CORRELATION

パブリッシュ・サブスクライブの通知

パブリッシュ・サブスクライブの通知機能により、OCI アプリケーションでは、クライアント通知の直接受信、通知を送信できる電子メール・アドレスの登録、通知を送信できる HTTP URL の登録、あるいは通知に対してコールされる PL/SQL プロシージャの登録を行うことができます。図 9-2 「パブリッシュ・サブスクライブのモデル」は、そのプロセスを示しています。

図 9-2 パブリッシュ・サブスクライブのモデル



OCI アプリケーションでは次の処理を行います。

- AQ 名前空間に通知の実行を登録し、エンキューが発生したときに通知を受け取ります。
- データベース・イベントのサブスクリプションの実行を登録し、そのイベントがトリガーされたときに通知を受け取ります。
- 一時的な登録の解除またはすべての登録の破棄など、登録を管理します。
- 登録されたクライアントに通知を送信します。

前述のすべての場合に、OCI アプリケーションでの通知の直接受信、事前に指定された電子メール・アドレスへの通知の送信、事前に定義された HTTP URL への通知の送信、または通知の結果として、事前に指定したデータベース PL/SQL プロシージャのコールが可能です。

登録されたクライアントは、イベントが発生した時、または明示的な AQ エンキューにある時、非同期に通知を受け取ります。クライアントは、データベースに接続する必要はありません。

関連項目：

- アドバンスト・キューイングの詳細は、9-48 ページの「[OCI およびアドバンスト・キューイング](#)」を参照してください。
- キューの作成の詳細、AQ の概念、機能および例などの詳細は、『[Oracle9i アプリケーション開発者ガイド - アドバンスト・キューイング](#)』のアドバンスト・キューイングの章を参照してください。
- トリガーの作成の詳細は、『[Oracle9i SQL リファレンス](#)』のコマンドに関する章を参照してください。

パブリッシュ・サブスクライブの登録関数

登録する方法には、次の 2 通りがあります。

- 直接データベースに登録します。この方法は単純で、登録は即時に有効になります。
- オープン登録します。データベースが登録要求を受け取る LDAP を使用して登録します。この方法は、クライアントがデータベースに接続できない場合（データベースの停止中にクライアントがデータベースのオープン・イベントに登録する場合）、またはクライアントが複数のデータベース内の同じイベントまたは複数のイベントに一度に登録する場合に役立ちます。

次の項では、これらの登録方法について説明します。

データベースへのパブリッシュ・サブスクライブの直接登録

イベント通知の登録および受信を行うには、OCI アプリケーションで次のステップを実行する必要があります。適切なイベント・トリガーまたは AQ エンキューがセットアップされていることを前提としています。初期化パラメータ COMPATIBLE はリリース 8.1 以上に設定されている必要があります。

関連項目：

- 各関数の詳細は、16-85 ページの「[アドバンスト・キューイング関数およびパブリッシュ・サブスクライブ関数](#)」を参照してください。
- アプリケーション内でのこれらの関数の使用例については、9-60 ページの「[パブリッシュ・サブスクライブの直接登録の例](#)」を参照してください。

注意： パブリッシュ・サブスクライブ機能は、マルチスレッド・プラットフォーム上でのみ使用できます。

1. OCI_EVENTS モードで OCIInitialize() を実行し、アプリケーションで通知の登録および受信を行うことを指定します。クライアント上で、通知専用のリスニング・スレッドが起動します。
2. ハンドル・タイプを OCI_HTYPE_SUBSCRIPTION に指定して OCIHandleAlloc() を実行し、サブスクリプション・ハンドルを割り当てます。
3. OCIAttrSet() を実行し、次のサブスクリプション・ハンドル属性を設定します。
 - OCI_ATTR_SUBSCR_NAME — サブスクリプション名
 - OCI_ATTR_SUBSCR_NAMESPACE — サブスクリプションの名前空間
 - OCI_ATTR_SUBSCR_CALLBACK — 通知コールバック
 - OCI_ATTR_SUBSCR_CTX — コールバック・コンテキスト
 - OCI_ATTR_SUBSCR_PAYLOAD — 送信用ペイロード・バッファ
 - OCI_ATTR_SUBSCR_RECPT — 受信者名
 - OCI_ATTR_SUBSCR_RECPTPROTO — 通知を受信するプロトコル
 - OCI_ATTR_SUBSCR_RECPTPRES — 通知を受信するための表示

サブスクリプションを登録する前に、OCI_ATTR_SUBSCR_NAME、OCI_ATTR_SUBSCR_NAMESPACE および OCI_ATTR_SUBSCR_RECPTPROTO を設定しておく必要があります。

OCI_ATTR_SUBSCR_RECPTPROTO を OCI_SUBSCR_PROTO_OCI に設定した場合は、OCI_ATTR_SUBSCR_CALLBACK および OCI_ATTR_SUBSCR_CTX も設定する必要があります。

OCI_ATTR_SUBSCR_RECPTPROTO を OCI_SUBSCR_PROTO_MAIL、OCI_SUBSCR_PROTO_SERVER または OCI_SUBSCR_PROTO_HTTP に設定した場合は、OCI_ATTR_SUBSCR_RECPT も設定する必要があります。

OCI_ATTR_SUBSCR_CALLBACK と OCI_ATTR_SUBSCR_RECPT を同時に設定すると、アプリケーション・エラーが発生します。

OCI_ATTR_SUBSCR_PAYLOAD は、サブスクリプションへの送信前に設定する必要があります。

関連項目： これらの属性の詳細は、A-54 ページの「[サブスクリプション・ハンドル属性](#)」を参照してください。

4. OCI_ATTR_SUBSCR_RECPTPROTO を OCI_SUBSCR_PROTO_OCI に設定した場合は、サブスクリプション・ハンドルで使用するコールバック・ルーチンを定義します。

関連項目： 9-58 ページ「[通知コールバック](#)」

5. OCI_ATTR_SUBSCR_RECPTPROTO を OCI_SUBSCR_PROTO_SERVER に設定した場合は、通知に対してコールする PL/SQL プロシージャをデータベースに定義します。

関連項目： 9-59 ページ「[通知プロシージャ](#)」

6. OCISubscriptionRegister() を実行してサブスクリプションに登録します。このコールにより、複数のサブスクリプションを同時に登録できます。

パブリッシュ・サブスクライブのオープン登録

オープン登録の前提条件は、次のとおりです。

- LDAP（オープン登録）を使用する登録では、クライアントがエンタープライズ・ユーザーであることが必要です。

関連項目： 『Oracle Advanced Security 管理者ガイド』のエンタープライズ・ユーザー・セキュリティの管理に関する項を参照してください。

- データベースの互換性は、リリース 1（9.0.1）以上であることが必要です。

- LDAP_REGISTRATION_ENABLED を TRUE に設定する必要があります。次のように設定します。

```
ALTER SYSTEM SET LDAP_REGISTRATION_ENABLED=TRUE
```

デフォルトは FALSE です。

- LDAP_REG_SYNC_INTERVAL を LDAP からの登録をリフレッシュする時間隔（秒単位）に設定します。

```
ALTER SYSTEM SET LDAP_REG_SYNC_INTERVAL = time_interval
```

デフォルトは 0（ゼロ）で、この場合リフレッシュは行われません。

- データベースで LDAP の登録情報を即時にリフレッシュするには、次のように設定します。

```
ALTER SYSTEM REFRESH LDAP_REGISTRATION
```

Oracle Enterprise Security Manager（OESM）を使用したオープン登録の手順は、次のとおりです。

1. 各エンタープライズ・ドメインで、エンタープライズ・ロールの ENTERPRISE_AQ_USER_ROLE を作成します。
2. エンタープライズ・ドメイン内の各データベースについて、グローバル・ロールの GLOBAL_AQ_USER_ROLE をエンタープライズ・ロールの ENTERPRISE_AQ_USER_ROLE に追加します。
3. 各エンタープライズ・ドメインについて、エンタープライズ・ロールの ENTERPRISE_AQ_USER_ROLE を、管理コンテキストの下のある権限グループの cn=OracleDBAUsers に追加します。
4. データベース内のイベントへの登録が承認された各エンタープライズ・ユーザーに対して、エンタープライズ・ロールの ENTERPRISE_AQ_USER_ROLE を付与します。

OCI を使用した LDAP のオープン登録

1. mode パラメータを OCI_EVENTS | OCI_USE_LDAP に設定して、OCIInitialize() をコールします。
2. OCIAttrSet() をコールし、次の環境ハンドル属性を設定して LDAP にアクセスします。
 - OCI_ATTR_LDAP_HOST: LDAP サーバーが常駐しているホスト名。
 - OCI_ATTR_LDAP_PORT: LDAP サーバーがリスニングしているポート。
 - OCI_ATTR_BIND_DN: LDAP サーバーにログインするための識別名。通常はエンタープライズ・ユーザーの DN です。
 - OCI_ATTR_LDAP_CRED: クライアントの認証に使用する資格証明。簡易認証（ユーザー名 / パスワード）の場合のパスワードなどです。

- OCI_ATTR_WALL_LOC: SSL 認証の場合のクライアント Wallet の場所。
- OCI_ATTR_LDAP_AUTH: 認証方式コード。

関連項目： 認証モードの完全なリストは、A-5 ページの「OCI_ATTR_LDAP_AUTH」を参照してください。

- OCI_ATTR_LDAP_CTX: LDAP サーバーの Oracle に対する管理コンテキスト。
3. ハンドル・タイプを OCI_HTYPE_SUBSCRIPTION に指定して OCIHandleAlloc() をコールし、サブスクリプション・ハンドルを割り当てます。
 4. 記述子タイプを OCI_DTYPE_SRVDN に指定して OCIDescriptorAlloc() をコールし、サーバーの DN 記述子を割り当てます。
 5. OCIAttrSet() をコールし、サーバーの DN 記述子属性を OCI_ATTR_SERVER_DN に設定します。これは、クライアントが通知を受信するデータベースの識別名です。OCIAttrSet() は、この属性について複数回コールできるため、登録には複数のデータベース・サーバーが含まれます。
 6. OCIAttrSet() をコールし、次の属性にサブスクリプション・ハンドル属性を設定します。
 - OCI_ATTR_SUBSCR_NAME: サブスクリプション名
 - OCI_ATTR_SUBSCR_NAMESPACE: サブスクリプションの名前空間
 - OCI_ATTR_SUBSCR_CALLBACK: 通知コールバック
 - OCI_ATTR_SUBSCR_CTX: コールバック・コンテキスト
 - OCI_ATTR_SUBSCR_PAYLOAD: 送信用ペイロード・バッファ
 - OCI_ATTR_SUBSCR_RECPT: 受信者名
 - OCI_ATTR_SUBSCR_RECPTPROTO: 通知を受信するプロトコル
 - OCI_ATTR_SUBSCR_RECPTRES: 通知を受信するための表示
 - OCI_ATTR_SUBSCR_SERVER_DN: ステップ 5 で設定した記述子ハンドル
 7. OCISubscriptionRegister() をコールし、サブスクリプションを登録します。データベースが LDAP にアクセスして新規登録を取り出すと、登録が有効になります。取り出す頻度は、REG_SYNC_INTERVAL の値によって決定されます。

パブリッシュ・サブスクライブ通知の管理に使用する OCI 関数

パブリッシュ・サブスクライブ通知の管理には、次の関数を使用します。

表 9-1 パブリッシュ・サブスクライブの関数

関数	用途
<code>OCISubscriptionDisable()</code>	サブスクリプションを無効にします。
<code>OCISubscriptionEnable()</code>	サブスクリプションを有効にします。
<code>OCISubscriptionPost()</code>	サブスクリプションを送信します。
<code>OCISubscriptionRegister()</code>	サブスクリプションを登録します。
<code>OCISubscriptionUnRegister()</code>	サブスクリプションの登録を解除します。

通知コールバック

クライアントは、登録したサブスクリプションに対してアクティビティが発生したときにコールされる、通知コールバックを登録する必要があります。たとえば、AQ 名前空間では、目的のメッセージがエンキューされたときに通知コールバックが発生します。

このコールバックは通常、サブスクリプション・ハンドルの `OCI_ATTR_SUBSCR_CALLBACK` 属性によって設定されます。

関連項目： 詳細は、A-54 ページの「[サブスクリプション・ハンドル属性](#)」を参照してください。

通知コールバックからは、`OCI_CONTINUE` 値が戻される必要があります。また、次の仕様に準拠している必要があります。

```
typedef ub4 (*OCISubscriptionNotify) ( dvoid          *pCtx,
                                       OCISubscription *pSubscrHp,
                                       dvoid          *pPayload,
                                       ub4            iPayloadLen,
                                       dvoid          *pDescriptor,
                                       ub4            iMode);
```

パラメータは次のように記述します。

pCtx (IN) コールバックが登録されたときに指定されたユーザー定義コンテキスト。

pSubscrHp (IN) コールバックが登録されたときに指定されたサブスクリプション・ハンドル。

pPayload (IN) この通知のペイロード。このリリースでは、ペイロードの `ub1 *` (バイト・シーケンス) のみがサポートされています。

iPayloadLen (IN) この通知のペイロードの長さ。

pDescriptor (IN) 名前空間固有の記述子。この記述子から名前空間指定のパラメータを抽出できます。この記述子の構造はユーザーには不透明で、型は名前空間によって異なります。

記述子の属性は、名前空間固有です。アドバンスト・キューイングの場合、記述子は `OCI_DTYPE_AQNFY` です。この記述子の属性は、次のとおりです。

- キュー名 — `OCI_ATTR_QUEUE_NAME`
- コンシューマ名 — `OCI_ATTR_CONSUMER_NAME`
- メッセージ ID — `OCI_ATTR_NFY_MSGID`
- メッセージ・プロパティ — `OCI_ATTR_MSG_PROP`

関連項目： OCI およびアドバンスト・キューイングの詳細は、9-48 ページの「[OCI およびアドバンスト・キューイング](#)」を参照してください。

iMode (IN) コール固有モードです。有効な値は次のとおりです。

- `OCI_DEFAULT` — デフォルトのコールを実行します。

通知プロシージャ

実行が登録されているサブスクリプションでなんらかのアクティビティが発生したときにコールする PL/SQL プロシージャは、データベース内に作成する必要があります。

このプロシージャは通常、サブスクリプション・ハンドルの `OCI_ATTR_SUBSCR_RECPT` 属性によって設定されます。詳細は、「サブスクリプション・ハンドル属性」を参照してください。

関連項目：

- A-54 ページの「[サブスクリプション・ハンドル属性](#)」を参照してください。
- PL/SQL プロシージャの仕様の詳細は、『Oracle9i PL/SQL パッケージ・プロシージャおよびタイプ・リファレンス』の「AQ PL/SQL コールバック」の項を参照してください。

パブリッシュ・サブスクライブの直接登録の例

この例では、システム・イベント、クライアント通知およびアドバンスト・キューイングが連動して、パブリッシュ・サブスクライブ通知を実装する方法を示します。

次の PL/SQL コードでは、ユーザー・スキーマ *pubsub* 環境でパブリッシュ・サブスクライブ・メカニズムをサポートするために必要なすべてのオブジェクトを作成します。このコードでは、エージェント *snoop* が、ログイン・イベント時にパブリッシュされるメッセージをサブスクライブします。ユーザー *pubsub* には、アドバンスト・キューイング機能を使用するために *AQ_ADMINISTRATOR_ROLE* および *AQ_USER_ROLE* の権限が必要です。システム・イベントのトリガーを有効にするには、初期化パラメータ *_SYSTEM_TRIG_ENABLED* を TRUE（デフォルト）に設定する必要があります。

```
Rem -----
REM create queue table for persistent multiple consumers
Rem -----
connect pubsub/pubsub;
Rem Create or replace a queue table
begin
    DBMS_AQADM.CREATE_QUEUE_TABLE(
        QUEUE_TABLE=>'pubsub.raw_msg_table',
        MULTIPLE_CONSUMERS => TRUE,
        QUEUE_PAYLOAD_TYPE =>'RAW',
        COMPATIBLE => '8.1.5');
end;
/
Rem -----
Rem Create a persistent queue for publishing messages
Rem -----
Rem Create a queue for logon events
begin
    DBMS_AQADM.CREATE_QUEUE(QUEUE_NAME=>'pubsub.logon',
        QUEUE_TABLE=>'pubsub.raw_msg_table',
        COMMENT=>'Q for error triggers');
end;
/
Rem -----
Rem Start the queue
Rem -----
begin
    DBMS_AQADM.START_QUEUE('pubsub.logon');
end;
/
Rem -----
Rem define new_enqueue for convenience
Rem -----
```

```

create or replace procedure new_enqueue(queue_name in varchar2,
                                     payload in raw ,
correlation in varchar2 := NULL,
exception_queue in varchar2 := NULL)
as
    enq_ct      dbms_aq.enqueue_options_t;
    msg_prop    dbms_aq.message_properties_t;
    enq_msgid   raw(16);
    userdata    raw(1000);
begin
    msg_prop.exception_queue := exception_queue;
    msg_prop.correlation := correlation;
    userdata := payload;
    DBMS_AQ.ENQUEUE(queue_name,enq_ct, msg_prop,userdata,enq_msgid);
end;
/
Rem -----
Rem add subscriber with rule based on current user name,
Rem using correlation_id
Rem -----
declare
subscriber sys.aq$_agent;
begin
    subscriber := sys.aq$_agent('SNOOP', null, null);
    dbms_aqadm.add_subscriber(queue_name => 'pubsub.logon',
                             subscriber => subscriber,
                             rule => 'CORRID = ''SCOTT'' ');
end;
/
Rem -----
Rem create a trigger on logon on database
Rem -----
Rem create trigger on after logon
create or replace trigger systrig2
    AFTER LOGON
    ON DATABASE
    begin
        new_enqueue('pubsub.logon', hextoraw('9999'), dbms_standard.login_user);
    end;
/
Rem -----
Rem create a PL/SQL callback for notification of logon
Rem of user 'scott' on database
Rem -----
Rem
create or replace procedure plsqliotifySnoop(

```

```
context raw, reginfo sys.aq$_reg_info, descr sys.aq$_descriptor,
payload raw, payloadl number)
as
begin
  dbms_output.putline('Notification : User Scott Logged on\n');
end;
/
```

サブスクリプションが作成された後、クライアントはコールバック関数を使用して通知を登録する必要があります。次のコード例では、登録に必要なステップを実行します。ここでは、わかりやすくするために、セッション・ハンドルの割当ておよび初期化のステップを省略しています。

```
...
ub4 namespace = OCI_SUBSCR_NAMESPACE_AQ;
/* callback function for notification of logon of user 'scott' on database */
ub4 notifySnoop(ctx, subscrhp, pay, payl, desc, mode)
  dvoid *ctx;
  OCISubscription *subscrhp;
  dvoid *pay;
  ub4 payl;
  dvoid *desc;
  ub4 mode;
{
  printf("Notification : User Scott Logged on\n");
}
int main()
{
  OCISession *authp = (OCISession *) 0;
  OCISubscription *subscrhpSnoop = (OCISubscription *)0;
  OCISubscription *subscrhpSnoopMail = (OCISubscription *)0;
  OCISubscription *subscrhpSnoopServer = (OCISubscription *)0;

  /*****
  Initialize OCI Process/Environment
  Initialize Server Contexts
  Connect to Server
  Set Service Context
  *****/
  /* Registration Code Begins */
  /* Each call to initSubscriptionHn allocates
     and Initialises a Registration Handle */

  /* Register for OCI notification */
  initSubscriptionHn( &subscrhpSnoop, /* subscription handle */
                    "PUBSUB.SNOOP:ADMIN", /* subscription name */
```

```

/* <queue_name>:<agent_name> */
    (dvoid*)notifySnoop, /* callback function */
    OCI_SUBSCR_PROTO_OCI, /* receive with protocol */
    (char *)0, /* recipient address */
    OCI_SUBSCR_PRES_DEFAULT); /* receive with presentation */

/* Register for email notification */
    initSubscriptionHn(    &subscrhpSnoopMail,    /* subscription handle */
        "PUBSUB.SNOOP:ADMIN", /* subscription name */
/* <queue_name>:<agent_name> */
    (dvoid*)0, /* callback function */
    OCI_SUBSCR_PROTO_MAIL, /* receive with protocol */
    "xyz@company.com", /* recipient address */
    OCI_SUBSCR_PRES_DEFAULT); /* receive with presentation */

/* Register for server to server notification */
    initSubscriptionHn(    &subscrhpSnoopServer,    /* subscription handle */
        "PUBSUB.SNOOP:ADMIN", /* subscription name */
/* <queue_name>:<agent_name> */
    (dvoid*)0, /* callback function */
    OCI_SUBSCR_PROTO_SERVER, /* receive with protocol */
    "pubsub.plsqlnotifySnoop", /* recipient address */
    OCI_SUBSCR_PRES_DEFAULT); /* receive with presentation */
/*****
The Client Process does not need a live Session for Callbacks
End Session and Detach from Server
*****/
    OCISessionEnd ( svchp, errhp, authp, (ub4) OCI_DEFAULT);
    /* detach from server */
    OCIserverDetach( srvhp, errhp, OCI_DEFAULT);
    while (1) /* wait for callback */
        sleep(1);
}

void initSubscriptionHn (subscrhp,
                        subscriptionName,
                        func,
                        recpproto,
                        recpaddr,
                        recppres)
OCISubscription **subscrhp;
char * subscriptionName;
dvoid * func;
ub4 recpproto;
char * recpaddr;
ub4 recppres;
{

```

```
/* allocate subscription handle */
(void) OCIHandleAlloc((dvoid *) envhp, (dvoid **)subscrhp,
    (ub4) OCI_HTYPE_SUBSCRIPTION,
    (size_t) 0, (dvoid **) 0);

/* set subscription name in handle */
(void) OCIAttrSet((dvoid *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
    (dvoid *) subscriptionName,
    (ub4) strlen((char *)subscriptionName),
    (ub4) OCI_ATTR_SUBSCR_NAME, errhp);

/* set callback function in handle */
if (func)
    (void) OCIAttrSet((dvoid *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
        (dvoid *) func, (ub4) 0,
        (ub4) OCI_ATTR_SUBSCR_CALLBACK, errhp);

/* set context in handle */
(void) OCIAttrSet((dvoid *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
    (dvoid *) 0, (ub4) 0,
    (ub4) OCI_ATTR_SUBSCR_CTX, errhp);

/* set namespace in handle */
(void) OCIAttrSet((dvoid *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
    (dvoid *) &namespace, (ub4) 0,
    (ub4) OCI_ATTR_SUBSCR_NAMESPACE, errhp);
checkerr(errhp, OCISubscriptionRegister(svchp, subscrhp, 1, errhp,
    OCI_DEFAULT));

/* set receive with protocol in handle */
(void) OCIAttrSet((dvoid *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
    (dvoid *) &recpproto, (ub4) 0,
    (ub4) OCI_ATTR_SUBSCR_RECPTPROTO, errhp);

/* set recipient address in handle */
if (recpaddr)
    (void) OCIAttrSet((dvoid *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
        (dvoid *) &recpaddr, (ub4) strlen(recpaddr),
        (ub4) OCI_ATTR_SUBSCR_RECPT, errhp);

/* set receive with presentation in handle */
(void) OCIAttrSet((dvoid *) *subscrhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
    (dvoid *) &recppres, (ub4) 0,
    (ub4) OCI_ATTR_SUBSCR_RECPTPRES, errhp);
}
...
```


ユーザー SCOTT がデータベースにログインした場合、クライアントは電子メールで通知を受け取り、コールバック関数 `notifySnoop` がコールされます。電子メール通知は、アドレス `xyz@company.com` に送信され、PL/SQL プロシージャ `plsqlnotifySnoop` もデータベースでコールされます。

パブリッシュ・サブスクライブの LDAP 登録の例

次のコード・フラグメントでは、LDAP 登録の方法を示します。プログラム・コメントはすべて目を通してください。

```
...

/* TO use LDAP registration feature, OCI_EVENTS | OCI_USE_LDAP must be set
   in OCIInitialize: */

(void) OCIInitialize((ub4) OCI_EVENTS|OCI_OBJECT|OCI_USE_LDAP, (dvoid *)0,
                    (dvoid * (*)(dvoid *, size_t)) 0,
                    (dvoid * (*)(dvoid *, dvoid *, size_t))0,
                    (void (*)(dvoid *, dvoid *)) 0 );

...

/* set LDAP attributes in the environment handle */

/* LDAP host name */
(void) OCIAttrSet((dvoid *)envhp, OCI_HTYPE_ENV, (dvoid *)"yow", 3
                OCI_ATTR_LDAP_HOST, (OCIError *)errhp);

/* LDAP server port */
ldap_port = 389;
(void) OCIAttrSet((dvoid *)envhp, OCI_HTYPE_ENV, (dvoid *)&ldap_port,
                (ub4)0, OCI_ATTR_LDAP_PORT, (OCIError *)errhp);

/* bind DN of the client, normally the enterprise user name */
(void) OCIAttrSet((dvoid *)envhp, OCI_HTYPE_ENV, (dvoid *)"cn=orcladmin",
                12, OCI_ATTR_BIND_DN, (OCIError *)errhp);

/* password of the client */
(void) OCIAttrSet((dvoid *)envhp, OCI_HTYPE_ENV, (dvoid *)"welcome",
                7, OCI_ATTR_LDAP_CRED, (OCIError *)errhp);

/* authentication method is "simple", username/password authentication */
ldap_auth = 0x01;
(void) OCIAttrSet((dvoid *)envhp, OCI_HTYPE_ENV, (dvoid *)&ldap_auth,
                (ub4)0, OCI_ATTR_LDAP_AUTH, (OCIError *)errhp);
```

```
/* administrative context: this is the DN above cn=oraclecontext */
(void) OCIAAttrSet((dvoid *)envhp, OCI_HTYPE_ENV, (dvoid *)"cn=acme,cn=com",
                  14, OCI_ATTR_LDAP_CTX, (OCIError *)errhp);

...

/* retrieve the LDAP attributes from the environment handle */

/* LDAP host */
(void) OCIAAttrGet((dvoid *)envhp, OCI_HTYPE_ENV, (dvoid *)&buf,
                  &szp, OCI_ATTR_LDAP_HOST, (OCIError *)errhp);

/* LDAP server port */
(void) OCIAAttrGet((dvoid *)envhp, OCI_HTYPE_ENV, (dvoid *)&intval,
                  0, OCI_ATTR_LDAP_PORT, (OCIError *)errhp);

/* client binding DN */
(void) OCIAAttrGet((dvoid *)envhp, OCI_HTYPE_ENV, (dvoid *)&buf,
                  &szp, OCI_ATTR_BIND_DN, (OCIError *)errhp);

/* client password */
(void) OCIAAttrGet((dvoid *)envhp, OCI_HTYPE_ENV, (dvoid *)&buf,
                  &szp, OCI_ATTR_LDAP_CRED, (OCIError *)errhp);

/* administrative context */
(void) OCIAAttrGet((dvoid *)envhp, OCI_HTYPE_ENV, (dvoid *)&buf,
                  &szp, OCI_ATTR_LDAP_CTX, (OCIError *)errhp);

/* client authentication method */
(void) OCIAAttrGet((dvoid *)envhp, OCI_HTYPE_ENV, (dvoid *)&intval,
                  0, OCI_ATTR_LDAP_AUTH, (OCIError *)errhp);

...

/* to set up the server DN descriptor in the subscription handle */

/* allocate a server DN descriptor, dn is of type "OCIServerDNs **",
   subhp is of type "OCISubscription **" */
(void) OCIDescriptorAlloc((dvoid *)envhp, (dvoid **)dn,
                          (ub4) OCI_DTYPE_SRVDN, (size_t)0, (dvoid **)0);

/* now *dn is the server DN descriptor, add the DN of the first database
   that we want to register */
(void) OCIAAttrSet((dvoid *)dn, (ub4) OCI_DTYPE_SRVDN,
                  (dvoid *)"cn=server1,cn=oraclecontext,cn=acme,cn=com",
                  42, (ub4)OCI_ATTR_SERVER_DN, errhp);
```

```
/* add the DN of another database in the descriptor */
(void) OCIAAttrSet((dvoid *)*dn, (ub4) OCI_DTYPE_SRVDN,
                  (dvoid *) "cn=server2,cn=oraclecontext,cn=acme,cn=com",
                  42, (ub4)OCI_ATTR_SERVER_DN, errhp);

/* set the server DN descriptor into the subscription handle */
(void) OCIAAttrSet((dvoid *) *subhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
                  (dvoid *) *dn, (ub4)0, (ub4) OCI_ATTR_SERVER_DNS, errhp);

...

/* now we will try to get the server DN information from the subscription
   handle */

/* first, get the server DN descriptor out */
(void) OCIAAttrGet((dvoid *) *subhp, (ub4) OCI_HTYPE_SUBSCRIPTION,
                  (dvoid *)dn, &szp, OCI_ATTR_SERVER_DNS, errhp);

/* then, get the number of server DNs in the descriptor */
(void) OCIAAttrGet((dvoid *) *dn, (ub4)OCI_DTYPE_SRVDN, (dvoid *)&intval,
                  &szp, (ub4)OCI_ATTR_DN_COUNT, errhp);

/* allocate an array of char * to hold server DN pointers returned by
   oracle */
if (intval)
{
    arr = (char **)malloc(intval*sizeof(char *));
    (void) OCIAAttrGet((dvoid *)*dn, (ub4)OCI_DTYPE_SRVDN, (dvoid *)arr,
                      &intval, (ub4)OCI_ATTR_SERVER_DN, errhp);
}

/* OCISubscriptionRegister() calls have two modes: OCI_DEFAULT and
   OCI_REG_LDAPONLY. If OCI_DEFAULT is used, there should be only one
   server DN in the server DN descriptor. The registration request will
   be sent to the database. If a database connection is not available,
   the registration request will be detoured to the LDAP server. On the
   other hand, if mode OCI_REG_LDAPONLY is used the registration request
   will be directly sent to LDAP. This mode should be used when there are
   more than one server DNs in the server DN descriptor, or we are sure
   that a database connection is not available.

   In this example, two DNs are entered; so we should use mode
   OCI_REG_LDAPONLY in register. */
OCISubscriptionRegister(svchp, subhp, 1, errhp, OCI_REG_LDAPONLY);

...
```

```
/* as OCISubscriptionRegister(), OCISubscriptionUnregister() also has
   mode OCI_DEFAULT and OCI_REG_LDAPONLY. The usage is the same. */

OCISubscriptionUnRegister(svchp, *subhp, errhp, OCI_REG_LDAPONLY);
}

...
```

第 II 部

OCI オブジェクトの概念

第 II 部は、OCI でのユーザー定義オブジェクトの使用方法を説明した章で構成されています。

- 第 10 章「OCI オブジェクト・リレーショナル・プログラミング」では、OCI でのオブジェクトの概念とオブジェクト・リレーショナル・プログラミングについて紹介します。型の変更についても説明します。
- 第 11 章「オブジェクト・リレーショナル・データ型」では、オブジェクト・データ型と、データベース・オブジェクトを C 構造体で表す方法について説明します。また、データ型をマッピングし操作する OCI 関数についても説明します。オブジェクト・リレーショナル・データ型のバインドと定義についても説明します。AnyType インタフェース、AnyData インタフェースおよび AnyDataSet インタフェースがあります。
- 第 12 章「ダイレクト・パス・ロード」では、データのダイレクト・パス・ロードについて説明します。
- 第 13 章「オブジェクトのキャッシュおよびナビゲーション」では、オブジェクト・キャッシュと、オブジェクトでのナビゲート方法について説明します。
- 第 14 章「Object Type Translator (OTT)」では、OTT を使用して、データベース型定義をホスト言語表現に変換する方法について説明します。

OCI オブジェクト・リレーショナル・プログラミング

この章では、オブジェクトを Oracle データベース・サーバーで操作するための OCI の機能を紹介しします。また、OCI のオブジェクト・ナビゲーション関数コールについても説明します。この章は、次の項目で構成されています。

- [OCI オブジェクトの概要](#)
- [OCI でのオブジェクトの操作](#)
- [OCI オブジェクト・アプリケーションの開発](#)
- [型の継承](#)

OCI オブジェクトの概要

OCI には、データベース・アクセス管理と SQL 文の処理に使用する関数が用意されています。これらの関数の詳細は、このマニュアルの第 I 部に記載されています。

OCI によって、アプリケーションは、スカラー値、コレクション、任意のオブジェクト型のインスタンスなど、Oracle データベース・サーバー内のすべてのデータ型にアクセスできます。次のデータ型があります。

- オブジェクト
- 可変長配列 (VARRAY)
- NESTED TABLE (多重集合)
- 参照 (REF)
- LOB

Oracle サーバーのオブジェクト機能を十分に活用するには、ほとんどのアプリケーションでは単にオブジェクトにアクセスするのみでは十分ではありません。アプリケーションでは、オブジェクトを取り出した後、そのオブジェクトから他のオブジェクトへ参照を通してナビゲートする必要があります。OCI はそのための機能を提供します。OCI のオブジェクト・ナビゲーション・コールを使用すると、アプリケーションはオブジェクトに対して次の関数を実行できます。

- オブジェクトの作成、アクセス、ロック、削除、コピーおよびフラッシュ
- オブジェクトとそのメタオブジェクトへの参照の取得
- オブジェクト属性の値の動的な取得および設定

OCI ナビゲーション・コールについては、この章の後半で詳しく説明します。

また、OCI は、Oracle のデータベースに格納されている型情報にアクセスする機能も備えています。アプリケーションでは、OCIDescribeAny() 関数を使用して、データベースに格納されている型に関するほとんどの情報（メソッド、属性、型メタデータなど）にアクセスできます。

関連項目： OCIDescribeAny() は、[第 6 章「スキーマ・メタデータの記述」](#)で説明します。

Oracle オブジェクトを操作するアプリケーションでは、オブジェクトをホスト言語形式で表現する手段が必要です。Oracle には、Object Type Translator (OTT) というユーティリティがあります。このユーティリティを使用すると、データベース内の型定義を C の構造体宣言に変換できます。宣言はヘッダー・ファイルに格納され、それを OCI アプリケーションに組み込むことができます。

型定義が C で表現されている場合、属性の型は特別な C 変数型にマップされます。OCI には、アプリケーションによるこれらのデータ型の操作とそれによるオブジェクトの属性の操作を可能にする、一連のデータ型マッピング関数および操作関数が含まれています。

関連項目： これらの関数については、[第 11 章「オブジェクト・リレーショナル・データ型」](#)で詳しく説明します。

オブジェクトに関する用語は混同することがあります。この章では、以降「オブジェクト」と「インスタンス」という用語は両方とも、データベースに格納されているオブジェクトか、オブジェクト・キャッシュにあるオブジェクトのいずれかを指します。

OCI でのオブジェクトの操作

リレーショナル OCI アプリケーションを制御するプログラミング原理の多くは、オブジェクト・リレーショナル・アプリケーションにも適用されます。オブジェクト・リレーショナル・アプリケーションでは、標準 OCI コールを使用してデータベース接続を確立し、SQL 文を処理します。その違いは、発行した SQL 文でオブジェクト参照を取り出し、その参照を OCI のオブジェクト関数で操作できる点にあります。オブジェクトは、値インスタンスとして（オブジェクト参照を使用せずに）直接操作することもできます。

基本的なオブジェクト・プログラム構造体

オブジェクトを使用する OCI アプリケーションの基本的な構造体は、本質的にはリレーショナル OCI アプリケーションと同じです。これについては、2-3 ページの「[OCI プログラム構造](#)」で説明しています。ここでは、基本的なオブジェクト機能についての追加情報とともに、そのモデルをもう一度示します。

1. OCI プログラミング環境を初期化します。環境は、オブジェクト・モードで初期化してください。

また、アプリケーションでは通常、ヘッダー・ファイルからデータベース・オブジェクトの C 構造体の表現を組み込む必要があります。

関連項目： これらの構造体はプログラマであれば作成できますが、より簡単な方法として、[第 14 章「Object Type Translator \(OTT\)」](#)で説明するように、Object Type Translator (OTT) でも生成できます。

2. 必要なハンドルを割り当て、サーバーとの接続を確立します。
3. SQL 文を準備して実行します。これは、ローカル・(クライアント側) ステップで、プレースホルダのバインドと出力変数の定義を行います。オブジェクト・リレーショナル・アプリケーションの場合、この SQL 文はオブジェクトへの参照 (REF) を戻します。

注意： オブジェクトは参照（REF）のみでなく、オブジェクト全体をフェッチすることもできます。参照可能なオブジェクトを選択（SELECT）する場合は、それを確保するのではなく、そのオブジェクトを値によって取得します。あるいは、10-15 ページの「[埋込みオブジェクトのフェッチ](#)」の説明のように、参照不可能なオブジェクトを選択できます。

4. プリコンパイルされた SQL 文をデータベース・サーバーに関連付けて実行します。
5. 戻された結果をフェッチします。

オブジェクト・リレーショナル・アプリケーションでは、このステップで REF を取り出し、参照対象のオブジェクトを確保します。オブジェクトを確保した後、次の一部またはすべてを実行します。

- オブジェクトの属性を操作し、それに使用済みのマークを設定します。
 - 別の 1 個のオブジェクトまたは一連のオブジェクトへの REF をたどります。
 - 型および属性の情報にアクセスします。
 - 複合オブジェクト検索グラフをナビゲートします。
 - 変更したオブジェクトをサーバーにフラッシュします。
6. トランザクションをコミットします。このステップでは暗黙的に、変更されたすべてのオブジェクトをサーバーにフラッシュし、変更をコミットします。
 7. 再利用しない文とハンドルを解放するか、プリコンパイルされた SQL 文を再実行します。

この章の後の項で、すべてのステップを詳しく説明します。

関連項目：

- OCI を使用したサーバーへの接続、SQL 文の処理およびハンドルの割当ての詳細は、[第 2 章「OCI プログラミングの基本」](#) および [第 15 章「OCI リレーショナル関数」](#) にある OCI リレーショナル関数の説明を参照してください。
- OTT の詳細は、10-8 ページの「[C アプリケーションでのオブジェクトの表現](#)」および [第 14 章「Object Type Translator \(OTT\)」](#) を参照してください。

永続オブジェクト、一時オブジェクトおよび値

Oracle の型のインスタンスは、その存続期間によって、永続オブジェクトと一時オブジェクトに分類されます。永続オブジェクトのインスタンスは、オブジェクト識別子による参照が可能かどうかによって、スタンドアロン・オブジェクトと埋込みオブジェクトに分割されます。

注意： このマニュアルでは、オブジェクトとインスタンスは同じ意味の用語です。

関連項目： オブジェクトの詳細は、『Oracle9i アプリケーション開発者ガイド - オブジェクト・リレーショナル機能』を参照してください。

永続オブジェクト

永続オブジェクトは Oracle データベースに格納されているオブジェクトです。永続オブジェクトは、OCI アプリケーションによってオブジェクト・キャッシュにフェッチされ、変更されます。永続オブジェクトは、それにアクセスしているアプリケーションの存続期間が過ぎても存続できます。作成された永続オブジェクトは、明示的に削除されるまでデータベースに残留します。永続オブジェクトには次の 2 種類があります。

- スタンドアロン・インスタンス。オブジェクト表の行に格納され、それぞれが他と重複しないオブジェクト識別子を持っています。OCI アプリケーションでは、スタンドアロン・インスタンスへの REF を取り出してオブジェクトを確保し、確保したオブジェクトからその他の関連オブジェクトにナビゲートできます。スタンドアロン・オブジェクトは、参照可能オブジェクトとも呼ばれます。

参照可能オブジェクトの選択 (SELECT) も可能で、その場合は REF をフェッチするか代わりに、オブジェクトを値によってフェッチします。

- 埋込みインスタンスは、オブジェクト表の行として格納されません。埋込みインスタンスは、他の構造体の中に埋め込まれます。埋込みオブジェクトの例は、別のオブジェクトの属性であるオブジェクトや、データベースの表のオブジェクト列に存在するインスタンスなどです。埋込みインスタンスにはオブジェクト識別子がないため、OCI アプリケーションでは、埋込みインスタンスへの REF を取得できません。

埋込みオブジェクトは、参照不可能なオブジェクトまたは値インスタンスとも呼ばれます。埋込みオブジェクトは値と呼ばれることもありますが、それがスカラー・データ値と混同して使用されることはありません。どちらの意味であるかは文脈で判断できます。

次の SQL の例では、この 2 種類の永続オブジェクトの違いを説明します。

例 1 スタンドアロン・オブジェクト

```
CREATE TYPE person_t AS OBJECT
  (name      varchar2(30),
   age       number(3));
CREATE TABLE person_tab OF person_t;
```

オブジェクト表 `person_tab` に格納されるオブジェクトは、スタンドアロン・インスタンスです。このオブジェクトはオブジェクト識別子を持ち、参照可能です。したがって、OCI アプリケーションで確保できます。

例 2 埋込みオブジェクト

```
CREATE TABLE department
  (deptno    number,
   deptname  varchar2(30),
   manager   person_t);
```

`department` 表の `manager` 列に格納されるオブジェクトは、埋込みオブジェクトです。このオブジェクトはオブジェクト識別子を持たず、参照不可能です。つまり、OCI アプリケーションで確保することはできないため、確保を解除する必要もありません。このオブジェクトは常に、オブジェクト・キャッシュ内に値によって取り込まれます。

一時オブジェクト

一時オブジェクトはオブジェクト型のインスタンスです。この一時オブジェクトは、オブジェクト識別子を持つ場合があります。一時オブジェクトは、アプリケーションの存続期間を超えて存続できません。また、一時オブジェクトはいつでもアプリケーションで削除できます。

アプリケーションでは、`OCIObjectNew()` 関数を使用して一時オブジェクトを作成し、計算のための一時的な値を格納することがよくあります。一時オブジェクトを永続オブジェクトに変換することはできません。オブジェクトのロールは、インスタンス化されたときに決定されます。

関連項目： `OCIObjectNew()` の使用方法については、10-32 ページの「[オブジェクトの作成](#)」を参照してください。

値

このマニュアルでは、値は次のいずれかを意味します。

- データベースの表の非オブジェクト列に格納されるスカラー値。OCI アプリケーションでは、SQL 文を発行してデータベースから値をフェッチできます。
- 埋込みまたは参照不可能オブジェクト。

どちらを指しているのかは文脈で判断できます。

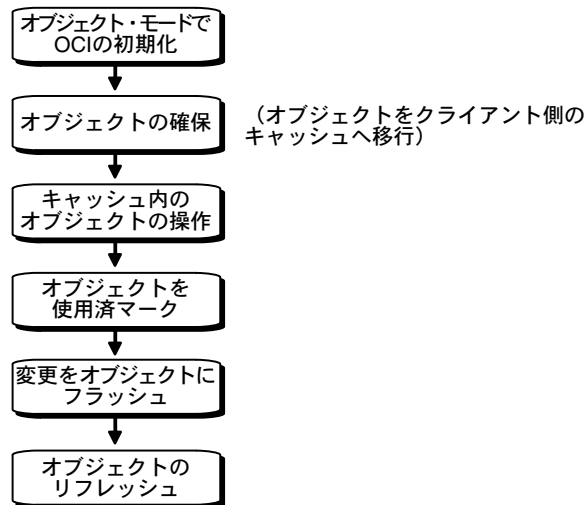
注意： 参照可能オブジェクトは、確保するかわりにオブジェクト・キャッシュ内で選択 (SELECT) することが可能です。その場合は、REF をフェッチせずに、オブジェクトを値によってフェッチします。

OCI オブジェクト・アプリケーションの開発

この項では、基本的な OCI オブジェクト・アプリケーションの開発に伴うステップを説明します。10-3 ページの「[基本的なオブジェクト・プログラム構造体](#)」で説明している各ステップについて、さらに詳しく説明します。

次の図は、アプリケーションがオブジェクトを操作する方法について、そのプログラム・ロジックを簡単に示しています。図を簡略にするために、必要なステップの一部が省略されています。この図にある各ステップについては、後の項で説明します。

図 10-1 基本的なオブジェクト操作のフロー



C アプリケーションでのオブジェクトの表現

OCI アプリケーションでオブジェクト型を操作するには、データベースにそのオブジェクト型が存在している必要があります。通常は、CREATE TYPE など、SQL の DDL 文で型を作成します。

型定義 DDL コマンドが Oracle サーバーで処理されると、型定義は型記述子オブジェクト (TDO) としてデータ・ディクショナリに格納されます。

アプリケーションでデータベースからオブジェクトのインスタンスを取り出す場合は、クライアント側のオブジェクト表現を使用する必要があります。C 言語のプログラムでは、オブジェクト型の表現は struct です。OCI オブジェクト・アプリケーションには、各オブジェクト型構造体に対応する NULL インジケータ構造体があります。

関連項目： アプリケーション・プログラマが、オブジェクトのキャッシュによって生成されるデフォルトの構造体以外のオブジェクト表現を利用する場合は、13-2 ページの「[オブジェクト・キャッシュおよびメモリー管理](#)」を参照してください。

Oracle には、データベースのオブジェクト型の C 構造体表現を生成する Object Type Translator (OTT) というユーティリティがあります。たとえば、次のように宣言した型がデータベースにあるとします。

```
CREATE TYPE emp_t AS OBJECT
( name      VARCHAR2(30),
  empno     NUMBER,
  deptno    NUMBER,
  hiredate  DATE,
  salary    NUMBER);
```

OTT では、次の C 構造体と、対応する NULL インジケータ構造体が生成されます。

```
struct emp_t
{
    OCIStrng    * name;
    OCINumber   empno;
    OCINumber   deptno;
    OCIDate     hiredate;
    OCINumber   salary;
};
typedef struct emp_t emp_t

struct emp_t_ind
{
    OCIIInd     _atomic;
    OCIIInd     name;
    OCIIInd     empno;
    OCIIInd     deptno;
```

```
OCIInd    hiredate;  
OCIInd    salary;  
};  
typedef struct emp_t_ind emp_t_ind;
```

この構造体宣言で使用している変数型は、OCI オブジェクト・コールで採用されている特殊な型です。OCI 関数のサブセットでは、このデータ型のデータを操作します。

関連項目： これらの関数については、この章の後半に説明がある他、[第 11 章「オブジェクト・リレーショナル・データ型」](#)でも詳しく述べられています。

これらの構造体の宣言は自動的に .h ファイルに書き込まれます。このファイルの名前は OTT 入力パラメータで判断されます。このヘッダー・ファイルをアプリケーションのコード・ファイルに組み込んで、オブジェクトにアクセスできます。

関連項目：

- OTT の詳細は、[第 14 章「Object Type Translator \(OTT\)」](#)を参照してください。
- NULL インジケータ構造体の使用方法の詳細は、10-30 ページの「[NULL かどうか](#)」を参照してください。

環境およびオブジェクト・キャッシュの初期化

OCI アプリケーションでオブジェクトにアクセスして操作する場合は、OCI アプリケーションの最初の OCI コールである `OCIEnvCreate()` コールの `mode` パラメータに、`OCI_OBJECT` の値を指定する必要があります。`mode` にこの値を指定して、アプリケーションでオブジェクトを操作することを OCI ライブラリに通知します。この通知には次の重要な効果があります。

- オブジェクト・ランタイム環境の確立
- オブジェクト・キャッシュの設定

オブジェクト・キャッシュ用のメモリーは、オブジェクトがキャッシュ内にロードされたときに、必要に応じて割り当てられます。

`OCIInitialize()` の `mode` パラメータに `OCI_OBJECT` を設定しないと、オブジェクト関連関数を使用しても結果はエラーになります。

クライアント側のオブジェクト・キャッシュは、プログラムのプロセス領域に割り当てられます。このキャッシュは、サーバーから取り出してアプリケーションで利用できるオブジェクトのメモリーです。

注意： OCI 環境をオブジェクト・モードで初期化すると、実際にオブジェクト・コールを使用するかどうかに関係なく、オブジェクト・キャッシュ用のメモリーを割り当てることになります。

関連項目： オブジェクト・キャッシュについては、この章の全体で説明されています。オブジェクト・キャッシュの詳しい説明は、[第 13 章「オブジェクトのキャッシュおよびナビゲーション」](#)を参照してください。

データベース接続の実行

アプリケーションでは、OCI 環境を正しく初期化した後、サーバーに接続できます。これは標準 OCI 接続コールを介して行われます。詳細は、2-20 ページの「[OCI プログラミング・ステップ](#)」を参照してください。標準 OCI 接続コールを使用すると、アプリケーションでオブジェクトにアクセスするための特別な配慮はありません。

OCI 環境ごとに、1 つのオブジェクト・キャッシュのみ割り当てられます。1 つの環境内で異なる接続を通じて取出しまたは作成されたオブジェクトは、すべて同じ物理的なオブジェクト・キャッシュを使用します。各接続には、それぞれ専用の論理的なオブジェクト・キャッシュがあります。

サーバーからのオブジェクト参照の取出し

アプリケーションでオブジェクトを操作するには、最初にサーバーから 1 つ以上のオブジェクトを取り出す必要があります。取出しは、1 つ以上のオブジェクトへの REF を戻す SQL 文を発行して行います。

注意： SQL 文によって、データベースから REF ではなく、埋込みオブジェクトをフェッチすることもできます。詳細は、10-15 ページの「[埋込みオブジェクトのフェッチ](#)」を参照してください。

次の例では、テキスト・ブロックがアプリケーションで宣言されています。このテキスト・ブロックには、データベース内の従業員のオブジェクト表 (emp_tab) から 1 つの従業員オブジェクトへの REF を取り出すように設計された SQL 文が格納されています。また、実行時に入力変数 (:emp_num) として渡される特定の従業員番号が指定されています。

```
text *selemp = (text *) "SELECT REF(e)
                        FROM emp_tab e
                        WHERE empno = :emp_num";
```


アプリケーションでは、[第2章「OCI プログラミングの基本」](#)で説明したリレーショナル SQL 文の操作と同じ方法でこの文を作成し、処理します。

- OCISstmtPrepare() を使用してアプリケーション要求を準備します。
- 適切なバインド・コールを使用してホスト入力変数をバインドします。
- 従業員オブジェクト参照を受け取る出力変数を宣言し、準備します。ここでは、10-8 ページの「[C アプリケーションでのオブジェクトの表現](#)」で宣言した参照に類似した従業員オブジェクト参照を使用します。

```
OCIRef *empl_ref = (OCIRef *) 0; /* reference to an employee object */
```

出力変数の定義時に、定義コールの *dt*y データ型パラメータを SQLT_REF (REF のデータ型定数) に設定します。

- OCISstmtExecute() で文を実行します。
- OCISstmtFetch() を使用して、結果の REF を empl_ref にフェッチします。

これで、オブジェクト参照を使用して、オブジェクトまたはデータベースからのオブジェクトにアクセスして操作できます。

関連項目：

- SQL 文の準備と実行に関する一般情報は、2-20 ページの「[OCI プログラミング・ステップ](#)」を参照してください。REF 変数のバインドと定義に関する具体的な情報は、5-10 ページの「[拡張バインド操作](#)」および 5-21 ページの「[拡張定義操作](#)」を参照してください。
- REF の取だしおよびその確保を示すコード例については、Oracle のインストールに付属のデモンストレーション・プログラムを参照してください。追加情報については、[付録 B「OCI デモ・プログラム」](#)を参照してください。

オブジェクトの確保

フェッチを行うステップが完了すると、アプリケーションでは、オブジェクトへの REF、つまりポインタを取得します。この時点ではまだ実際のオブジェクトは操作できません。オブジェクトを操作するには、そのオブジェクトの確保が必要です。オブジェクトの確保では、オブジェクト・インスタンスをオブジェクト・キャッシュにロードすることにより、必要に応じて、インスタンスの属性へのアクセスおよび変更や、1 オブジェクトからその他のオブジェクトへの参照追跡を可能にします。さらにアプリケーションでは、変更したオブジェクトをいつサーバーに書き込むか制御することもできます。

注意： この項では、一度に 1 つのオブジェクトを確保する単純な操作の例を示します。複合オブジェクト検索を介した複数オブジェクトの取だしの詳細は、10-20 ページの「[複合オブジェクト検索](#)」を参照してください。

アプリケーションでオブジェクトを確保するには、関数 `OCIObjectPin()` をコールします。この関数のパラメータによって、オブジェクトの PIN オプション、確保継続時間およびロック・オプションを指定できます。

次のコード例は、前の項で取り出した従業員参照の PIN オペレーションを示しています。

```
if (OCIObjectPin(env, err, &emp1_ref, (OCIComplexObject *) 0,
    OCI_PIN_ANY,
    OCI_DURATION_TRANS,
    OCI_LOCK_X, &emp1) != OCI_SUCCESS)
    process_error(err);
```

この例の `process_error()` はエラー処理関数を表しています。`OCIObjectPin()` 関数で、`OCI_SUCCESS` 以外の値が戻された場合、このエラー処理関数がコールされます。`OCIObjectPin()` 関数のパラメータは次のとおりです。

- `env` は OCI 環境ハンドルです。
- `err` は OCI エラー・ハンドルです。
- `emp1_ref` は、SQL によって取り出された参照です。
- `(OCIComplexObject *) 0` は、この PIN オペレーションで複合オブジェクト検索を行わないことを示します。
- `OCI_PIN_ANY` は PIN オプションです。詳細は、13-7 ページの「[オブジェクト・コピーの確保](#)」を参照してください。
- `OCI_DURATION_TRANS` は確保継続時間です。詳細は、13-15 ページの「[オブジェクト継続時間](#)」を参照してください。
- `OCI_LOCK_X` はロック・オプションです。詳細は、13-13 ページの「[更新するためのオブジェクトのロック](#)」を参照してください。
- `emp1` は出力パラメータで、確保したオブジェクトへのポインタを戻します。

これでオブジェクトが確保され、OCI アプリケーションでは、そのオブジェクトを変更できます。この単純な例では、オブジェクトは他のオブジェクトへの参照を含んでいません。

関連項目： あるインスタンスから別のインスタンスへのナビゲーション
例は、13-18 ページの「[単純なオブジェクト・ナビゲーション](#)」を参照してください。

配列確保

OCI アプリケーションでは、`OCIObjectArrayPin()` をコールすることによって、参照の配列に対してオブジェクトの配列を確保できます。参照は、異なる種類のオブジェクトを指し示すことができます。この関数によって、異なる表から異なる型のオブジェクトを 1 回のネットワーク・ラウンドトリップでフェッチできます。

オブジェクト属性の操作

これでオブジェクトが確保され、OCI アプリケーションでは、その属性を変更できます。OCI には、オブジェクト型構造体のデータ型を操作する一連の関数である、OCI データ型マッピング関数および操作関数があります。

注意： オブジェクト・キャッシュで確保したオブジェクトに対する変更内容は、オブジェクトのコピー（インスタンス）にのみ反映され、データベースにある元のオブジェクトには影響を与えません。アプリケーションで行った変更をデータベースに反映するには、変更内容をサーバーにフラッシュまたはコミットする必要があります。詳細は、10-14 ページの「[オブジェクトのマークおよび変更のフラッシュ](#)」を参照してください。

たとえば、従業員給与を増額できるよう前項で従業員オブジェクトを確保したとします。また、この会社では、入社して 180 日間未満の従業員には、年間昇給率が案分比例されるとします。

この例の場合、従業員の入社日にアクセスし、現行の日付の 180 日前より前か後をチェックする必要があります。その計算を基盤として、従業員の給与を \$5000（180 日より前）または \$3000（180 日より後）分増額します。次のページのコード例では、このプロセスを示しています。

データ型マッピング関数および操作関数で操作できるのは特定のいくつかのデータ型であり、**int** 型などのその他の型は、適切な OCI 型に変換しなければ計算で使用できないことに注意してください。

```
/* assume that sysdate has been fetched into sys_date, a string. */
/* emp1 and emp1_ref are the same as in previous sections. */
/* err is the OCI error handle. */
/* NOTE: error handling code is not included in this example. */

sb4 num_days;          /* the number of days between today and hiredate */
OCIDate curr_date;      /* holds the current date for calculations */
int raise;             /* holds the employee's raise amount before calculations */
OCINumber raise_num;    /* holds employee's raise for calculations */
OCINumber new_sal;      /* holds the employee's new salary */

/* convert date string to an OCIDate */
OCIDateFromText(err, (text *) sys_date, (ub4) strlen(sys_date), (text *)
                NULL, (ub1) 0, (text *) NULL, (ub4) 0, &curr_date);

/* get number of days between hire date and today */
OCIDateDaysBetween(err, &curr_date, &emp1->hiredate, &num_days);

/* calculate raise based on number of days since hiredate */
if num_days > 180
```

```
        raise = 5000
    else
        raise = 3000;

    /* convert raise value to an OCINumber */
    OCINumberFromInt(err, (dvoid *)&raise, (uword)sizeof(raise),
                     OCI_NUMBER_SIGNED, &raise_num);

    /* add raise amount to salary */
    OCINumberAdd(err, &raise_num, &empl->salary, &new_sal);
    OCINumberAssign(err, &new_sal, &empl->salary);
```

この例は、値をパラメータとして OCI データ型マッピング関数および操作関数に渡す前に、どのように OCI データ型（**OCIDate** や **OCINumber** など）に変換するかを示しています。

関連項目： OCI データ型、データ型マッピング関数および操作関数の詳細は、[第 11 章「オブジェクト・リレーショナル・データ型」](#)を参照してください。

オブジェクトのマークおよび変更のフラッシュ

前の項の例では、オブジェクト・インスタンスの属性を変更しました。ただし、この時点では変更はクライアント側のオブジェクト・キャッシュにのみ存在しています。変更をデータベースに確実に書き込むために、アプリケーションで特定のステップを実行する必要があります。

最初のステップでは、オブジェクトが変更されたことを示します。これは、**OCIObjectMarkUpdate()** 関数を使用して行います。この関数は、オブジェクトに使用済み（変更あり）のマークを設定します。

使用済みフラグが設定されているオブジェクトは、変更をデータベースに記録するために、サーバーにフラッシュする必要があります。次の 3 つの方法でこれを実行します。

- 単一の使用済みオブジェクトは、**OCIObjectFlush()** をコールしてフラッシュします。
- **OCICacheFlush()** を使用して、キャッシュ全体をフラッシュします。この場合、OCI はキャッシュで管理されている使用済みリストを横断して、使用済みオブジェクトをサーバーにフラッシュします。
- **OCITransCommit()** をコールして、トランザクションをコミットします。この方法でも、使用済みリストを横断してオブジェクトをサーバーにフラッシュします。

フラッシュ操作は、キャッシュの永続オブジェクトのみに作用します。一時オブジェクトはサーバーにフラッシュされません。

オブジェクトをサーバーにフラッシュすることで、データベースのトリガーをアクティブにすることができます。実際、アプリケーションでオブジェクトを明示的にフラッシュし、サーバー側のトリガーを起動することが必要な場合があります。

関連項目：

- OCITransCommit() の詳細は、8-2 ページの「[OCI でのトランザクションのサポート](#)」を参照してください。
- 一時オブジェクトと永続オブジェクトの詳細は、10-32 ページの「[オブジェクトの作成](#)」を参照してください。
- 使用済みなどのオブジェクトのメタ属性の照会とチェックの詳細は、10-16 ページの「[オブジェクトのメタ属性](#)」を参照してください。

埋込みオブジェクトのフェッチ

使用しているアプリケーションで、埋込みオブジェクト・インスタンス（オブジェクト表でなく、通常の表の列に格納されているオブジェクト）をフェッチする必要がある場合、10-10 ページの「[サーバーからのオブジェクト参照の取出し](#)」で説明する REF 検索機能は使用できません。埋込みインスタンスにはオブジェクト識別子がなく、埋込みインスタンスへの REF は取得できません。これは埋込みインスタンスがオブジェクト・ナビゲーションの基礎として機能しないことを意味します。しかし、アプリケーションで埋込みインスタンスをフェッチすることが必要な状況は数多くあります。

たとえば、address 型が作成されたとします。

```
CREATE TYPE address AS OBJECT
(
  street1      varchar2(50),
  street2      varchar2(50),
  city         varchar2(30),
  state        char(2),
  zip          number(5));
```

その型を、他の表で列のデータ型として使用することができます。

```
CREATE TABLE clients
(
  name      varchar2(40),
  addr      address);
```

OCI アプリケーションで次の SQL 文を発行します。

```
SELECT addr FROM clients
WHERE name='BEAR BYTE DATA MANAGEMENT'
```

この文は、clients 表の埋込みオブジェクト address を戻します。アプリケーションでは、このオブジェクトの属性値を別の処理で使用できます。

アプリケーションでは、第 2 章「[OCI プログラミングの基本](#)」で説明したリレーショナル SQL 文の操作と同じ方法でこの文を作成し、処理します。

- OCISstmtPrepare() を使用してアプリケーション要求を準備します。
- 適切なバインド・コールを使用して入力変数をバインドします。

- address インスタンスを受け取る出力変数を定義します。10-8 ページの「[C アプリケーションでのオブジェクトの表現](#)」で説明するとおり、OTT で生成されたオブジェクト型の C 構造体表現を使用します。

```
addr1      *address; /* variable of the address struct type */
```

出力変数を定義するときは、定義コールのデータ型パラメータ `dtty` を `SQLT_NTY`（名前付きデータ型のデータ型定数）に設定する必要があります。

- `OCIStmtExecute()` で文を実行します。
- `OCIStmtFetch()` を使用して結果のインスタンスを `addr1` にフェッチします。

次に、10-13 ページの「[オブジェクト属性の操作](#)」で説明しているように、インスタンスの属性にアクセスするか、あるいはインスタンスを別の SQL 文の入力パラメータとして渡すことができます。

注意： 埋込みインスタンスに対する変更内容は、SQL の UPDATE 文の実行によってのみ永続的にできます。

関連項目： SQL 文の準備と実行の詳細は、2-20 ページの「[OCI プログラミング・ステップ](#)」を参照してください。

オブジェクトのメタ属性

オブジェクトのメタ属性には、オブジェクトの状態に関する情報を、アプリケーションまたはオブジェクト・キャッシュに提供できるフラグの役割があります。たとえば、オブジェクトのメタ属性の 1 つとして、オブジェクトがサーバーにフラッシュ済みかどうかを示す属性があります。これは、アプリケーションでインスタンスの動作を制御するのに役立ちます。

永続オブジェクトのインスタンスと一時オブジェクトのインスタンスには、それぞれ異なるメタ属性があります。永続オブジェクトのメタ属性は、さらに永続メタ属性と一時メタ属性に分かれます。一時メタ属性は、インスタンスがメモリー内にあるときにのみ存在します。永続メタ属性は、サーバーに格納されているオブジェクトにも適用されます。

永続オブジェクトのメタ属性

次の表は、スタンドアロン永続オブジェクトのメタ属性を示しています。

永続オブジェクトのメタ属性	意味
既存	オブジェクトが存在しているか
NULL かどうか	インスタンスの NULL 情報
ロック	オブジェクトはロックされているか
使用済み	オブジェクトに使用済みのマークが設定されているか

一時メタ属性	意味
確保済み	オブジェクトは確保済みか
割当て時間	13-15 ページの「 オブジェクト継続時間 」を参照
確保継続時間	13-15 ページの「 オブジェクト継続時間 」を参照

注意： 埋込み永続オブジェクトの属性は、一時属性である「NULL かどうか」と「割当て時間」のみです。

OCI は、アプリケーションでオブジェクトのあらゆる属性のステータスをチェックできる `OCIObjectGetProperty()` 関数を提供します。この関数の構文は次のとおりです。

```
sword OCIObjectGetProperty ( OCIEnv          *envh,
                             OCIError       *errh,
                             CONST dvoid    *obj,
                             OCIObjectPropId propertyId,
                             dvoid         *property,
                             ub4           *size );
```

`propertyId` および `property` パラメータは、あらゆるプロパティまたは属性に関する情報の取出しに使用されます。

異なるプロパティ ID とそれに対応する `property` 引数の型を次に示します。

関連項目： 詳細は、17-26 ページの `OCIObjectGetProperty()` を参照してください。

OCI_OBJECTPROP_LIFETIME

このプロパティによって指定されたオブジェクトが永続オブジェクト、一時オブジェクトまたは、値のインスタンスかが識別されます。`property` 引数は **OCIObjectLifetime** 型の変数へのポインタにしてください。可能な値は、次のとおりです。

- OCI_OBJECT_PERSISTENT
- OCI_OBJECT_TRANSIENT
- OCI_OBJECT_VALUE

OCI_OBJECTPROP_SCHEMA

このプロパティによって、オブジェクトが存在する表のスキーマ名が戻されます。指定されたオブジェクトが一時インスタンスまたは値を指し示している場合は、エラーが戻されます。スキーマ名の保持に入力バッファのサイズが足りない場合は、エラーが戻され、エラー・メッセージで必要なサイズが表示されます。成功した場合は、`size` によって、戻されたスキーマ名のサイズがバイト単位で戻されます。`property` 引数は、**text** 型の配列にしてください。コール元は、`size` をバイト単位で配列のサイズに設定する必要があります。

OCI_OBJECTPROP_TABLE

このプロパティによってそのオブジェクトが存在する表名が戻されます。指定されたオブジェクトが一時インスタンスまたは値を指し示している場合は、エラーが戻されます。表名の保持に入力バッファが足りない場合は、エラーが戻され、エラー・メッセージで必要なサイズが表示されます。成功した場合は、`size` によって、戻された表名のサイズがバイト単位で戻されます。`property` 引数は、**text** 型の配列にしてください。コール元では、`size` を配列のサイズにバイト単位で設定する必要があります。

OCI_OBJECTPROP_PIN_DURATION

このプロパティによってオブジェクトの確保継続時間が戻されます。指定されたオブジェクトが値のインスタンスを指し示している場合は、エラーが戻されます。`property` 引数は **OCIDuration** 型の変数へのポインタにしてください。次の値が有効です。

- OCI_DURATION_SESSION
- OCI_DURATION_TRANS

関連項目： 継続時間については、13-15 ページの「[オブジェクト継続時間](#)」を参照してください。

OCI_OBJECTPROP_ALLOC_DURATION

このプロパティによって、オブジェクトの割当て時間が戻されます。*property* 引数は **OCIDuration** 型の変数へのポインタにしてください。次の値が有効です。

- OCI_DURATION_SESSION
- OCI_DURATION_TRANS

関連項目： 継続時間については、13-15 ページの「[オブジェクト継続時間](#)」を参照してください。

OCI_OBJECTPROP_LOCK

このプロパティによってオブジェクトのロック状況が戻されます。可能なロック状況は、**OCILockOpt** で計数値です。指定されたオブジェクトが一時インスタンスまたは値インスタンスを指し示している場合は、エラーが戻されます。*property* 引数は **OCILockOpt** 型の変数へのポインタにしてください。オブジェクトのロック状況は、**OCIObjectIsLocked()** 関数をコールしても取り出せます。

OCI_OBJECTPROP_MARKSTATUS

このプロパティによって使用済みステータスが戻され、オブジェクトが新規のオブジェクトであるか、更新されたオブジェクトであるか、削除されたオブジェクトであるかが示されます。指定されたオブジェクトが一時インスタンスまたは値インスタンスを指し示している場合は、エラーが戻されます。*property* 引数は、**OCIObjectMarkStatus** 型にしてください。次の値が有効です。

- OCI_OBJECT_NEW
- OCI_OBJECT_DELETED
- OCI_OBJECT_UPDATED

次のマクロを使用してマーク状態をテストすることが可能です。

- OCI_OBJECT_IS_UPDATED (flag)
- OCI_OBJECT_IS_DELETED (flag)
- OCI_OBJECT_IS_NEW (flag)
- OCI_OBJECT_IS_DIRTY (flag)

OCI_OBJECTPROP_VIEW

このプロパティによって指定されたオブジェクトがビュー・オブジェクトであるかどうか識別されます。戻されたプロパティ値が **TRUE** であれば、オブジェクトはビューであり、**TRUE** でない場合はオブジェクトはビューではありません。指定されたオブジェクトが一時インスタンスまたは値インスタンスを指し示している場合は、エラーが戻されます。この *property* 引数はブール型にする必要があります。

その他の属性関数

次の表に示すように、OCI には、アプリケーションでこれらの属性を直接的または間接的に設定あるいはチェックできるルーチンが提供されています。

メタ属性	設定に使用する関数	チェックに使用する関数
NULL かどうか	< なし >	OCIObjectGetInd()
存在	< なし >	OCIObjectExists()
ロック	OCIObjectLock()	OCIObjectIsLocked()
使用済み	OCIObjectMark()	OCIObjectIsDirty()

一時オブジェクトのメタ属性

一時オブジェクトには次の一時属性があります。一時オブジェクトに永続属性はありません。

一時メタ属性	意味
既存	オブジェクトが存在しているか
確保済み	オブジェクトはアプリケーションによってアクセスされるか
使用済み	オブジェクトに使用済みのマークが設定されているか
NULL かどうか	インスタンスの NULL 情報
割当て時間	13-15 ページの「 オブジェクト継続時間 」を参照
確保継続時間	13-15 ページの「 オブジェクト継続時間 」を参照

複合オブジェクト検索

ここまでの例は、一度に 1 つのインスタンスのみのフェッチまたは確保を行う例でした。その場合、オブジェクトを取り出すためのサーバー・ラウンドトリップが、PIN オペレーションごとに別個に発生します。

オブジェクト指向アプリケーションでは、相互に関連したオブジェクトの集合として問題をモデル化することがよくあります。これらのオブジェクトは、オブジェクトのグラフを形成します。アプリケーションでは、初期オブジェクト集合の一部からオブジェクトの処理を開始し、その一部のオブジェクトの参照を使用して残りのオブジェクトを横断します。クライアント / サーバー設定では、横断のたびにオブジェクトをフェッチするためのネットワーク・ラウンドトリップが発生し、非効率的です。

オブジェクトの処理時に、複合オブジェクト検索 (COR) を使用すると、アプリケーションのパフォーマンスが向上する場合があります。これはプリフェッチ・メカニズムであり、アプリケーションでは、このメカニズムを使用して、リンクされた一連のオブジェクトを1回の操作で取り出すための基準を指定します。

注意： 後述するように、プリフェッチ・オブジェクトがすべて確保されているとはかぎりません。それらはオブジェクト・キャッシュ内にフェッチされるため、後続の確保コールはローカル操作で行われます。

複合オブジェクトは、論理的に関連した複数のオブジェクトの集合です。この集合は、ルート・オブジェクトと、指定したネスト・レベルに基づいてそれぞれプリフェッチされた一連のオブジェクトで構成されています。ルート・オブジェクトは、明示的にフェッチまたは確保されます。ネスト・レベルは、複合オブジェクトのルート・オブジェクトから指定のプリフェッチ・オブジェクトまで最短で横断した場合の参照の数です。

アプリケーションで複合オブジェクトを指定するときは、複合オブジェクトの内容と境界を記述します。複合オブジェクトのフェッチは、環境のプリフェッチ制限、つまり、複数オブジェクトのプリフェッチに使用可能なオブジェクト・キャッシュ内のメモリー容量によって制約されます。

注意： COR によって機能性が追加されるわけではなく、パフォーマンスが改善されるのみです。オプションとして使用してください。

この説明の例として、次の型宣言があるとします。

```
CREATE TYPE customer(...);
CREATE TYPE line_item(...);
CREATE TYPE line_item_varray as VARRAY(100) of REF line_item;
CREATE TYPE purchase_order AS OBJECT
( po_number      NUMBER,
  cust           REF customer,
  related_orders REF purchase_order,
  line_items     line_item_varray);
```

purchase_order 型には、スカラー値 po_number、VARRAY の line_items および2つの参照が含まれています。1つは customer 型への参照、もう1つは purchase_order 型への参照であり、この型がリンク済みリストとして実装されることを示しています。

複合オブジェクトをフェッチする場合は、アプリケーションで次の指定をする必要があります。

1. 目的のルート・オブジェクトへの REF。

2. 複合オブジェクトの境界を指定するための1組以上の型およびネストの情報。型情報は、COR でたどる REF 属性を示し、ネスト・レベルはリンクをたどるレベル数を示します。

前述の発注情報オブジェクトの場合は、アプリケーションで次の指定をする必要があります。

1. ルート発注情報オブジェクトへの REF。
2. cust、related_orders または line_items についての1組以上の型およびネストの情報。

発注情報をプリフェッチするアプリケーションでは、発注書を発行した顧客の情報にアクセスすることも十分考えられます。単純なナビゲーションでは、2つのオブジェクトをフェッチするためにサーバーに2回アクセスすることが必要です。複合オブジェクト検索によって、発注情報を確保するときに顧客をプリフェッチできます。この場合、複合オブジェクトは、発注情報オブジェクトと参照する顧客のオブジェクトで構成されます。

前の例では、アプリケーションは REF に purchase_order を指定し、cust REF 属性をネスト・レベル1までたどるように指示します。

1. REF(PO object)
2. {(customer, 1)}

アプリケーションで、purchase_order オブジェクトと、オブジェクト・グラフに含まれるすべてのオブジェクトをプリフェッチする必要がある場合、cust および related_orders は両方とも、最大のネスト・レベルまでたどるようにアプリケーションで指示します。

1. REF(PO object)
2. {(customer, UB4MAXVAL), (purchase_order, UB4MAXVAL)}

UB4MAXVAL は、ルート・オブジェクトから参照を通して到達できる指定の型のオブジェクトすべてのプリフェッチを指定します。

アプリケーションで、PO および関連する明細項目すべてをフェッチする場合は、次の指定をします。

1. REF(PO object)
2. {(line_item, 1)}

アプリケーションでは、REF を通して特定の深さまで到達できる（推移閉包）オブジェクトすべてをフェッチすることも選択できます。そのためには、目的の深さをレベル・パラメータとして設定する必要があります。前述の2つの例では、それぞれアプリケーションで (PO object REF, UB4MAXVAL) と (PO object REF, 1) を指定して、必要なオブジェクトをプリフェッチすることもできます。この場合は多数のフェッチが余計に発生しますが、指定が非常に簡単であり、サーバー・ラウンドトリップが1回のみで済みます。

オブジェクトのプリフェッチ

複合オブジェクトを指定してフェッチした後に行う、複合オブジェクトに含まれるオブジェクトのフェッチでは、ネットワーク・ラウンドトリップは発生しません。これは、オブジェクトがすでにプリフェッチ済みであり、オブジェクト・キャッシュ内にあるためです。プリフェッチするオブジェクトが多すぎると、オブジェクト・キャッシュがあふれることがあるので注意が必要です。オブジェクト・キャッシュがあふれると、アプリケーションですでに確保している他のオブジェクトがキャッシュから押し出され、パフォーマンスの向上ではなく逆にパフォーマンスの低下につながる場合があります。

注意： プリフェッチ・オブジェクトすべてを保持するためのメモリーがキャッシュにない場合は、オブジェクトの一部がプリフェッチされない場合があります。アプリケーションでは、後でそれらのオブジェクトにアクセスするとき、ネットワーク・ラウンドトリップが発生します。

プリフェッチ・オブジェクトすべてに、SELECT 権限が必要です。複合オブジェクト内のオブジェクトに対する SELECT 権限がアプリケーションにない場合、そのオブジェクトはプリフェッチできません。

OCI での複合オブジェクト検索の実装

複合オブジェクト検索 (COR) によって、アプリケーションは、ルート・オブジェクトをフェッチするときに複合オブジェクトのプリフェッチができます。単純オブジェクトに対して使用するのと同じ `OCIObjectPin()` 関数に、複合オブジェクト指定を渡します。

アプリケーションでは、複合オブジェクト検索ハンドルを使用して、複合オブジェクト検索のパラメータを指定します。このハンドルは、**OCIComplexObject** 型であり、その他の OCI ハンドルと同じ方法で割り当てます。

複合オブジェクト検索ハンドルの内容は、複合オブジェクト検索記述子のリストです。複合オブジェクト検索記述子は、**OCIComplexObjectComp** 型であり、その他の OCI 記述子と同じ方法で割り当てます。

各 COR 記述子には、REF 型とネスト・レベルが含まれています。REF 型では、複合オブジェクトを組み立てるときにたどる参照の型を指定します。ネスト・レベルで、特定の型の参照をどこまでたどるかを指定します。ネスト・レベルの最大値には、整数値または定数 **UB4MAXVAL** を指定します。

アプリケーションでは、タイプ・パラメータとネスト・レベル・パラメータの COR 記述子を作成しないで、COR ハンドルにネスト・レベルを指定することもできます。この場合、すべての REF について、COR ハンドルに指定されたネスト・レベルまでたどります。また、要求があったときに個別にコレクション属性がフェッチされる (アウト・ライン) かどうかも、COR ハンドルで指定できます。デフォルトでは、コレクション属性は、それを含むオブジェクトとともにフェッチされます (インライン)。

アプリケーションで COR ハンドルの属性を設定するには、OCIAttrSet() を使用します。次の属性があります。

OCI_ATTR_COMPLEXOBJECT_LEVEL — ネスト・レベル

OCI_ATTR_COMPLEXOBJECT_COLL_OUTOFLINE — オブジェクト型のコレクション属性をアウト・ラインでフェッチ

アプリケーションで、OCIDescriptorAlloc() を使用して COR 記述子を割り当て、次の属性を設定できます。

OCI_ATTR_COMPLEXOBJECTCOMP_TYPE — REF 型

OCI_ATTR_COMPLEXOBJECTCOMP_LEVEL — 前述の型の参照に使用するネスト・レベル

これらの属性を設定すると、アプリケーションでは OCIParamSet() をコールして記述子を複合オブジェクト検索ハンドルに設定します。ハンドルにはハンドルの記述子の数を指定する OCI_ATTR_PARAM_COUNT 属性があります。この属性は OCIAttrGet() で読み込みます。

ハンドルに記述子を挿入した後、ハンドルを OCIObjectPin() コールに渡してルート・オブジェクトを確保し、複合オブジェクトの残りのオブジェクトのプリフェッチを行います。

複合オブジェクト検索ハンドルと記述子は、必要がなくなったとき、明示的に解放する必要があります。

関連項目： ハンドルと記述子の詳細は、2-5 ページの「[ハンドル](#)」および 2-15 ページの「[記述子](#)」を参照してください。

COR プリフェッチ

アプリケーションでは、ルート・オブジェクトのフェッチ時に複合オブジェクトを指定します。プリフェッチ・オブジェクトは、指定のルート・オブジェクトを基点とするオブジェクトのグラフに対して、幅優先横断を実行することによって取得されます。横断は、必要なオブジェクトがすべてプリフェッチされたとき、またはプリフェッチ・オブジェクトの合計サイズがプリフェッチ制限を超えたときに停止します。

COR インタフェース

複合オブジェクトをフェッチするためのインタフェースは、OCI 確保インタフェースです。アプリケーションは、初期化済みの COR ハンドルを `OCIObjectPin()` に（またはハンドルの配列を `OCIObjectArrayPin()` に）渡して、ルート・オブジェクト、および COR ハンドルで指定したプリフェッチ・オブジェクトをフェッチすることができます。

```

sword OCIObjectPin ( OCIEEnv          *env,
                    OCIError          *err,
                    OCIRef             *object_ref,
                    OCIComplexObject   *corhdl,
                    OCIPinOpt          pin_option,
                    OCIDuration        pin_duration,
                    OCILockOpt         lock_option,
                    dvoid              **object );

sword OCIObjectArrayPin ( OCIEEnv          *env,
                        OCIError          *err,
                        OCIRef             **ref_array,
                        ub4                array_size,
                        OCIComplexObject   **cor_array,
                        ub4                cor_array_size,
                        OCIPinOpt          pin_option,
                        OCIDuration        pin_duration,
                        OCILockOpt         lock,
                        dvoid              **obj_array,
                        ub4                *pos );

```

COR の使用時には、次の点について留意してください。

1. NULL の COR ハンドル引数は、デフォルトでルート・オブジェクトのみを確保します。
2. ルート・オブジェクト型でネスト・レベル 0 の COR ハンドルは、ルート・オブジェクトのみをフェッチします。つまり、NULL の COR ハンドルに相当します。
3. ロック・オプションは、ルート・オブジェクトのみに適用されます。

注意： プリフェッチ・オブジェクトにロック・オプションを指定するために、アプリケーションでは配列インタフェース (OCIObjectArrayPin()) を使用して、複合オブジェクト内のオブジェクトすべてにアクセスし、REF の配列を作成し、別のラウンドトリップで複合オブジェクト全体をロックできます。

COR の例

次の例では、複合オブジェクト検索を行うためにアプリケーション・プログラムを修正する方法を詳しく説明します。

発注情報とそれに関連した明細項目を表示するアプリケーションを考えてみます。太字のコードでこの処理を実行します。残りのコードでは、プリフェッチの複合オブジェクト検索を使用するため、アプリケーションのパフォーマンスが向上します。

```
OCIEnv *envhp;
OCIError *errhp;
OCIRef *liref;
OCIRef *poref;
OCIIter *itr;
boolean eoc;
purchase_order *po = (purchase_order *)0;
line_item *li = (line_item *)0;
OCISvcCtx *svchp;
OCIComplexObject *corhp;
OCIComplexObjectComp *cordp;
OCIType *litdo;
ub4 level = 0;

/* get COR Handle */
OCIHandleAlloc((dvoid *) envhp, (dvoid **) &corhp, (ub4)
    OCI_HTYPE_COMPLEXOBJECT, 0, (dvoid **)0);

/* get COR descriptor for type line_item */
OCIDescriptorAlloc((dvoid *) envhp, (dvoid **) &cordp, (ub4)
    OCI_DTYPE_COMPLEXOBJECTCOMP, 0, (dvoid **) 0);

/* get type of line_item to set in COR descriptor */
OCITypeByName(envhp, errhp, svchp, (const text *) 0, (ub4) 0,
    const text *) "LINE_ITEM", (ub4) strlen((const char *)
    "LINE_ITEM"), OCI_DURATION_SESSION, &litdo);

/* set line_item type in COR descriptor */
OCIAttrSet((dvoid *) cordp, (ub4) OCI_DTYPE_COMPLEXOBJECTCOMP,
    dvoid *) litdo, (ub4) sizeof(dvoid *), (ub4)
    OCI_ATTR_COMPLEXOBJECTCOMP_TYPE, (OCIError *) errhp);
```



```
level = 1;

/* set depth level for line_item_varray in COR descriptor */
OCIAttrSet( (dvoid *) cordp, (ub4) OCI_DTYPE_COMPLEXOBJECTCOMP,
            (dvoid *) &level, (ub4) sizeof(ub4), (ub4)
            OCI_ATTR_COMPLEXOBJECTCOMP_TYPE_LEVEL, (OCIError *) errhp);

/* put COR descriptor in COR handle */
OCIParamSet(corhp, OCI_HTYPE_COMPLEXOBJECT, &errhp, cordp,
            OCI_DTYPE_COMPLEXOBJECTCOMP, 1);

/* pin the purchase order */
OCIObjectPin(envhp, errhp, poref, corhp, OCI_PIN_LATEST,
            OCI_REFRESH_LOADED, OCI_DURATION_SESSION,
            OCI_LOCK_NONE, (ub2) 1, (dvoid **)&po);

/* free COR descriptor and COR handle */
OCIDescriptorFree((dvoid *) cordp, (ub4) OCI_DTYPE_COMPLEXOBJECTCOMP);
OCIHandleFree((dvoid *) corhp, (ub4) OCI_HTYPE_COMPLEXOBJECT);

/* iterate and print line items for this purchase order */
OCIIterCreate(envhp, errhp, po.line_items, &itr);

/* get first line item */
OCIIterNext(envhp, errhp, itr, &liref, (dvoid **)0, &eoc);
while (!eoc)          /* not end of collection */
{
    /* pin line item */
    OCIObjectPin(envhp, errhp, liref, (dvoid *)0, OCI_PIN_RECENT,
                OCI_REFRESH_LOADED, OCI_DURATION_SESSION,
                OCI_LOCK_NONE, (ub2) 1, (dvoid **)&li);
    display_line_item(li);

    /* get next line item */
    OCIIterNext(envhp, errhp, itr, &liref, (dvoid **)0, &eoc);
}
```

OCI と SQL のオブジェクトへのアクセス

アプリケーションで複数オブジェクト（オブジェクト参照によって相互に関連）のグラフを操作する必要がある場合は、オブジェクトへのアクセスに、SQL インタフェースよりも OCI インタフェースを使用する方が効率的です。SQL インタフェースを使用してオブジェクトのグラフを取り出すには、複数の SELECT 文の実行、つまり、複数のネットワーク・ラウンドトリップが必要になる場合があります。OCI によって提供される複合オブジェクト検索機能を使用すると、アプリケーションは 1 つの OCIObjectPin() コール内で複数のオブジェクトのグラフを取り出せます。

アプリケーションでオブジェクトのグラフを取り出し、それをユーザーとの対話を基に変更して、その変更をデータベース内に持続させる場合の更新について考えてみます。SQL インタフェースを使用する場合、アプリケーションではオブジェクトのグラフの更新に複数の UPDATE 文を実行する必要があります。新しいオブジェクトの作成と既存のオブジェクトの削除が変更に含まれている場合は、対応する INSERT 文および DELETE 文も実行する必要があります。さらに、アプリケーションでは、表名の変更状況など、INSERT 文、UPDATE 文または DELETE 文の実行に必要な、さらに多くの情報を記録する必要があります。

OCI の OCICacheFlush() 関数を使用すると、アプリケーションですべての変更（オブジェクトの挿入、削除および更新）を単一の操作でフラッシュできます。OCI ではすべてが記録されるので、アプリケーション側のコーディングが少なく済みます。このように、オブジェクトのグラフの操作に OCI を使用すると、効率的なだけでなく使用しやすいインタフェースが提供されます。

たとえば、アプリケーションで REF を使用してオブジェクトをフェッチする必要がある場合を考えてみます。OCI の場合、このフェッチは、OCIObjectPin() コールを使用してそのオブジェクトを確保することで、実行されます。SQL インタフェースの場合、このフェッチは、SELECT 文（たとえば、SELECT Deref(ref) from tbl;）の REF を参照解除することで実行されます。同じ REF（同じオブジェクトへの参照）が 1 つのトランザクション内で複数回参照解除されるような状況について考えてみます。OCI_PIN_RECENT オプションとともに OCIObjectPin() をコールすると、トランザクションに対してオブジェクトがサーバーから 1 度のみフェッチされ、同じ REF で確保が繰り返し行われる結果、すでにキャッシュ内に確保されているオブジェクトへのポインタが戻されます。SQL インタフェースの場合は、SELECT Deref... 文の実行のたびに、サーバーからオブジェクトがフェッチされるため、サーバーへのラウンドトリップが何度も行われ、同じオブジェクトが複数コピーされる結果となります。

最後に、アプリケーションで、参照不可能なオブジェクトをフェッチする必要がある場合について考えてみます。たとえば、次のとおりです。

```
CREATE TABLE department
(
  deptno number,
  deptname varchar2(30),
  manager employee_t
);
```

`manager` 列に格納された `employee_t` インスタンスは、参照不可能です。`manager` 列インスタンスをフェッチするには、SQL インタフェースのみが使用できます。ただし、`employee_t` に `REF` 属性がある場合は、OCI コールを使用して `REF` をナビゲートできません。

確保カウントおよび確保解除

オブジェクト・キャッシュ内の各オブジェクトには、確保カウントが関連付けられています。本来、確保カウントはオブジェクトに並行アクセスするコード・モジュールの数を表します。オブジェクトを初めてキャッシュに確保したときに、確保カウントが1に設定されます。複合オブジェクト検索でプリフェッチ・オブジェクトをオブジェクト・キャッシュに入れた時点では、オブジェクトの確保カウントは0（ゼロ）です。

すでに確保されているオブジェクトを確保することができます。そうすると、確保カウントが1増えます。プロセスでオブジェクトの使用が終了したときは、`OCIObjectUnpin()` を使用して確保解除を行う必要があります。このコールによって確保カウントが1減ります。

オブジェクトの確保カウントが0（ゼロ）になった場合、必要があればオブジェクトをキャッシュから出し、オブジェクトが占めているメモリー領域を解放できます。

オブジェクトの確保カウントは、`OCIObjectPinCountReset()` をコールして、明示的に0（ゼロ）に設定できます。

アプリケーションでは `OCICacheUnpin()` をコールして、キャッシュ内にある特定の接続に関連した全オブジェクトを確保解除できます。

関連項目：

- 確保カウントが0（ゼロ）のオブジェクトがキャッシュから削除される場合の条件については、13-9 ページの「[オブジェクト・コピーの解放](#)」を参照してください。
- オブジェクトまたはキャッシュ全体の明示的なフラッシュについては、10-14 ページの「[オブジェクトのマークおよび変更のフラッシュ](#)」を参照してください。
- キャッシュからエージ・アウトされるオブジェクトについては、13-9 ページの「[オブジェクト・コピーの解放](#)」を参照してください。

NULL かどうか

データベース表で、行の中に値のない列がある場合、その列は NULL または NULL を含むと呼ばれます。オブジェクトには、次の 2 種類の NULL を適用できます。

- オブジェクトの属性はすべて、NULL 値を持つことができます。これはオブジェクトの属性の値が不明であることを意味します。
- オブジェクトのインスタンスは、アトミック NULL にできます。これはオブジェクト全体の値が不明なことを意味します。

アトミック NULL とは、オブジェクトが存在しないということではありません。アトミック NULL のインスタンスは、値が不明なだけで、存在しています。つまり、データを持たないオブジェクトとみなすことができます。

OCI でオブジェクトを操作する場合は、アプリケーションで使用する各オブジェクト型の NULL インジケータ構造体を定義できます。そのために必要な作業は、通常、OTT で生成した NULL インジケータ構造体を構造体宣言とともに組み込むことのみです。OTT の出力ヘッダー・ファイルを組み込むと、アプリケーションで NULL インジケータ構造体を使用できるようになります。

各型に対する NULL インジケータ構造体の内容は、アトミック NULL インジケータ (OCIInd 型) とインスタンスの各属性の NULL インジケータです。型にオブジェクト属性がある場合、NULL インジケータ構造体には、その属性の NULL インジケータ構造体が含まれます。次の例は、それぞれ対応する NULL インジケータ構造体を持つ型の C 言語での表現方法を示しています。

```
struct address
{
    OCINumber    no;
    OCISString    *street;
    OCISString    *state;
    OCISString    *zip;
};
typedef struct address address;

struct address_ind
{
    OCIInd        _atomic;
    OCIInd        no;
    OCIInd        street;
    OCIInd        state;
    OCIInd        zip;
};
typedef struct address_ind address_ind;
```

```

struct person
{
    OCIStrng      *fname;
    OCIStrng      *lname;
    OCINumber      age;
    OCIDate        birthday;
    OCIArray       *dependentsAge;
    OCITable       *prevAddr;
    OCIRaw         *comment1;
    OCILobLocator  *comment2;
    address        addr;
    OCIRef         *spouse;
};
typedef struct person person;

struct person_ind
{
    OCIInd         _atomic;
    OCIInd         fname;
    OCIInd         lname;
    OCIInd         age;
    OCIInd         birthday;
    OCIInd         dependentsAge;
    OCIInd         prevAddr;
    OCIInd         comment1;
    OCIInd         comment2;
    address_ind    addr;
    OCIInd         spouse;
};
typedef struct person_ind person_ind;

```

注意： `person_ind` の `dependentsAge` フィールドは、VARRAY (person の配列 `dependentsAge` フィールド) 全体がアトミック NULL かどうかを示します。 `dependentsAge` フィールドの個々の要素の NULL 情報は、`OCICollGetElem()` のコールの `elemind` パラメータを通して取り出せます。同様に、`person_ind` の `prevAddr` フィールドは、NESTED TABLE (person の `prevAddr` フィールド) 全体がアトミック NULL かどうかを示します。 `prevAddr` フィールドの個々の要素の NULL 情報は、`OCICollGetElem()` のコールの `elemind` パラメータを通して取り出せます。

オブジェクト型インスタンスの場合は、NULL インジケータ構造体の最初のフィールドがアトミック NULL インジケータで、残りのフィールドはレイアウトがオブジェクト型インスタンスの属性に似ている属性 NULL インジケータです。

アプリケーションでは、アトミック NULL インジケータの値をチェックすることで、インスタンスがアトミック NULL かどうかをテストできます。また、他のインスタンスをチェックすることで、次のコード例で示すように、アプリケーションでその属性の NULL 状態をテストできます。

```
person_ind *my_person_ind
if ( my_person_ind -> _atomic = OCI_IND_NULL)
{
    /* instance is atomically NULL */
}
if ( my_person_ind -> fname = OCI_IND_NULL)
{
    /* fname attribute is NULL */
}
```

前述の例では、アトミック NULL インジケータまたは属性の NULL インジケータの値が事前定義の値 `OCI_IND_NULL` と比較され、NULL 状態がテストされます。この比較では、事前定義済みの次の値が使用可能です。

- `OCI_IND_NOTNULL` — 値が NULL でないことを示します。
- `OCI_IND_NULL` — 値が NULL であることを示します。
- `OCI_IND_BADNULL` — 囲みオブジェクト（親オブジェクト）が NULL であることを示します。この値は PL/SQL で使用され、`INVALID_NULL` としても参照されます。たとえば、型インスタンスが NULL の場合、その属性は `INVALID_NULL` です。

17-37 ページの `OCIObjectGetInd()` 関数を使用して、記憶域の割当てを行い、オブジェクトの NULL インジケータ構造体を取り出します。

関連項目： OTT で生成する NULL インジケータ構造体の詳細は、[第 14 章「Object Type Translator \(OTT\)」](#)を参照してください。

オブジェクトの作成

OCI アプリケーションでは、`OCIObjectNew()` を使用してあらゆるオブジェクトを作成できます。アプリケーションで永続オブジェクトを作成するには、新規オブジェクトを挿入するオブジェクト表を指定してください。この値は、`OCIObjectPinTable()` をコールして取り出し、`table` パラメータで渡します。一時オブジェクトを作成するには、作成するオブジェクトの型に対する型記述子オブジェクト（`OCIDescribeAny()` をコールして取出し）のみを渡す必要があります。

また、`typecode` パラメータに適切な値を渡し、`OCIObjectNew()` を使用して、スカラーのインスタンス（REF、LOB、文字列、ロー、数値、日付など）およびコレクション（VARRAY や NESTED TABLE など）を作成できます。

新しいオブジェクトの属性値

デフォルトでは、新規作成されたオブジェクトのすべての属性に NULL 値があります。属性データを初期化した後で、対応する各属性の NULL ステータスを NULL 以外に変更する必要があります。

オブジェクトの作成時に、属性を NULL 以外の値に設定できます。これは、OCIAttrSet() を使用して環境ハンドルの OCI_OBJECT_NEWNOTNULL 属性を TRUE に設定することにより実行できます。その後、この属性を FALSE に設定することにより、このモードを無効にできます。

OCI_OBJECT_NEWNOTNULL を TRUE に設定した場合は、OCIObjectNew() によって、NULL でない値のオブジェクトが作成されます。オブジェクトの属性には、次の表に示したデフォルト値があり、対応する NULL インジケータは、NOT NULL に設定されています。

表 10-1 新しいオブジェクトの属性値

属性の型	デフォルト値
REF	オブジェクトに REF 属性がある場合は、オブジェクトをフラッシュする前にこの属性を有効な REF に設定する必要があります。そうしないとエラーが戻されます。
DATE	Oracle で扱える最古の日付。01-JAN-4712 BCE (ユリウス日の第 1 日) の午前 0 時。
ANSI DATE	Oracle で扱える最古の日付。01-JAN-4712 BCE (ユリウス日の第 1 日)。
TIMESTAMP	Oracle で扱える最古の日時。01-JAN-4712 BCE (ユリウス日の第 1 日) の午前 0 時。
TIMESTAMP WITH TIME ZONE	Oracle で扱える最古の日時。UTC (0:0) タイム・ゾーンで、01-JAN-4712 BCE (ユリウス日の第 1 日) の午前 0 時。
TIMESTAMP WITH LOCAL TIME ZONE	Oracle で扱える最古の日時。UTC (0:0) タイム・ゾーンで、01-JAN-4712 BCE (ユリウス日の第 1 日) の午前 0 時。
INTERVAL YEAR TO MONTH	INTERVAL '0-0' YEAR TO MONTH
INTERVAL DAY TO SECOND	INTERVAL '0 0:0:0' DAY TO SECOND
FLOAT	0
NUMBER	0
DECIMAL	0
RAW	長さが 0 に設定されたロー・データ。 注意: RAW 属性のデフォルト値は、NULL の RAW 属性のデフォルト値と同じです。

表 10-1 新しいオブジェクトの属性値（続き）

属性の型	デフォルト値
VARCHAR2、 NVARCHAR2	長さが 0 で最初の文字が NULL に設定された OCIStrng。デフォルト値は、ヌル文字列属性のデフォルト値と同じです。
CHAR、NCHAR	長さが 0 で最初の文字が NULL に設定された OCIStrng。デフォルト値は、ヌル文字列属性のデフォルト値と同じです。
VARCHAR	長さが 0 で最初の文字が NULL に設定された OCIStrng。デフォルト値は、ヌル文字列属性のデフォルト値と同じです。
VARRAY	要素が 0 のコレクション。
NESTED TABLE	要素が 0 の表。
CLOB、NCLOB	空の CLOB。
BLOB	空の BLOB。
BFILE	ディレクトリ別名およびファイル名を設定することによって、BFILE を有効な値に初期化する必要があります。

オブジェクトの解放およびコピー

OCIObjectFree() を使用して、OCIObjectNew() で割り当てたメモリーを解放できます。オブジェクト・インスタンスには、追加メモリー（2 次メモリー・チャンク）へのポインタである属性を設定できます。

関連項目： 詳細は、13-17 ページの「[インスタンスのメモリー・レイアウト](#)」を参照してください。

オブジェクトの解放とは、関連する NULL インジケータ構造体や 2 次メモリー・チャンクなど、オブジェクトに割り当てたすべてのメモリーの割当てを解除することです。2 次メモリー・チャンクを明示的に解放したり、ポインタの割当てを削除することはできません。これは、メモリー・リークやメモリーの破損が発生する可能性があるためです。このプロセスによって、一時オブジェクトが、存続期間内に削除されます（永続オブジェクトは削除されません）。アプリケーションでは、永続オブジェクトを削除するには、OCIObjectMarkDelete() を使用します。

アプリケーションでは、OCIObjectCopy() を使用して、1 つのインスタンスを同じ型の別のインスタンスにコピーできます。

関連項目： これらの関数の詳細は、第 17 章「[OCI のナビゲーションル関数と型関数](#)」の説明を参照してください。

オブジェクト参照と型参照

アプリケーションでは、OCI のオブジェクト拡張機能を利用して、オブジェクトの内容に対してポインタや参照による柔軟なアクセスができます。OCI では、ポインタが示すオブジェクトに参照を戻す `OCIObjectGetObjectRef()` 関数を提供しています。

また、アプリケーションで、オブジェクトの型情報へのアクセスを行う場合は、OCI では `OCIObjectGetProperty()` 関数で、ポインタが示すオブジェクトの型記述子オブジェクト (TDO) に参照を戻します。

オブジェクト・ビューまたはユーザー定義 OID に基づいたオブジェクトの作成

アプリケーションで、`OCIObjectNew()` コールを使用して、オブジェクト・ビューまたはユーザー定義 OID に基づいたオブジェクトを作成できます。`OCIObjectNew()` が、オブジェクト・ビューまたはユーザー定義 OID を使用した表へのハンドルを受け取った場合、このコールによって戻される参照は擬似参照になります。この擬似参照は、他のオブジェクトには保存できませんが、`OCIObjectGetObjectRef()` を使用して主キーを基にした参照が取得できるように、オブジェクトの属性に指定することはできます。

この処理には、次のステップが必要です。

1. 新しいオブジェクトの基になるオブジェクト・ビューまたはオブジェクト表を確保します。
2. ステップ 1 の PIN オペレーションで取得した表またはビューへのハンドルを渡して、`OCIObjectNew()` を使用して新しいオブジェクトを作成します。
3. オブジェクトに対し、必要な値を指定します。指定する値には、オブジェクト表またはオブジェクト・ビューのユーザー定義 OID を構成する属性も含まれます。
4. 必要に応じて、`OCIObjectGetObjectRef()` を使用して主キーを基にしたオブジェクトへの参照を取得します。さらにオブジェクトを作成する場合は、ステップ 2 に戻ります。
5. 新しく作成されたオブジェクトをサーバーにフラッシュします。

次のコード例では、このプロセスを実装して、`scott` スキーマ内の `emp_view` オブジェクト・ビューに対して新しいオブジェクトを作成する方法を示します。

```
void object_view_new ()
{
    dvoid      *table;
    OCIRef     *pkref;
    dvoid      *object;
    ....
    /* Set up the service context, error handle and so on. */
    ...
    /* Pin the object view */
```

```
OCIObjectPinTable(envp,errorp,svctx, "scott", strlen("scott"), "emp_view",
    strlen("emp_view"),(dvoid *) 0, OCI_DURATION_SESSION, (dvoid **) &table);

/* Create a new object instance */
OCIObjectNew(envp, errorp, svctx, OCI_TYPECODE_OBJECT,(OCIType *)0, table,
OCI_DURATION_SESSION,FALSE,&object);

/* Populate the attributes of "object" */
OCIObjectSetAttr(...);
...
/* Allocate an object reference */
OCIObjectNew(envp, errorp, svctx, OCI_TYPECODE_REF, (OCIType *)0, (dvoid *)0,
    OCI_DURATION_SESSION,TRUE,&pkref);

/* Get the reference using OCIObjectGetObjectRef */
OCIObjectGetObjectRef(envp,errorp,object,pkref);
...
/* Flush new object(s) to server */
...
} /* end function */
```

オブジェクト・アプリケーションでのエラー処理

アプリケーションでオブジェクトを使用するかどうかにかかわらず、OCI アプリケーションでのエラー処理は同じです。

関連項目： 関数のリターン・コードとエラー・メッセージの詳細は、2-31 ページの「[エラー処理](#)」を参照してください。

型の継承

オブジェクトの型の継承は、C++ や Java での継承に類似している点が数多くあります。オブジェクト型は、既存のオブジェクト型のサブタイプとして作成できます。サブタイプは、元の型であるスーパータイプの属性とメソッド（メンバー関数とプロシージャ）をすべて継承します。単一の継承のみがサポートされます。つまり、1 つのオブジェクトが複数のスーパータイプを所有することはできません。サブタイプが継承した属性やメソッドには、新しい属性やメソッドを追加できます。サブタイプが継承した任意のメソッドをオーバーライド（実装を再定義）することもできます。サブタイプは、そのスーパータイプの拡張（つまり、スーパータイプの継承）といえます。

関連項目： この項目の詳細は、『Oracle9i アプリケーション開発者ガイド - オブジェクト・リレーショナル機能』を参照してください。

たとえば、`Person_t` 型は、サブタイプ `Student_t` とサブタイプ `Employee_t` を所有できます。さらに、`Student_t` も独自のサブタイプ `PartTimeStudent_t` を所有できます。サブタイプを所有するには、型宣言にフラグ `NOT FINAL` が必要です。デフォルトのフラグは `FINAL` で、この型はサブタイプを所有できないことを意味します。

この章でこれまで説明した型は、すべて `FINAL` です。リリース 1 (9.0.1) より前に開発されたアプリケーションの型は、すべて `FINAL` です。`FINAL` の型は、`NOT FINAL` に変更できます。サブタイプを所有しない `NOT FINAL` 型は、`FINAL` に変更できます。次の例では、`Person_t` が `NOT FINAL` として宣言されています。

```
CREATE TYPE Person_t AS OBJECT
( ssn NUMBER,
  name VARCAHR2(30),
  address VARCHAR2(100)) NOT FINAL;
```

サブタイプは、スーパータイプで宣言された属性とメソッドをすべて継承します。また、新しい属性やメソッドを宣言することもできます。その場合は、スーパータイプとは異なる名前を指定する必要があります。次のように、キーワード `UNDER` によって、スーパータイプが識別されます。

```
CREATE TYPE Student_t UNDER Person_t
( deptid NUMBER,
  major VARCHAR2(30)) NOT FINAL;
```

新しく宣言した属性 `deptid` および `major` は、サブタイプ `Student_t` に所属しています。たとえば、サブタイプ `Employee_t` を次のように宣言します。

```
CREATE TYPE Employee_t UNDER Person_t
( empid NUMBER,
  mgr VARCHAR2(30));
```

この結果 OTT によって生成される構造体については、次の項目を参照してください。

関連項目： 14-16 ページ「[OTT による型の継承のサポート](#)」

このサブタイプ `Student_t` は、`PartTimeStudent_t` などの独自のサブタイプを所有できます。

```
CREATE TYPE PartTimeStuden_t UNDER Student_t
( numhours NUMBER) ;
```

代入性

ポリモフィズムの利点の一部は、プロパティの持つ代入性によるものです。代入性によって、サブタイプの値をスーパータイプ用に記述された元のコードで使用できます。この場合、サブタイプに関する事前の知識は特に必要ありません。サブタイプの値は、メソッドの特化範囲内で異なるメカニズムが使用されている場合でも、周辺のコードに対応してスーパータイプの値と同じように動作します。

インスタンスの代入性とは、サブタイプのオブジェクト値を、スーパータイプで宣言されたコンテキスト内で使用できるようにする機能を指します。REF の代入性とは、サブタイプへの REF を、スーパータイプへの REF で宣言されたコンテキスト内で使用できるようにする機能を指します。

REF 型の属性は置換可能です。つまり、REF T として定義された属性は、T のインスタンスまたはそのサブタイプの任意のインスタンスへの REF を保持できます。

オブジェクト型の属性は置換可能です。つまり、(オブジェクト) 型 T として定義された属性は、T のインスタンスまたはそのサブタイプの任意のインスタンスを保持できます。

コレクションの要素の型は置換可能です。つまり、T 型の要素を持つコレクションを定義した場合、そのコレクションは、T 型のインスタンスとそのサブタイプの任意のインスタンスを保持できます。オブジェクト属性の代入性の例を次に示します。

```
CREATE TYPE Book_t AS OBJECT
( title VARCHAR2(30),
  author Person_t      /* substitutable */);
```

したがって、Book_t インスタンスは、次のように、タイトル文字列と Person_t（または、Person_t の任意のサブタイプ）インスタンスを指定することによって作成できます。

```
Book_t('My Oracle Experience',
      Employee_t(12345, 'Joe', 'SF', 1111, NULL))
```

NOT INSTANTIABLE の型とメソッド

型は NOT INSTANTIABLE として宣言できます。これは、その型に（デフォルトまたはユーザー定義の）コンストラクタがないことを意味します。したがって、この型のインスタンスは構成できません。通常、このような型は、インスタンス化可能なサブタイプの定義に使用されます。このプロパティの使用方法を次に示します。

```
CREATE TYPE Address_t AS OBJECT(...) NOT INSTANTIABLE NOT FINAL;
CREATE TYPE USAddress_t UNDER Address_t(...);
CREATE TYPE IntlAddress_t UNDER Address_t(...);
```

型のメソッドは、NOT INSTANTIABLE として宣言できます。メソッドを NOT INSTANTIABLE として宣言することは、型がそのメソッドを実装しないことを意味します。また、NOT INSTANTIABLE メソッドが含まれている型は、必ず NOT INSTANTIABLE として宣言する必要があります。たとえば、次のように宣言します。

```
CREATE TYPE T AS OBJECT
(
  x NUMBER,
  NOT INSTANTIABLE MEMBER FUNCTION func1() RETURN NUMBER
) NOT INSTANTIABLE;
```

NOT INSTANTIABLE 型のサブタイプは、スーパータイプの任意の NOT INSTANTIABLE メソッドをオーバーライドして、具体的な実装を行うことができます。NOT INSTANTIABLE メソッドが残っている場合、サブタイプも必ず NOT INSTANTIABLE として宣言する必要があります。

NOT INSTANTIABLE サブタイプは、インスタンス化可能なスーパータイプの下に定義できます。NOT INSTANTIABLE 型を FINAL として宣言することは無効なため、使用できません。

OCI での型の継承のサポート

次のコールは、型の継承をサポートしています。

OCIDescribeAny()

継承された型に固有の情報を提供する関数が拡張されました。継承された型のプロパティにはその他の属性が追加されています。たとえば、型のスーパータイプを取得できます。

関連項目： `OCIDescribeAny()` を使用して読み込める属性については、[表 6-7「型に所属する属性」](#) および [表 6-9「型メソッドに所属する属性」](#) を参照してください。

バインド関数および定義関数

OCI のバインド関数は、REF、インスタンスおよびコレクション要素の代入性をサポートしています（スーパータイプがある場合は、サブタイプのインスタンスを渡すことができます）。すべての型のチェックと変換は、サーバー側で実行されるため、OCI のバインド・インタフェースに対する変更はありません。

OCI の定義関数も、代入性をサポートしています（サブタイプのインスタンスをスーパータイプの保持を宣言した定義変数にフェッチできます）。ただし、この場合、サブタイプのインスタンスを保持するために、システムによるメモリーのサイズ変更が必要になることがあります。

注意： この場合、クライアント・プログラムでは、オブジェクト・キャッシュから割り当てられた（したがって、サイズ変更が可能な）オブジェクトを使用する必要があります。

値が多相になる可能性がある場合、クライアントでは、(スタック上に割り当てられた) 構造体を定義変数として使用しないでください。

関連項目： バインド処理と定義処理の詳細は、[第 11 章「オブジェクト・リレーショナル・データ型」](#) を参照してください。

OCIObjectGetTypeRef()

この関数は、入力オブジェクトの最も特定な型の TDO を戻します。この操作では、ユーザーに最も特定な型に対する権限がない場合、エラーが戻されます。

OCIObjectCopy()

ソースは、ターゲットの型に関しては、任意のサブタイプのインスタンスにできます。たとえば、Employee_t インスタンスを保持するソース・オブジェクトをターゲットの Person_t オブジェクトに割り当てることができます。オブジェクトのコピーでは、サブタイプの属性も含めてすべての属性がソースからターゲットにコピーされます。このコピーによって、ターゲット・オブジェクトの最も特定な型が Employee_t に変更されます。

TDO の引数は、ソース・オブジェクトの最も特定な型を参照します。

OCICollAssignElem()

入力要素は、宣言された型のサブタイプのインスタンスにできます。コレクションの型が Person_t の場合は、この関数を使用して、Employee_t インスタンスをコレクションの要素として割り当てることができます。

OCICollAppend()

入力要素は、宣言された型のサブタイプのインスタンスにできます。コレクションの型が、Person_t の場合は、この関数を使用して、コレクションに Employee_t インスタンスを追加できます。

OCICollGetElem()

戻されたコレクション要素は、宣言された型のサブタイプのインスタンスにできます。

OTT での型の継承のサポート

Object Type Translator (OTT) は、オブジェクトの型の継承をサポートします。このサポートは、継承される属性を '_super' というカプセル化された構造体内で最初に宣言し、その後に新しい属性を宣言することで実行されます。これを実行する理由は、C では型の継承がサポートされていないためです。

関連項目： 例および説明は、14-25 ページの「[OTT リファレンス](#)」を参照してください。

型の変更

型属性の追加、削除および変更がサポートされています。この概念は、型の変更と呼ばれています。型の変更は、次のマニュアルで説明しています。

関連項目：『Oracle9i アプリケーション開発者ガイド - オブジェクト・リレーショナル機能』

OCIDescribeAny() は、要求された型の最新バージョンの情報を戻します。ただし、これは、入力オブジェクトの型が OCI_OTYPE_NAME で、記述済みのオブジェクトの型が OCI_PTYPE_TYPE の場合、つまり、OCIDescribeAny() に入力する名前が型名である場合です。

型情報にアクセスするには、OCIDescribeAny() 以外に、次の関数を使用します。

関連項目： [OCITypeArrayByName\(\)](#) および [OCITypeByName\(\)](#)

型の変更のオブジェクト・キャッシュへの影響は、次の項目を参照してください。

関連項目： 13-23 ページ「[型の変更とオブジェクト・キャッシュ](#)」

オブジェクト・リレーショナル・データ型

バインドおよび定義の概念は、第2章「OCIプログラミングの基本」および第5章「バインドと定義」で紹介および説明されています。この章では、オブジェクト・アプリケーションを開発するときに必要な追加情報を提供します。この章は、次の項目で構成されています。

- オブジェクトに対する OCI 関数の概要
- Oracle データ型の C へのマッピング
- OCI での C データ型の操作
- 日付 (OCIDate)
- 日時および時間隔 (OCIDateTime、OCIInterval)
- 数値 (OCINumber)
- 固定長または可変長文字列 (OCIString)
- ロー (OCIRaw)
- コレクション (OCITable、OCIArray、OCIColl、OCIIter)
- マルチレベル・コレクション型
- REF (OCIRef)
- オブジェクト型情報の格納およびアクセス
- AnyType インタフェース、AnyData インタフェースおよび AnyDataSet インタフェース
- 名前付きデータ型のバインド
- 名前付きデータ型の定義
- Oracle C データ型のバインドおよび定義
- SQLT_NTY のバインドおよび定義の例

オブジェクトに対する OCI 関数の概要

OCI データ型のマッピング関数および操作関数を使用すると、事前定義済みの Oracle C データ型のインスタンスを操作できます。これらのデータ型は、Oracle のオブジェクト型など、ユーザー定義のデータ型の属性を表現するために使用します。

OCI の各関数グループは、特定のネーミング規則によって区別されます。たとえば、データ型のマッピング関数および操作関数は、関数名が OCI 接頭辞で始まり、その後にデータ型が続くため、簡単に認識できます（例：OCIDateFromText() および OCIRawSize()）。後述するように、名前はさらに特定の型のデータを操作する関数グループに分類できます。

また、これらの関数を操作する事前定義済みの Oracle C 型は、OCI の接頭辞で始まる名前でも区別できます（例：OCIDate や OCISString）。

データ型のマッピング関数および操作関数は、アプリケーションで、Oracle データベースに格納されたオブジェクト、または SQL 問合せで取り出されたオブジェクトの属性を操作、バインドまたは定義する必要がある場合に使用します。第 13 章「オブジェクトのキャッシュおよびナビゲーション」で説明したとおり、取り出したオブジェクトはクライアント側のオブジェクト・キャッシュに格納されます。

この章では、OCI データ型のマッピング関数および操作関数で操作できる各データ型の用途と構造について説明します。また、関数をグループ別にまとめ、使用可能な関数とその目的のリストを示します。

この章では、OCI アプリケーションのバインド操作と定義操作でこれらのデータ型を使用する方法も説明します。

これらの関数は、OCI アプリケーションをオブジェクト・モードで実行している間にのみ有効です。OCI のオブジェクト・モードでの初期化、オブジェクトにアクセスして操作する OCI アプリケーションの作成については、10-9 ページの「環境およびオブジェクト・キャッシュの初期化」を参照してください。

関連項目： オブジェクト型、属性およびコレクション・データ型の詳細は、『Oracle9i データベース概要』を参照してください。

Oracle データ型の C へのマッピング

Oracle には、表を作成し、ユーザー定義のデータ型（オブジェクト型を含む）を指定できる、一連の事前定義済みのデータ型があります。オブジェクト型は Oracle の機能を拡張するものであり、オブジェクト型を使用すると、処理対象のデータの型を正確にモデル化したデータ型を作成できます。そのため、プログラムは、より効率的で簡単なデータ・アクセスができます。

NCHAR および NVARCHAR2 はオブジェクトの属性として使用でき、C では **OCISString *** にマップされます。

データベースの表およびオブジェクト型は、Oracle が供給するデータ型に基づいています。データベースの表およびオブジェクト型は、SQL 文で作成し、VARCHAR2 や NUMBER などの特定の Oracle 内部データ型を使用して格納します。たとえば、次の SQL 文では、ユーザー定義の address データ型と、そのデータ型のインスタンスを格納するオブジェクト表を作成します。

```
CREATE TYPE address AS OBJECT
(street1    varchar2(50),
street2    varchar2(50),
city       varchar2(30),
state      char(2),
zip        number(5));
CREATE TABLE address_table OF address;
```

この新しい address 型を使用して、次のようなオブジェクト列を持つ標準表を作成したとします。

```
CREATE TABLE employees
(name          varchar2(30),
birthday      date,
home_addr     address);
```

OCI アプリケーションでは、SQL 文に対応付けられた、簡単なバインドおよび定義操作を使用して、情報を employees 表の name および birthday の列で操作できます。オブジェクトの属性として格納されている情報にアクセスするには、いくつかの追加ステップが必要です。

OCI アプリケーションでは、まずオブジェクトを C 言語書式で表現する手段が必要です。そのためには、Object Type Translator (OTT) を使用して、ユーザー定義型の C 構造体表現を生成します。生成した構造体の要素には、Oracle のデータ型を C 言語にマッピングしたデータ型が含まれます。

関連項目： オブジェクト属性の型として使用できる Oracle の型と、対応する C マッピングについては、表 14-1「オブジェクト型属性のオブジェクト・データ型マッピング」を参照してください。

その他の C 型として **OCIInd** があります。これは、オブジェクト型の属性に対応する NULL インジケータ情報を表現するために使用します。

関連項目： OTT の使用方法の詳細と例は、[第 14 章「Object Type Translator \(OTT\)」](#)を参照してください。

OCI 型マッピングの方法論

事前定義済みの Oracle 型についてマッピングを指定する場合、Oracle では独特の設計方針に従ってきました。カレント・システムには、次のメリットがあります。

- **OCINumber** のようなデータ型の実際の表現は、クライアントのアプリケーションでは不透明で、データ型は事前に定義済みの一連の関数によって操作されます。これによって、将来改善する際にも、ユーザー・コードを中断せずに内部表現を変更できます。
- オブジェクト指向パラダイムに従った実装がなされています。このパラダイムではクラスの実装の詳細は隠されており、見えるのは必要な操作のみです。
- この実装方法はプログラマにとって有利です。Oracle の数値で提供される Oracle 数値変数を、精度を落とすことなく操作する場合の C プログラムを考えてみます。これを Oracle リリース 7.x で行うには、`SELECT ...FROM DUAL` 文の発行が必要でした。以降のリリースでは、`OCINumber*`() 関数をコールするのみです。

OCI での C データ型の操作

OCI アプリケーションで、2 つの整数変数を合計して結果を第 3 の変数に格納するという非常に簡単なデータ操作を行うとします。

```
int    int_1, int_2, sum;
...
/* some initialization occurs */
...
sum = int_1 + int_2;
```

C 言語は、**整数型**のように単純な型の一連の事前定義済み操作を提供します。ただし、[表 14-1「オブジェクト型属性のオブジェクト・データ型マッピング」](#)にリストする C データ型は、単純な C 基本形ではありません。**OCIString** および **OCINumber** のような型は、実際は特定の Oracle 定義の内部構造体を持つ構造体です。単に、2 つの **OCINumber** を合計して、その値を第 3 の変数に格納することはできません。

次の内容は、有効ではありません。

```
OCINumber    num_1, num_2, sum;
...
/* some initialization occurs */
...
sum = num_1 + num_2;           /* NOT A VALID OPERATION */
```

このような新しいデータ型を操作するための関数が、OCI データ型のマッピング関数および操作関数です。たとえば、この例で **OCINumber** を加算するには、`OCINumberAdd()` 関数を使用します。

```
OCINumber    num_1, num_2, sum;
...
/* some initialization occurs */
...
OCINumberAdd(errhp, &num_1, &num_2, &sum): /* errhp is error handle */
```

OCI には、新規データ型をそれぞれ操作する関数があります。関数の名前から、その関数で操作するデータ型がわかります。最初の 3 文字「OCI」は、関数が OCI の一部であることを示します。関数名の次の部分は、関数で操作するデータ型を示します。次の表では、各種の関数の接頭辞、関数名の例および関数の操作対象のデータ型を示します。

関数の接頭辞	例	操作対象
OCIColl	<code>OCICollGetElem()</code>	OCIColl、OCIIter、OCITable、OCIArray
OCIDate	<code>OCIDateDaysBetween()</code>	OCIDate
OCIDateTime	<code>OCIDateTimeSubtract()</code>	OCIDate、OCIDateTime
OCIInter	<code>OCIInterToText()</code>	OCIInterval
OCIIter	<code>OCIIterInit()</code>	OCIIter
OCINumber	<code>OCINumberAdd()</code>	OCINumber
OCIRaw	<code>OCIRawResize()</code>	OCIRaw *
OCISRef	<code>OCISRefAssign()</code>	OCISRef *
OCISString	<code>OCISStringSize()</code>	OCISString *
OCITable	<code>OCITableLast()</code>	OCITable *

各データ型の構造については、そのデータ型を操作対象にする関数のリストとともに、この章で後述します。

Oracle 数値操作の精度

Oracle の数値の精度は、10 進数で 38 桁です。Oracle の数値操作は、次の例外を除いて、全桁で行います。

- 逆三角関数の精度は、10 進数で 28 桁です。
- 三角関数を含むその他の超越関数の精度は、およそ 10 進数で 37 桁です。
- 固有の浮動小数点型との間の変換では、適切な浮動小数点型の精度になり、10 進数で 38 桁を超えることはありません。

日付（OCIDate）

Oracle の日付書式は、不透明な C 構造体である **OCIDate** 型で C にマップされます。構造体の要素は日付の年、月、日、時間、分および秒を表します。適切な OCI 関数を使用すると、特定の要素を設定および取出しできます。

OCIDate データ型は、バインド・コールまたは定義コールで、外部型コード **SQLT_ODT** を使用して直接バインドまたは定義できます。

OCI の日付操作関数は、機能性ごとに編成された次の表にリストされています。表内の日付という用語は、指定がないかぎり、**OCIDate** 型の値を指します。

関連項目： すべての関数のプロトタイプと説明は、[第 18 章「OCI のデータ型マッピング関数および操作関数」](#)を参照してください。

日付変換関数

次の関数は、日付変換を実行します。

関数	用途
OCIDateToText ()	日付を文字列に変換します。
OCIDateFromText ()	テキスト文字列を日付に変換します。
OCIDateZoneToZone ()	日付をあるタイム・ゾーンから別のゾーンに変換します。

日付割当ておよび日付検索関数

次の関数は、日付要素の取出しと割当てを行います。

関数	用途
OCIDateAssign()	OCIDate を割り当てます。
OCIDateGetDate()	OCIDate の日付部分を取得します。
OCIDateSetDate()	OCIDate の日付部分を設定します。
OCIDateGetTime()	OCIDate の時間部分を取得します。
OCIDateSetTime()	OCIDate の時間部分を設定します。

日付算術および日付比較関数

次の関数は、日付算術および比較を実行します。

関数	用途
OCIDateAddDays()	日を加算します。
OCIDateAddMonths()	月を加算します。
OCIDateCompare()	日付を比較します。
OCIDateDaysBetween()	2つの日付間の日数を計算します。

日付情報アクセッサ関数

次の関数は、日付情報にアクセスします。

関数	用途
OCIDateLastDay()	月の最終日
OCIDateNextDay()	指定の日付後最初の日
OCIDateSysDate()	システム日付

日付の妥当性チェック関数

次の関数は、日付の妥当性を検査します。

関数	用途
OCIDateCheck()	指定の日付が妥当かどうかを検査します。

日付の例

次のコード例では、OCI コールを使用して、**OCIDate** 型の属性を操作する方法を示します。この例では、**OCIEnv** および **OCIError** は、[第 2 章「OCI プログラミングの基本」](#)の説明に従って初期化されているとします。確保については、[第 13 章「オブジェクトのキャッシュおよびナビゲーション」](#)を参照してください。

```
#define FMT "DAY, MONTH DD, YYYY"
#define LANG "American"
struct person
{
    OCIDate start_date;
};
typedef struct person person;

OCIError *err;
person *tim;
sword status; /* error status */
uword invalid;
OCIDate last_day, next_day;
text buf[100], last_day_buf[100], next_day_buf[100];
ub4 buflen = sizeof(buf);

/* Pin tim person object in the object cache. */
/* For this example, assume that
/* tim is pointing to the pinned object. */
/* set the start date of tim */
OCIDateSetTime(&tim->start_date,8,0,0);
OCIDateSetDate(&tim->start_date,1990,10,5)

/* check if the date is valid */
if (OCIDateCheck(err, &tim->start_date, &invalid) != OCI_SUCCESS)
/* error handling code */

if (invalid)
/* error handling code */
```



```

/* get the last day of start_date's month */
if (OCIDateLastDay(err, &tim->start_date, &last_day) != OCI_SUCCESS)
/* error handling code */

/* get date of next named day */
if (OCIDateNextDay(err, &tim->start_date, "Wednesday",      strlen("Wednesday"),
&next_day) != OCI_SUCCESS)
/* error handling code */
/* convert dates to strings and print the information out */
/* first convert the date itself*/
buflen = sizeof(buf);
if (OCIDateToText(err, &tim->start_date, FMT, sizeof(FMT)-1, LANG,
      sizeof(LANG)-1, &buflen, buf) != OCI_SUCCESS)
/* error handling code */

/* now the last day of the month */
buflen = sizeof(last_day_buf);
if (OCIDateToText(err, &last_day, FMT, sizeof(FMT)-1, LANG,      sizeof(LANG)-1,
&buflen, last_day_buf) != OCI_SUCCESS)
/* error handling code */

/* now the first Wednesday after this date */
buflen = sizeof(next_day_out);
if (OCIDateToText(err, &next_day, FMT, sizeof(FMT)-1, LANG,
      sizeof(LANG)-1, &buflen, next_day_buf) != OCI_SUCCESS)
/* error handling code */

/* print out the info */
printf("For: %s\n", buf);
printf("The last day of the month is: %s\n", last_day_buf);
printf("The next Wednesday is: %s\n", next_day_buf);

```

出力は次のようになります。

```

For: Monday, May 13, 1996
The last day of the month is: Friday, May 31
The next Wednesday is: Wednesday, May 15

```

日時および時間隔（OCIDateTime、OCIInterval）

OCIDateTime データ型は、Oracle のタイムとタイムスタンプのデータ型（TIME、TIME WITH TIME ZONE、TIMESTAMP、TIMESTAMP WITH TIME ZONE）および ANSI DATE データ型の表現に使用する不透明な構造体です。適切な OCI 関数を使用すると、これらの型（つまり、年、日、小数秒）でのデータの設定または取出しを実行できます。

OCIInterval データ型も不透明な構造体で、Oracle の時間隔データ型（INTERVAL YEAR TO MONTH、INTERVAL DAY TO SECOND）の表現に使用されます。

バインドまたは定義のコールで、次の外部型コードを使用すると、**OCIDateTime** と **OCIInterval** のデータをバインドおよび定義できます。

OCI データ型	データの型	バンド / 定義の外部型コード
OCIDateTime	ANSI DATE	SQLT_DATE
OCIDateTime	TIMESTAMP	SQLT_TIMESTAMP
OCIDateTime	TIMESTAMP WITH TIME ZONE	SQLT_TIMESTAMP_TZ
OCIDateTime	TIMESTAMP WITH LOCAL TIME ZONE	SQLT_TIMESTAMP_LTZ
OCIInterval	INTERVAL YEAR TO MONTH	SQLT_INTERVAL_YM
OCIInterval	INTERVAL DAY TO SECOND	SQLT_INTERVAL_DS

日時と時間隔のデータを操作する OCI 関数を次の表にリストします。これらの関数の詳細は、18-27 ページの [OCI の日付関数](#)、[日時関数および時間隔関数](#)を参照してください。

通常、**OCIDateTime** データを操作する関数は、**OCIDate** データにも有効です。

日時関数

次の関数は、**OCIDateTime** の値を操作します。この関数の中には、日時と時間隔の値に関する算術演算を行う関数もあります。また、特定の日時型に対してのみ動作する関数もあります。可能な型は、次のとおりです。

- SQLT_DATE – DATE
- SQLT_TIMESTAMP – TIMESTAMP
- SQLT_TIMESTAMP_TZ – TIMESTAMP WITH TIME ZONE
- SQLT_TIMESTAMP_LTZ – TIMESTAMP WITH LOCAL TIME ZONE

特定の関数に有効な入力型の詳細は、各関数の説明を参照してください。

関数	用途
OCIDateTimeAssign () 18-48 ページ	日時の割当てを実行します。
OCIDateTimeCheck () 18-49 ページ	指定の日付が有効かどうかをチェックします。
OCIDateTimeCompare () 18-51 ページ	2 つの日時の値を比較します。
OCIDateTimeConstruct () 18-52 ページ	日時の記述子を作成します。
OCIDateTimeConvert () 18-54 ページ	日時の型を 1 つの型から別の型に変換します。
OCIDateTimeFromArray () 18-55 ページ	配列のサイズ OCI_DT_ARRAYLEN を OCIDateTime 記述子に変換します。
OCIDateTimeFromText () 18-57 ページ	指定の書式に従って、指定された文字列を OCIDateTime 記述子の Oracle の日時型に変換します。
OCIDateTimeGetDate () 18-59 ページ	日時の値の日付（年、月、日）部分を取得します。
OCIDateTimeGetTime () 18-60 ページ	日時の値からタイム（時間、分、秒、小数秒）を取得します。
OCIDateTimeGetTimeZoneName () 18-62 ページ	日付の値からタイム・ゾーンの名前部分を取得します。
OCIDateTimeGetTimeZoneOffset () 18-63 ページ	日付の値からタイム・ゾーン（時間、分）部分を取得します。
OCIDateTimeIntervalAdd () 18-64 ページ	日時に時間隔を加算して、結果の日時を生成します。
OCIDateTimeIntervalSub () 18-65 ページ	日時から時間隔を減算して、その結果を日時に格納します。
OCIDateTimeSubtract () 18-66 ページ	2 つの日時を入力として受け入れ、その差異を時間隔に格納します。
OCIDateTimeSysTimeStamp () 18-67 ページ	現行のシステム日付と時刻を、タイム・ゾーンとともにタイムスタンプとして取得します。
OCIDateTimeToArray () 18-68 ページ	OCIDateTime 記述子を配列に変換します。
OCIDateTimeToText () 18-70 ページ	指定された日付を指定の書式の文字列に変換します。
OCIDateZoneToZone () 18-72 ページ	あるタイム・ゾーンの日付を別のゾーンの日付に変換します。

日時の例

次のコードの断片によって、TIMESTAMP WITH LOCAL TIME ZONE 列からデータを選択する **OCIDateTime** データ型の使用方法を示します。

...

```
/* allocate the program variable for storing the data */
OCIDateTime *tstmpltz = (OCIDateTime *)NULL;
```

```
/* Coll is a timestamp with local time zone column */
OraText *sqlstmt = (OraText *) "SELECT coll FROM foo";
```

```
/* Allocate the descriptor (storage) for the datatype */
status = OCIDescriptorAlloc(envhp, (dvoid **)&tstmpltz, OCI_DTYPE_TIMESTAMP_LTZ,
    0, (dvoid **)0);
....
```

```
status = OCISmtPrepare (stmthp, errhp, sqlstmt, (ub4)strlen ((char *)sqlstmt),
    (ub4)OCI_NTV_SYNTAX, (ub4)OCI_DEFAULT);
```

```
/* specify the define buffer for coll */
status = OCIDefineByPos(stmthp, &defnp, errhp, 1, &tstmpltz, sizeof(tstmpltz),
    SQLT_TIMESTAMP_LTZ, 0, 0, 0, OCI_DEFAULT));
```

```
/* Execute and Fetch */
OCISmtExecute(svchp, stmthp, errhp, 1, 0, (OCISnapshot *) NULL,
    (OCISnapshot *) NULL, OCI_DEFAULT)
```

At this point tstmpltz contains a valid timestamp with local time zone data. You can get the time zone name of the datetime data using:

```
status = OCIDateTimeGetTimeZoneName(envhp, errhp, tstmpltz, (ub1 *)buf,
    (ub4 *)buflen);
```

...

時間隔関数

次の関数は、時間隔データのみを操作します。対象となる時間隔の型の指定が必要な場合があります。可能な型は、次のとおりです。

- `SQLT_INTERVAL_YM` — 年から月までの時間隔
- `SQLT_INTERVAL_DS` — 日から秒までの時間隔

詳細は、各関数の説明を参照してください。

関連項目： 各関数の名前と用途のリストおよび詳細は、18-27 ページの [OCI の日付関数、日時関数および時間隔関数](#) を参照してください。

関数	用途
OCIIntervalAdd() 18-74 ページ	2 つの時間隔を加算して、結果の時間隔を生成します。
OCIIntervalAssign() 18-75 ページ	1 つの時間隔を別の時間隔にコピーします。
OCIIntervalCheck() 18-76 ページ	時間隔の妥当性をチェックします。
OCIIntervalCompare() 18-78 ページ	2 つの時間隔を比較します。
OCIIntervalDivide() 18-79 ページ	時間隔を Oracle 数値で除算して、時間隔を生成します。
OCIIntervalFromNumber() 18-80 ページ	Oracle 数値を時間隔に変換します。
OCIIntervalFromText() 18-81 ページ	時間隔文字列が指定されている場合、その文字列で表現される時間隔を生成します。
OCIIntervalGetDaySecond() 18-84 ページ	時間隔から日と秒の値を取得します。
OCIIntervalGetYearMonth() 18-86 ページ	時間隔から年と月を取得します。
OCIIntervalMultiply() 18-87 ページ	時間隔を Oracle 数値で乗算して、時間隔を生成します。
OCIIntervalSetDaySecond() 18-88 ページ	日と秒を時間隔に設定します。
OCIIntervalSetYearMonth() 18-90 ページ	年と月を時間隔に設定します。

関数	用途
OCIIntervalSubtract() 18-91 ページ	2つの時間隔を減算して、結果を時間隔に格納します。
OCIIntervalToNumber() 18-92 ページ	時間隔を Oracle 数値に変換します。
OCIIntervalToText() 18-93 ページ	時間隔が指定されている場合、その時間隔を表現する文字列を生成します。

数値（OCINumber）

OCINumber データ型は、Oracle の数値データ型（NUMBER、FLOAT、DECIMAL など）の表現に使用する不透明な構造体です。この型は、バインド・コールまたは定義コールで外部型コード `SQLT_VNU` を使用して、バインドまたは定義できます。

OCINumber 操作関数を機能別にまとめて次の表に示します。表内の数値という用語は、指定がないかぎり、**OCINumber** 型の値を表します。

関連項目： すべての関数のプロトタイプと説明は、[第 18 章「OCI のデータ型マッピング関数および操作関数」](#)を参照してください。

数値算術関数

算術演算を行う関数は、次のとおりです。

関数	用途
<code>OCINumberAbs()</code>	数値の絶対値を取得します。
<code>OCINumberAdd()</code>	2つの数値を合計します。
<code>OCINumberCeil()</code>	数値の上限値を取得します。
<code>OCINumberDec()</code>	数値を減らします。
<code>OCINumberDiv()</code>	1つの数値を別の数値で除算します。
<code>OCINumberFloor()</code>	数値の下限値を取得します。
<code>OCINumberInc()</code>	数値を増やします。
<code>OCINumberMod()</code>	2つの数値の除算からモジュロを取得します。
<code>OCINumberMul()</code>	2つの数値を乗算します。
<code>OCINumberNeg()</code>	数値を否定演算します。
<code>OCINumberRound()</code>	数値を指定の桁まで丸めます。

関数	用途
OCINumberShift()	数値を特定の桁数に変えます。
OCINumberSign()	数値の符号を取得します。
OCINumberSqrt()	数値の平方根を取得します。
OCINumberSub()	1つの数値を別の数値から減算します。
OCINumberTrunc()	数値を指定の桁で切り捨てます。
OCINumberSign()	指定の数値の符号を戻します。

数値変換関数

次の関数では、実数、整数および文字列の間で数値の変換を実行します。

関数	用途
OCINumberToInt()	数値を整数に変換します。
OCINumberFromInt()	整数を数値に変換します。
OCINumberToReal()	数値を実数に変換します。
OCINumberFromReal()	実数を数値に変換します。
OCINumberToText()	数値を文字列に変換します。
OCINumberFromText()	文字列を数値に変換します。

指数関数および対数関数

指数演算および対数演算を行う関数は、次のとおりです。

関数	用途
OCINumberPower()	指定の数値指数の数値基数をとります。
OCINumberExp()	基数 e の指数をとります。
OCINumberLog()	指定の基数の対数をとります。
OCINumberLn()	自然対数（基数 e ）をとります。
OCINumberIntPower()	指定の整数累乗の数値基数をとります。

三角関数

数値に対して三角関数演算を行う関数は、次のとおりです。

関数	用途
OCINumberArcCos()	アーク・コサイン（逆余弦）を計算します。
OCINumberArcSin()	アーク・サイン（逆正弦）を計算します。
OCINumberArcTan() / OCINumberArcTan2()	2つの数字のアーク・タンジェント（逆正接）を演算します。
OCINumberCos()	コサイン（余弦）を計算します。
OCINumberHypCos()	コサイン・ハイパボリック（余弦双曲線）を計算します。
OCINumberSin()	サイン（正弦）を計算します。
OCINumberHypSin()	サイン・ハイパボリック（正弦双曲線）を計算します。
OCINumberTan()	タンジェント（正接）を計算します。
OCINumberHypTan()	タンジェント・ハイパボリック（正接双曲線）を計算します。

数値の代入、比較および評価関数

数値に対して代入演算および比較演算を行う関数は、次のとおりです。

関数	用途
OCINumberAssign()	1つの数値を別の数値に代入します。
OCINumberCmp()	2つの数値を比較します。
OCINumberIsInt()	整数かどうかテストします。
OCINumberIsZero()	0に等しいかテストします。
OCINumberPrec()	精度を設定します。
OCINumberSetPi()	数値を円周率に設定します。
OCINumberSetZero()	数値を0に初期化します。

数値の例

次の例は、**OCINumber** 型の属性の処理方法を示しています。

```

struct person
{
    OCINumber sal;
};
typedef struct person person;
OCLError *err;
person* steve;
person* scott;
person* jason;
OCINumber *stevesal;
OCINumber *scottsal;
OCINumber *debsal;
sword status;
int inum;
double dnum;
OCINumber ornum;
char buffer[21];
ub4 buflen;
sword result;

/* For this example, assume OCIEnv and OCLError are initialized. */
/* For this example, assume that steve, scott and jason are pointing to
   person objects which have been pinned in the object cache. */
stevesal = &steve->sal;
scottsal = &scott->sal;
debsal = &jason->sal;

/* initialize steve's salary to be $12,000 */
inum = 12000;
status = OCINumberFromInt(err, &inum, sizeof(inum), OCI_NUMBER_SIGNED,
    stevesal);
if (status != OCI_SUCCESS) /* handle error from OCINumberFromInt */;

/* initialize scott's salary to be same as steve */
OCINumberAssign(err, stevesal, scottsal);

/* initialize jason's salary to be 20% more than steve's */
dnum = 1.2;
status = OCINumberFromReal(err, &dnum, DBL_DIG, &ornum);
if (status != OCI_SUCCESS) /* handle error from OCINumberFromReal */;
status = OCINumberMul(err, stevesal, &ornum, debsal);
if (status != OCI_SUCCESS) /* handle error from OCINumberMul */;

```

```

/* give scott a 50% raise */
dnum = 1.5;
status = OCINumberFromReal(err, &dnum, DBL_DIG, &ornum);
if (status != OCI_SUCCESS) /* handle error from OCINumberFromReal */;
status = OCINumberMul(err, scottsal, &ornum, scottsal);
if (status != OCI_SUCCESS) /* handle error from OCINumberMul */;

/* double steve's salary */
status = OCINumberAdd(err, stevesal, stevesal, stevesal);
if (status != OCI_SUCCESS) /* handle error from OCINumberAdd */;

/* get steve's salary in integer */
status = OCINumberToInt(err, stevesal, sizeof(inum), OCI_NUMBER_SIGNED,
    &inum);
if (status != OCI_SUCCESS) /* handle error from OCINumberToInt */;

/* inum is set to 24000 */
/* get jason's salary in double */
status = OCINumberToReal(err, debsal, sizeof(dnum), &dnum);
if (status != OCI_SUCCESS) /* handle error from OCINumberToReal */;

/* dnum is set to 14400 */
/* print scott's salary as DEM0001`8000.00 */
buflen = sizeof(buffer);
status = OCINumberToText(err, scottsal, "C0999G9999D99", 13,
    "NLS_NUMERIC_CHARACTERS='.' NLS_ISO_CURRENCY='Germany'",
    54, &buflen, buffer);
if (status != OCI_SUCCESS) /* handle error from OCINumberToText */;
printf("scott's salary = %s\n", buffer);

/* compare steve and scott's salaries */
status = OCINumberCmp(err, stevesal, scottsal, &result);
if (status != OCI_SUCCESS) /* handle error from OCINumberCmp */;

/* result is positive */
/* read jason's new salary from string */
status = OCINumberFromText(err, "48`000.00", 9, "99G999D99", 9,
    "NLS_NUMERIC_CHARACTERS='.'", 27, debsal);
if (status != OCI_SUCCESS) /* handle error from OCINumberFromText */;
/* jason's salary is now 48000.00 */

```

固定長または可変長文字列（OCIString）

固定長または可変長の文字列データは、C プログラムに対して **OCIString *** として表現されます。

文字列の長さには、NULL 文字は含まれていません。

OCIString * 型の変数のバインドと定義には、外部型コード **SQLT_VST** を使用します。

関連項目： すべての関数のプロトタイプと説明は、[第 18 章「OCI のデータ型マッピング関数および操作関数」](#)を参照してください。

文字列関数

C のプログラマは、次の関数を使用して文字列のインスタンスを操作できます。

関数	用途
OCIStringAllocSize()	文字列メモリーの割り当てられたサイズ（バイト単位）を取得します。
OCIStringAssign()	1 つの文字列を別の文字列に代入します。
OCIStringAssignText()	テキスト文字列を文字列に代入します。
OCIStringPtr()	文字列の文字列部分へのポインタを取得します。
OCIStringResize()	文字列メモリーをサイズ変更します。
OCIStringSize()	文字列サイズを取得します。

文字列の例

この例では、テキスト文字列を文字列に代入し、文字列の文字列部分へのポインタを文字列サイズとともに取得して、出力します。

OCIStringAssignText() で、*vstring1* パラメータを渡すために二重間接演算が使用されていることに注意してください。

```
OCIEnv      *envhp;
OCIError    *errhp;
OCIString   *vstring1 = (OCIString *)0;
OCIString   *vstring2 = (OCIString *)0;
text        c_string[20];
text        *text_ptr;
sword       status;
```

```
strcpy(c_string, "hello world");
/* Assign a text string to an OCIStrng */
status = OCIStrngAssignText(envhp, errhp, c_string,
                             (ub4)strlen(c_string), &vstring1);
/* Memory for vstring1 is allocated as part of string assignment */

status = OCIStrngAssignText(envhp, errhp, "hello again",
                             (ub4)strlen("This is a longer string."), &vstring1);
/* vstring1 is automatically resized to store the longer string */

/* Get a pointer to the string part of vstring1 */
text_ptr = OCIStrngPtr(envhp, vstring1);
/* text_ptr now points to "hello world" */
printf("%s\n", text_ptr);
```

ロー (OCIRaw)

可変長ロー・データは、**OCIRaw *** データ型を使用して C で表現されます。
OCIRaw * 型の変数のバインドと定義には、外部型コード **SQLT_LVB** を使用します。

関連項目： すべての関数のプロトタイプと説明は、[第 18 章「OCI のデータ型マッピング関数および操作関数」](#) を参照してください。

ロー関数

OCIRaw の操作を行う関数は、次のとおりです。

関数	用途
OCIRawAllocSize()	ロー・メモリーの割り当てられたサイズ (バイト単位) を取得します。
OCIRawAssignBytes()	ロー・データ (ub1 *) を OCIRaw * に代入します。
OCIRawAssignRaw()	1 つの OCIRaw * を別の OCIRaw * に代入します。
OCIRawPtr()	ロー・データへのポインタを取得します。
OCIRawResize()	可変長ロー・データのメモリーのサイズ変更を行います。
OCIRawSize()	ロー・データのサイズを取得します。

ローの例

この例では、ロー・データのブロックを設定し、そのデータへのポインタを取得します。

OCIRawAssignBytes() のコールで二重間接演算が使用されていることに注意してください。

```
OCIEnv      *envhp;
OCIError    *errhp;
sword       status;
ub1         data_block[10000];
ub4         data_block_len = 10000;
OCIRaw      *rawl;
ub1 *rawl_pointer;

/* Set up the RAW */
/* assume 'data_block' has been initialized */
status = OCIRawAssignBytes(envhp, errhp, data_block, data_block_len, &rawl);

/* Get a pointer to the data part of the RAW */
rawl_pointer = OCIRawPtr(envhp, rawl);
```

コレクション (OCITable、OCIArray、OCIColl、OCIIter)

Oracle には、可変長配列 (VARRAY) と NESTED TABLE の 2 種類のコレクションがあります。C アプリケーションでは、VARRAY は **OCIArray *** と表現され、NESTED TABLE は、**OCITable *** と表現されます。これらの両方のデータ型は (後述する OCIColl と OCIIter と同様に)、不透明な構造体です。

様々な汎用コレクション関数により、コレクション・データを操作できます。汎用コレクション関数は、VARRAY と NESTED TABLE の両方に使用できます。また、NESTED TABLE 固有の一連の関数もあります。

関連項目： 11-25 ページ [「NESTED TABLE 操作関数」](#)

VARRAY または NESTED TABLE のインスタンスは、OCIObjectNew() を使用して割り当て、OCIObjectFree() を使用して解放できます。

関連項目： すべての関数のプロトタイプと説明は、18-5 ページの [「OCI コレクションおよびイテレータ関数」](#) を参照してください。

汎用コレクション関数

Oracle には、可変長配列（VARRAY）と NESTED TABLE の 2 種類のコレクションがあります。VARRAY と NESTED TABLE は、どちらも汎用コレクション型のサブタイプとみなすことができます。

C では、汎用コレクションを **OCIColl ***、VARRAY を **OCIArray *** および NESTED TABLE を **OCITable *** と表現します。Oracle には、汎用コレクションを操作するための一連の関数群（**OCIColl *** など）があります。これらの関数は、OCICollGetElem() のように接頭辞 OCIColl で始まります。OCIColl*() 関数をコールして、VARRAY と NESTED TABLE を操作することもできます。

汎用コレクション関数は、2 つの主なカテゴリにグループ化できます。

- VARRAY または NESTED TABLE のデータを操作するグループ
- コレクション・イテレータを使用してコレクションをスキャンするグループ

汎用コレクション関数は、VARRAY を操作する関数の完全な集合です。その他の関数は、特に NESTED TABLE で操作するための関数です。これらの関数は、OCITableExists() のように接頭辞 OCITable で識別できます。

関連項目： 11-25 ページ「[NESTED TABLE 操作関数](#)」

注意： コレクション関数に渡された索引は、0（ゼロ）が基数です。

コレクション・データ操作関数

次の汎用関数は、コレクション・データを操作します。

関数	用途
OCICollAppend()	要素を追加します。
OCICollAssignElem()	指定の索引位置に要素を割り当てます。
OCICollAssign()	1 つのコレクションを別のコレクションに割り当てます。
OCICollGetElem()	指定の索引位置にある要素へのポインタを取得します。
OCICollMax()	コレクションの上限を取得します。
OCICollSize()	コレクションのカレント・サイズを取得します。
OCICollTrim()	コレクションの終わりから n 個の要素を切り捨てます。

コレクション・スキャン関数

次の汎用関数により、コレクション・イテレータを使用してコレクションをスキャンできます。イテレータは、**OCIIter** 型であり、最初に `OCIIterCreate()` をコールして作成します。

関数	用途
<code>OCIIterCreate()</code>	コレクションをスキャンするためのイテレータを作成します。
<code>OCIIterDelete()</code>	イテレータを削除します。
<code>OCIIterGetCurrent()</code>	イテレータが指し示すカレント要素へのポインタを取得します。
<code>OCIIterInit()</code>	イテレータを初期化して指定のコレクションをスキャンします。
<code>OCIIterNext()</code>	次要素へのポインタを取得します。
<code>OCIIterPrev()</code>	前要素へのポインタを取得します。

VARRAY/ コレクション・イテレータの例

この例では、コレクション・イテレータを作成し、それを使用して **VARRAY**（可変長配列）をスキャンします。

```
OCIEnv      *envhp;
OCIError     *errhp;
text         *text_ptr;
sword        status;
OCIArray     *clients;
OCIString    *client_elem;
OCIIter      *iterator;
boolean      eoc;
dvoid        *elem;
OCIInd       *elemind;

/* Assume envhp, errhp have been initialized */
/* Assume clients points to a varray */

/* Print the elements of clients */
/* To do this, create an iterator to scan the varray */
status = OCIIterCreate(envhp, errhp, clients, &iterator);

/* Get the first element of the clients varray */
printf("Clients' list:\n");
status = OCIIterNext(envhp, errhp, iterator, &elem,
                    (dvoid **) &elemind, &eoc);
```

```
while (!eoc && (status == OCI_SUCCESS))
{
    client_elem = *(OCIStrng)**elem;
                                /* client_elem points to the string */

    /*
       the element pointer type returned by OCIIterNext() through 'elem' is
       the same as that of OCICollGetElem(). Refer to OCICollGetElem() for
       details. */

    /*
       client_elem points to an OCIStrng descriptor, so to print it out,
       get a pointer to where the text begins
    */
    text_ptr = OCIStrngPtr(envhp, client_elem);

    /*
       text_ptr now points to the text part of the client OCIStrng, which is a
       NULL-terminated string
    */
    printf(" %s\n", text_ptr);
    status = OCIIterNext(envhp, errhp, iterator, &elem,
                        (dvoid **)&elemind, &eoc);
}

if (status != OCI_SUCCESS)
{
    /* handle error */
}

/* destroy the iterator */
status = OCIIterDelete(envhp, errhp, &iterator);
```


NESTED TABLE 操作関数

表はネストできます。つまり、変数、属性、パラメータまたは列として別の表の中に含めることができます。NESTED TABLE には、OCITableDelete() 関数で削除された要素がある場合があります。

たとえば、10 個の要素を指定して作成した表があり、0 から 4 と 9 の索引を持つ要素を OCITableDelete() で削除したとします。既存の最初の要素は要素 5 になり、既存の最後の要素は要素 8 になります。

前述のように、汎用コレクション関数を使用しても、NESTED TABLE へのマッピングや NESTED TABLE の操作ができます。さらに、次に示す NESTED TABLE 専用の関数もあります。NESTED TABLE 専用の関数は、VARRAY では使用できません。

関数	用途
OCITableDelete()	指定の索引位置にある要素を削除します。
OCITableExists()	指定の索引位置に要素が存在するかどうかをテストします。
OCITableFirst()	表の最初の既存要素の索引を戻します。
OCITableLast()	表の最後の既存要素の索引を戻します。
OCITableNext()	表の次の既存要素の索引を戻します。
OCITablePrev()	表の前の既存要素の索引を戻します。
OCITableSize()	削除した要素を除いた表のサイズを戻します。

NESTED TABLE の要素への順序付け

NESTED TABLE がオブジェクト・キャッシュにフェッチされた場合、その要素には、0（ゼロ）から始まり要素数から 1 を引いた数で終わる順序が一時的に付けられます。たとえば、40 個の要素を持つ表では、0 から 39 まで順序付けが行われます。

これらの位置序数を使用して、要素の値のフェッチと代入が行われます（たとえば、要素 *i* へのフェッチや、要素 *j* への代入など。*i* と *j* は、指定の表の有効な位置序数です）。

表をデータベースにコピーすると、一時的な順序付けは失われます。表の要素に対して削除の操作を行うことがあります。操作を削除すると、一時的な空きが作成されます。つまり残りの表要素の位置序数は変更されません。

NESTED TABLE のロケータ

NESTED TABLE のロケータを取り出すことができます。ロケータはコレクション値へのハンドルのようなもので、検索時に存在するデータベース・スナップショットについての情報を含みます。このスナップショット情報は、後にロケータを使用してコレクション要素がフェッチされるときに、データベースでコレクション値の正しいインスタンス化を取り出すために役立ちます。

LOB ロケータと異なり、コレクション・ロケータでは、コレクション・インスタンスを変更できません。コレクション・ロケータは、単に正しいデータの検索のみを行います。このロケータを使用すると、アプリケーションは、大きなコレクション全体を取り出すことなく、NESTED TABLE にハンドルを戻すことができます。

コレクション列または属性をフェッチするときにロケータが戻される必要がある場合、ユーザーは、RETURN AS LOCATOR 指定を使用して、表がいつ作成されるかを指定します。

関連項目： 詳細は、『Oracle9i SQL リファレンス』を参照してください。

コレクションがロケータかどうかを判断するには、OCI CollIsLocator() 関数を使用します。

マルチレベル・コレクション型

コレクション要素自体が直接的または間接的に別のコレクション型である場合があります。マルチレベル・コレクション型とは、このようなトップレベルのコレクション型に付けられた名前です。

マルチレベル・コレクションには、次のような特性があります。

- 他のコレクション型の集まりである場合があります。
- コレクション属性を持つオブジェクトの集まりである場合があります。
- ネスト・レベルの数に制限がありません。
- VARRAY と NESTED TABLE の組合せを含めることができます。
- 表の列として使用できます。

OCI ルーチンは、マルチレベル・コレクションを処理します。次のルーチンは、パラメータ *elem の OCIColl* を戻すことができるため、このパラメータは任意のコレクションのルーチンで使用できます。

- OCICollgetElem()
- OCIIterGetCurrent()
- OCIIterNext()
- OCIIterPrev()

次の関数は、コレクション要素を取得し、それを既存のコレクションに追加します。要素型が別のコレクションの場合、パラメータ `elem` は、**OCIColl*** となることがあります。

- OCICollAssignElem()
- OCICollAppend()

マルチレベル・コレクション型の例

たとえば、次の型と表を使用するとします。

```
type_1 (a NUMBER, b NUMBER)
NT1 TABLE OF type_1
NT2 TABLE OF NT1
```

次のコードの断片がマルチレベル・コレクション全体で反復されます。

```
...
OCIColl *outer_coll;
OCIColl *inner_coll;
OCIIter *itr1, *itr2;
Type_1 *type_1_instance;
....
/* assume outer_coll points to a valid coll of type NT2 */
checkerr(OCIIterCreate(envhp, errhp, outer_coll, &itr1));
for(eoc = FALSE; !OCIIterNext(envhp, errhp, itr, (dvoid **) &elem,
                             (dvoid **) &elem_null, &eoc) && !eoc;)
{
    inner_coll = (OCIColl *)elem;
    /* iterate over inner collection.. */
    checkerr(errhp, OCIIterCreate(envhp, errhp, inner_coll, &itr2));
    for(eoc2 = FALSE; !OCIIterNext(envhp, errhp, itr2, (dvoid **) &elem2,
                                   (dvoid **) &elem2_null, &eoc2) && !eoc2;)
    {
        type_1_instance = (Type_1 *)elem2;
        /* use the fields of type_1_instance */
    }
    /* close iterator over inner collection */
    checkerr(errhp, OCIIterDelete(envhp, errhp, &itr2));
}
/* close iterator over outer collection */
checkerr(errhp, OCIIterDelete(envhp, errhp, &itr1));
...
```

REF (OCIRef)

REF (参照) とは、オブジェクトを識別する識別子です。これは、一意にオブジェクトを検索する不透明な構造体です。REF によって、オブジェクトは別のオブジェクトを指し示すことができます。

C アプリケーションでは、**OCIRef *** で REF を表現します。

関連項目： すべての関数のプロトタイプと説明は、[第 18 章「OCI のデータ型マッピング関数および操作関数」](#) を参照してください。

REF 操作関数

REF 操作を行う関数は、次のとおりです。

関数	用途
OCIRefAssign()	1 つの REF を別の REF に代入します。
OCIRefClear()	REF を消去または NULL 化します。
OCIRefFromHex()	16 進数文字列を REF に変換します。
OCIRefHexSize()	REF の 16 進数文字列表現のサイズを戻します。
OCIRefIsEqual()	2 つの REF が等しいかどうかを比較します。
OCIRefIsNull()	REF が NULL かどうかをテストします。
OCIRefToHex()	REF を 16 進数文字列に変換します。

REF の例

この例では、2 つの REF が NULL かどうかをテストし、両者が等しいかどうかを比較し、一方の REF をもう一方の REF に代入します。OCIRefAssign() のコールで二重間接演算を使用していることに注意してください。

```
OCIEnv      *envhp;
OCIError    *errhp;
sword       status;
boolean      refs_equal;
OCIRef      *ref1, *ref2;

/* assume refs have been initialized to point to valid objects */
/*Compare two REFs for equality */
refs_equal = OCIRefIsEqual(envhp, ref1, ref2);
printf("After first OCIRefIsEqual:\n");
if(refs_equal)
```

```
printf("REFs equal\n");
else
    printf("REFs not equal\n");

/*Assign ref1 to ref2 */
status = OCIRefAssign (envhp, errhp, ref1, &ref2);
if(status != OCI_SUCCESS)
/*error handling*/

/*Compare the two REFs again for equality */
refs_equal = OCIRefIsEqual(envhp, ref1, ref2);
printf("After second OCIRefIsEqual:\n");
if(refs_equal)
    printf("REFs equal\n");
else
    printf("REFs not equal\n");
```

オブジェクト型情報の格納およびアクセス

記述子オブジェクト

ある型を CREATE TYPE 文で作成すると、その型はサーバーに格納され、型記述子オブジェクト (TDO) に関連付けられます。さらに、型の各データ属性、型の各メソッド、各メソッドの各パラメータ、メソッドが戻す結果の記述子オブジェクトが、データベースに格納されます。各型の記述子オブジェクトに関連付けられた OCI データ型のリストを次の表に示します。

情報の型		OCI データ型
型		OCIType
型属性メソッド・パラメータ	コレクション要素 メソッドの結果	OCITypeElem
メソッド		OCITypeMethod

OCI 関数の中には (OCIBindObject() および OCIObjectNew() を含む)、TDO を入力パラメータとして必要とする関数があります。アプリケーションで、その型の TDO を OCIType 変数で取得できる OCITypeByName() をコールして TDO を取得できます。TDO を取得すると、必要に応じて、その TDO をその他のコールに渡すことができます。

AnyType インタフェース、AnyData インタフェースおよび AnyDataSet インタフェース

これらの機能によって、自己記述データをモデル化できます。異機種間データ型を同じ列に格納し、アプリケーションでそのデータの型を問い合わせることができます。

後続の項の説明では、これらの定義が使用されています。

- 永続型 — 永続型は、SQL の CREATE TYPE 文を使用して作成します。データベースに永続的に格納されます。
- 一時型 — データベースに永続的に格納されない匿名型記述子。プログラムによって一時的に作成されます。一時型は、型情報をアプリケーションの様々なコンポーネント間で、必要に応じて動的に交換するのに便利です。
- 自己記述データ — 型情報をその実際の内容とともにカプセル化したデータ。OCI では、**OCIAnyData** データ型によって、このようなデータがモデル化されます。大部分の SQL 型のデータ値は、**OCIAnyData** に変換できます。また、元のデータ値に変換しなおすこともできます。SQL や PL/SQL では、SYS.ANYDATA 型によって、このようなデータがモデル化されます。
- 自己記述データセット — データ・インスタンスの集合（すべて同じ型）およびその型記述のカプセル化。この集合には、すべて同じ型記述が含まれている必要があります。OCI では、**OCIDataAnySet** データ型によって、このデータがモデル化されます。SQL や PL/SQL では、SYS.ANYDATASET 型によって、このデータセットがモデル化されます。

これらの型記述および自己記述データの作成と操作に必要なインタフェースは、OCI（C 言語）および SQL と PL/SQL の両方で使用可能です。次の各項では、関連する OCI インタフェースを説明します。

関連項目： 概要は、10-5 ページの「[永続オブジェクト、一時オブジェクトおよび値](#)」および『[Oracle9i SQL リファレンス](#)』の「Oracle が提供する型」の項を参照してください。

型インタフェース

型インタフェースを使用すると、名前付きおよび匿名の一時オブジェクト型（属性付きで構造化されたもの）とコレクション型を作成できます。OCITypeBeginCreate() コールを使用して、一時オブジェクト型とコレクション型の作成を開始します（作成する型は、型コード・パラメータによって決定されます）。

OCIDescriptorAlloc() を使用して、パラメータ・ハンドルを割り当てる必要があります。次に、OCIAttrSet() を使用して、型情報（オブジェクト型の属性とコレクション要素の型に関する情報）を設定する必要があります。オブジェクト型の場合は、OCITypeAddAttr() を使用して属性情報をその型に追加します。最新の属性情報が追加された後で、OCITypeEndCreate() をコールする必要があります。

次に例を示します。

```
OCITypeBeginCreate( ...)      /* Begin Type Creation */
OCIDescriptorAlloc(...)
OCIAttrSet(...)
OCITypeAddAttr(...)          /* Add attribute 1 */
OCIAttrSet(...)
OCITypeAddAttr(...)          /* Add attribute 2 */
...
OCITypeEndCreate(...)        /* End Type Creation */
```

コレクション型の場合は、コレクション要素型に関する情報を、`OCITypeSetCollection()` を使用して設定する必要があります。次に、`OCITypeEndCreate()` をコールして構成を終了します。

次に例を示します。

```
OCITypeBeginCreate( ...)      /* Begin Type Creation */
OCIDescriptorAlloc(...)
OCIAttrSet(...)
OCITypeSetCollection(...)     /* Set information on collection element etc.*/
OCITypeEndCreate(...)        /* End Type Creation */
```

`OCIDescribeAny()` コールを使用すると、永続型に対応する **OCIType** を取得できます。

OCIType コールのパラメータ記述子の作成

`OCIDescriptorAlloc()` コールを使用すると、**OCIPParam**（親ハンドルが環境ハンドルの場合）を割り当てることができます。その後、次の使用可能な属性の型を使用して、`OCIAttrSet()` をコールし、関連する型情報を設定できます。

■ OCI_ATTR_PRECISION

数値精度を設定します。(ub1 *) 属性値を、精度値を保持しているバッファに渡します。

■ OCI_ATTR_SCALE

数値のスケールを設定します。(sb1 *) 属性値を、スケール値を保持しているバッファに渡します。

■ OCI_ATTR_CHARSET_ID

キャラクタ・タイプのキャラクタ・セット ID を設定します。(ub2 *) 属性値を、キャラクタ・セット ID を保持しているバッファに渡します。

■ OCI_ATTR_CHARSET_FORM

キャラクタ・タイプのキャラクタ・セット・フォームを設定します。(ub1 *) 属性値を、キャラクタ・セット・フォーム値を保持しているバッファに渡します。

- OCI_ATTR_DATA_SIZE

VARCHAR2、RAW などの長さ。(ub4 *) 属性値を、長さを保持しているバッファに渡します。

- OCI_ATTR_TYPECODE

型コードを設定します。(ub2 *) 属性値を、型コードを保持しているバッファに渡します。この属性は、最初に設定する必要があります。

- OCI_ATTR_TDO

オブジェクトまたはコレクション属性の **OCIType** を設定します。(OCIType *) 属性値を、その属性に対応する **OCIType** に渡します。**AnyType** の構成中に、この **OCIParam** を使用する場合、**OCIType** が確保されていることを確認してください。一時型属性の場合の割当て時間は、少なくともトップレベルの **OCIType** が作成されている時間内である必要があります。そうでない場合は、例外が戻されます。

- 組み込み型の場合、SQL 型属性にアクセス可能な型コード (OCI_ATTR_TYPECODE に対する許容値) は、次のとおりです。

OCI_TYPECODE_DATE、OCI_TYPECODE_NUMBER、OCI_TYPECODE_VARCHAR、
OCI_TYPECODE_RAW、OCI_TYPECODE_CHAR、OCI_TYPECODE_VARCHAR2、
OCI_TYPECODE_VARCHAR、OCI_TYPECODE_BLOB、OCI_TYPECODE_BFILE、
OCI_TYPECODE_CLOB

OCI_TYPECODE_TIMESTAMP、OCI_TYPECODE_TIMESTAMP_TZ、
OCI_TYPECODE_TIMESTAMP_LTZ

OCI_TYPECODE_INTERVAL_YM、OCI_TYPECODE_INTERVAL_DS

- 属性 / コレクション要素の型自体が別の一時型である場合は、OCI_ATTR_TYPECODE を次のいずれかに設定します。

OCI_TYPECODE_OBJECT、OCI_TYPECODE_REF (REF 用)、
OCI_TYPECODE_VARRAY または OCI_TYPECODE_TABLE。次に、OCI_ATTR_TDO
をその一時型に対応する **OCIType** に設定します。

- ユーザー定義の型属性の場合、OCI_ATTR_TYPECODE に対する許容値は、次のとおりです。

- OCI_TYPECODE_OBJECT (オブジェクト型用)

- OCI_TYPECODE_REF (REF 型用)

- OCI_TYPECODE_VARRAY または OCI_TYPECODE_TABLE (コレクション用)

ユーザー定義の場合は、OCI_ATTR_TDO を適切なユーザー定義型の **OCIType** に設定する必要があります。

永続型の OCIType の取得

OCIDescribeAny() コールを使用すると、永続型に対応する **OCIType** を取得できます。次に例を示します。

```
OCIDescribeAny(svchp, errhp, (dvoid *)"SCOTT.EMP", (ub4)strlen("SCOTT.EMP"),
               (ub1)OCI_OTYPE_NAME, (ub1)OCI_DEFAULT, OCI_PTYPE_TYPE, dschp);
```

記述ハンドル (*dschp*) から、OCIAttrGet() コールを使用して、**OCIType** を取得できます。

型のアクセス・コール

一時型の記述を使用して OCIDescribeAny() をコールし、その型の動的記述にアクセスできます。**OCIType** のポインタは *objtype* が OCI_OTYPE_PTR に設定されている場合、OCIDescribeAny() に直接渡すことができます。これによって、属性情報を名前および位置で取得できます。

OCIDescribeAny() への拡張

一時型が組み込み型である（組み込みの型コードで作成されている）場合、この一時型（OCI_PTYPE_TYPE 型に相当）を記述するパラメータ・ハンドルでは、次の追加の属性がサポートされます。

OCI_ATTR_DATA_SIZE

OCI_ATTR_TYPECODE

OCI_ATTR_DATA_TYPE

OCI_ATTR_PRECISION

OCI_ATTR_SCALE

OCI_ATTR_CHARSET_ID

OCI_ATTR_CHARSET_FORM

OCI_ATTR_LFPRECISION

OCI_ATTR_FSPRECISION

これらの属性には、型属性の記述時に通常付与される内容が含まれています。

注意： これらの属性は、一時的な組み込み型に対してのみサポートされます。属性 OCI_ATTR_IS_TRANSIENT_TYPE と OCI_ATTR_IS_PREDEFINED_TYPE は、これらの型に該当します。永続型の場合、これらの属性は、その型の属性（OCI_PTYPE_TYPE_ATTR 型に相当）のパラメータ・ハンドルでのみサポートされます。

OCIAnyData インタフェース

OCIAnyData は、型情報とその型のデータ・インスタンス（つまり、自己記述データ）をカプセル化します。**OCIAnyData** は、`OCIAnyDataConvert()` コールを使用して、組込み型またはユーザー定義型のインスタンスから作成できます。このコールが **OCIAnyData** への変換（キャスト）を実行します。

オブジェクト型とコレクション型は、個別に作成することもできます。つまり、一度に1つのオブジェクト型の属性または一度に1つのコレクション要素を作成できます。個別に作成する場合は、型情報（**OCIType**）を使用して `OCIAnyDataBeginCreate()` をコールする必要があります。次に、`OCIAnyDataAttrSet()` を使用してオブジェクト型をコールし、`OCIAnyDataCollAddElem()` を使用してコレクション型をコールします。構成処理を終了するには、`OCIAnyDataEndCreate()` をコールする必要があります。

その後で、アクセス・ルーチンを起動できます。**OCIAnyData** を対応する型のインスタンスに変換（キャスト）するには、`OCIAnyDataAccess()` コールを使用します。

オブジェクト型またはコレクション型に基づいた **OCIAnyData** も個別にアクセスできます。

パフォーマンス改善のために、特別なコレクション構成とアクセス・コールが用意されています。このコールを使用すると、メモリー内での不要なコレクションの作成やコレクション全体のコピーなどが回避できます。次に例を示します。

```
OCIAnyDataConvert(...)      /* Cast a builtin or user-defined type instance
                             to an OCIAnyData in 1 call. */

OCIAnyDataBeginCreate(...)   /* Begin AnyData Creation */

OCIAnyDataAttrSet(...)       /* Attribute-wise construction for object types */

または

OCIAnyDataCollAddElem(...)   /* Element-wise construction for collections */

OCIAnyDataEndCreate(...)     /* End OCIAnyData Creation */
```

OCIAnyData 関数の NCHAR 型コード

`OCIAnyDataTypeCodeToSqlit()` 関数は、AnyData 値の **OCITypeCode** を、OCIAnyData API が戻す値の表現に対応する **SQLT** コードに変換します。

次の型コードは、OCIAnyData 関数でのみ使用されます。

- OCI_TYPECODE_NCHAR
- OCI_TYPECODE_NVARCHAR2
- OCI_TYPECODE_NCLOB

OCIDescribeAny() などその他の関数のコールでは、これらの型コードは戻されず、キャラクタ・セット・フォームを使用してデータが NCHAR かどうか（キャラクタ・セット・フォームが SQLCS_NCHAR かどうか）を判断する必要があります。

OCIAnyDataTypeCodeToSqlt() は、OCI_TYPECODE_VARCHAR2 および OCI_TYPECODE_CHAR を、キャラクタ・セット・フォームが SQLCS_IMPLICIT の出力値 SQLT_VST (OCIString マッピングに対応) に変換します。また、OCI_TYPECODE_NVARCHAR2 は、キャラクタ・セット・フォームが SQLCS_NCHAR の SQLT_VST (OCIString マッピングは OCIAnyData API で使用されます) を戻します。

関連項目： 詳細は、20-33 ページの「[OCIAnyDataTypeCodeToSqlt\(\)](#)」を参照してください。

OCIAnyDataSet インタフェース

OCIAnyDataSet は、型情報とその型のインスタンスの集合をカプセル化します。構成処理を開始するには、OCIAnyDataSetBeginCreate() をコールします。

OCIAnyDataSetAddInstance() をコールして新規インスタンスを追加すると、このコールは、そのインスタンスに対応する **OCIAnyData** を戻します。

次に、このインスタンスを作成するための OCIAnyData 関数を呼び出すことができます。すべてのインスタンスが追加された後に、OCIAnyDataSetEndCreate() をコールします。

アクセスする場合は、OCIAnyDataSetGetInstance() をコールして、そのインスタンスに対応する **OCIAnyData** を取得します。順次アクセスのみがサポートされます。その後、OCIAnyData アクセス関数を呼び出すことができます。たとえば、次のように行います。

```
OCIAnyDataSetBeginCreate(...)    /* Begin AnyDataSet Creation */
OCIAnyDataSetAddInstance(...)    /* Add a new instance to the AnyDataSet */
                                /* Use the OCIAnyData*() functions to create
                                the instance */
OCIAnyDataSetEndCreate(...)      /* End OCIAnyDataSet Creation */
```

注意： これらのインタフェースでの各コールの詳細は、第 20 章「[OCI の任意型関数および任意データ関数](#)」を参照してください。

名前付きデータ型のバインド

この項では、オブジェクトやコレクションなどの名前付きデータ型と REF のバインドについて説明します。

名前付きデータ型のバインド

名前付きデータ型（オブジェクト型やコレクション）のバインドでは、`OCIBindByName()` または `OCIBindByPos()` の後にもう 1 つのバインド・コールが必要です。`OCIBindObject()` という OCI オブジェクト型バインド・コールで、オブジェクト型のバインド固有の追加属性を設定します。OCI アプリケーションでは、このコールを使用して、オブジェクト・データ型の列を持つ表からデータをフェッチします。

`OCIBindObject()` コールでパラメータの 1 つとして、名前付きデータ型の TDO を受け取ります。TDO（データ型 **OCIType**）は、名前付きデータ型の作成時に作成され、データベースに格納されます。TDO には、型とその属性についての情報が入っています。アプリケーションでは、`OCITypeByName()` をコールして TDO を取得できます。

`OCIBindObject()` コールでは、名前付きデータ型バインド用のインジケータ変数または構造体も設定します。

名前付きデータ型をバインドするときは、`SQLT_NTY` データ型定数を使用して、バインドするプログラム変数のデータ型を指定します。`SQLT_NTY` は、名前付きデータ型を表現する C 構造体をバインドすることを示します。この構造体へのポインタをバインド・コールに渡します。

スーパータイプがある場合は、継承とインスタンスの代入性を使用して、サブタイプのインスタンスをバインドできます。

場合によっては、名前付きデータ型の操作に 3 つのバインド・コールが必要になることがあります。たとえば、名前付きデータ型の静的配列を PL/SQL 表にバインドするには、`OCIBindByName()`、`OCIBindArrayOfStruct()` および `OCIBindObject()` という 3 つのコールを呼び出してください。

関連項目：

- これらのデータ型を使用してデータベースから埋込みオブジェクトをフェッチする方法の詳細は、10-15 ページの「[埋込みオブジェクトのフェッチ](#)」を参照してください。
- 他の重要な情報は、11-37 ページの「[名前付きデータ型および REF バインドの情報](#)」を参照してください。
- 記述子オブジェクトの詳細は、11-29 ページの「[記述子オブジェクト](#)」を参照してください。

REF のバインド

名前付きデータ型と同様に、REF のバインドも 2 ステップで処理します。最初に `OCIBindByName()` または `OCIBindByPos()` をコールし、次に `OCIBindObject()` をコールします。

REF は、`SQLT_REF` データ型を使用してバインドします。`SQLT_REF` を使用する場合、バインドするプログラム変数は、`OCIRef *` 型の変数にする必要があります。

スーパータイプへの REF がある場合は、継承と REF の代入性を使用して、REF の値をサブタイプのインスタンスにバインドできます。

関連項目：

- REF のオブジェクトへのバインドと確保については、10-10 ページの「[サーバーからのオブジェクト参照の取出し](#)」を参照してください。
- 他の重要な情報は、11-37 ページの「[名前付きデータ型および REF バインドの情報](#)」を参照してください。

名前付きデータ型および REF バインドの情報

この項には、名前付きデータ型と REF のバインドを処理するときに注意する必要がある重要な追加情報について説明します。メモリー割当てとインジケータ変数の使用方法についての参照先も示します。

- バインド対象のデータ型が `SQLT_NTY` である場合は、`OCIBindObject()` コールのインジケータ構造体パラメータ (`dvoid ** indpp`) が使用され、スカラー・インジケータは完全に無視されます。
- データ型が `SQLT_REF` の場合は、スカラー・インジケータが使用され、`OCIBindObject()` のインジケータ構造体のパラメータは完全に無視されます。
- インジケータ構造体の使用はオプションです。ユーザーは、`indpp` パラメータの `NULL` ポインタを `OCIBindObject()` コールに渡すことができます。これは、バインド時に、オブジェクトがアトミック `NULL` ではないこと、そのオブジェクトのどの属性も `NULL` ではないことを意味します。
- `OCIBindObject()` コールのインジケータ構造体サイズ・ポインタ `indsp` およびプログラム変数サイズ・ポインタ `pgvsp` は、オプションです。これらのパラメータが不要な場合、ユーザーは `NULL` を渡すことができます。

配列バインドに関する情報

配列挿入または配列フェッチ用に、名前付きデータ型または REF の配列バインドを行うには、ユーザーはポインタの配列を適切な型のバッファ（事前割当て済みまたはそれ以外）に渡す必要があります。同様に、スカラー・インジケータの配列（`SQLT_REF` 型）またはポインタ配列を、インジケータ構造体（`SQLT_NTY` 型）に渡す必要があります。

関連項目： `SQLT_NTY` の詳細は、3-18 ページの「[新しい外部データ型](#)」を参照してください。

名前付きデータ型の定義

この項では、名前付きデータ型（オブジェクト、コレクションなど）と `REF` の定義について説明します。

名前付きデータ型出力変数の定義

名前付きデータ型（オブジェクト型、`NESTED TABLE`、`VARRAY`）の定義には、2 つの定義コールが必要です。最初に `dtv` パラメータに `SQLT_NTY` を指定して、`OCIDefineByPos()` をコールします。アプリケーションでは、`OCIDefineByPos()` に続いて、`OCIDefineObject()` をコールする必要があります。この場合、`OCIDefineByPos()` のデータ・バッファ・ポインタは無視され、名前付きデータ型定義に関係するその他の属性は、OCI オブジェクト属性定義コールである `OCIDefineObject()` を使用して設定されます。

そのコールでは名前付きデータ型定義用の `SQLT_NTY` データ型定数を指定します。この場合、結果のデータが名前付きデータ型のホスト言語表現の中にフェッチされます。通常、これは `Object Type Translator (OTT)` によって生成される C 構造体になります。

`OCIDefineObject()` コールを行うときは、C 構造体のアドレスへのポインタ（事前割当て済みまたはそれ以外）を定義する必要があります。オブジェクトは、`OCIObjectNew()` を使用して作成され、キャッシュまたはユーザー定義のメモリーに割り当てられます。

ただし、継承が存在する場合は、オブジェクト・キャッシュ内のオブジェクトを使用し、スタックのユーザー・メモリーから割り当てられたオブジェクトを渡さないことをお勧めします。インスタンスの代入性により、クライアントにスーパータイプのインスタンスがあるときは、サーバーがサブタイプのインスタンスを戻す場合があるためです。サーバーは、オブジェクトを動的にサイズ変更する必要があります。この操作は、キャッシュ内のオブジェクトに対してのみ可能です。

注意： 名前付きデータ型の定義に関するその他の詳細は、11-39 ページの「[名前付きデータ型、REF 定義および PL/SQL OUT バインドの情報](#)」を参照してください。

REF 出力変数の定義

名前付きデータ型と同様に、REF 出力変数の定義も 2 ステップで処理します。第 1 ステップで `OCIDefineByPos()` をコールし、第 2 ステップで `OCIDefineObject()` をコールします。また、名前付きデータ型の場合と同様に、`OCIDefineByPos()` の `dtype` パラメータに `SQLT_REF` データ型定数を渡します。

`SQLT_REF` では、アプリケーションで結果として得るデータを `OCIRef *` 型変数にフェッチすることを示します。第 6 章で説明するとおり、この REF をオブジェクトの確保とナビゲーションの一部として使用します。

注意： REF の定義に関するその他の詳細は、11-39 ページの「[名前付きデータ型、REF 定義および PL/SQL OUT バインドの情報](#)」を参照してください。

名前付きデータ型、REF 定義および PL/SQL OUT バインドの情報

この項には、名前付きデータ型と REF の定義を処理するときに注意する必要がある重要な追加情報について説明します。メモリー割当てとインジケータ変数の使用方法についての参照先も示します。

PL/SQL OUT バインドとは、プレースホルダを PL/SQL ブロックの出力変数にバインドすることです。SQL 文では出力バッファを定義コールで設定しますが、PL/SQL ブロックでは出力バッファをバインド・コールで設定します。詳細は、5-5 ページの「[PL/SQL のプレースホルダのバインド](#)」を参照してください。

- 定義対象のデータ型が `SQLT_NTY` の場合は、`OCIDefineObject()` コールのインジケータ構造体パラメータ (`dvoid ** indpp`) が使用され、スカラー・インジケータは完全に無視されます。
- データ型が `SQLT_REF` の場合、スカラー・インジケータが使用され、`OCIDefineObject()` のインジケータ構造体のパラメータは完全に無視されます。
- インジケータ構造体の使用はオプションです。ユーザーは、`indpp` パラメータの NULL ポインタを `OCIDefineObject()` コールに渡すことができます。これは、フェッチまたは PL/SQL OUT バインド時に、ユーザーは NULL かどうかについての情報に意識しないことを意味します。
- SQL 定義または PL/SQL OUT バインドの場合、プログラマは出力変数またはインジケータのいずれかに対して事前割当済みメモリーを渡すことができます。渡されると、その事前割当て済みメモリーが結果データの格納に使用され、2 次メモリー（アウト・ライン・メモリー）がある場合は、その内容がすべて割当て解除されます。事前割当て済みメモリーは、キャッシュ (`OCIObjectNew()` コールの結果) から取得する必要があります。

注意： キャッシュからではなく、クライアント・アプリケーションで所有するメモリ領域からメモリを割り当てる場合は、オブジェクトにアウト・ライン 2 次メモリがないことを確認する必要があります。

SQLT_NTY 型を使用したオブジェクト定義の場合、オブジェクト・メモリの事前割当てを行うクライアント・アプリケーションでは `OCIObjectNew()` 関数を使用する必要があります。クライアント・アプリケーションでは、`malloc()` を使用する割当てや、スタックへの割当てなど、独自のプライベート・メモリ領域へのオブジェクトの割当ては行わないでください。`OCIObjectNew()` 関数によって、オブジェクト・キャッシュにオブジェクトが割り当てられます。割り当てられたオブジェクトは、`OCIObjectFree()` を使用すると解放できます。`OCIObjectNew()` および `OCIObjectFree()` の詳細は、[第 17 章「OCI のナビゲーションル関数と型関数」](#) を参照してください。

注意： ユーザーがオブジェクト・メモリーを事前割当てせずに、出力変数を NULL ポインタ値に初期化した場合でも、`OCIDefineObject()` の動作に変化はありません。この場合、オブジェクトは OCI ライブラリによって暗黙的にオブジェクト・キャッシュ内に割り当てられます。

- SQL 定義または PL/SQL OUT バインドの場合、ユーザーが出力変数またはインジケータに対して NULL アドレスを渡すと、その変数またはインジケータ用のメモリーは、OCI によって暗黙的に割り当てられます。
- SQLT_NTY 型の出力オブジェクトが（SQL 定義または PL/SQL OUT バインドで）アトミック NULL の場合は、NULL インジケータ構造体のみが（必要であれば暗黙的に）割り当てられてデータが挿入され、オブジェクトがアトミック NULL であることが明示されます。トップレベルのオブジェクト自体は、暗黙的に割り当てられません。
- アプリケーションでは、`OCIObjectFree()` をコールしてインジケータを解放できます。トップレベルのオブジェクトがある場合は（アトミック NULL でないオブジェクトの場合など）、そのトップレベルのオブジェクトを `OCIObjectFree()` で解放すると、インジケータも解放されます。オブジェクトがアトミック NULL である場合は、トップレベルのオブジェクトがないので、インジケータを個別に解放する必要があります。
- `OCIDefineObject()` コールのインジケータ構造体サイズ・ポインタ `indsp`、およびプログラム変数サイズ・ポインタ `pgvsp` は、オプションです。これらのパラメータが不要な場合、ユーザーは NULL を渡すことができます。

配列定義に関する情報

名前付きデータ型または REF の配列定義を行うには、ユーザーはポインタ配列を適切な型のバッファ（事前割当て済みまたはそれ以外）に渡す必要があります。同様に、スカラー・インジケータの配列（SQLT_REF 型）またはポインタ配列を、インジケータ構造体（SQLT_NTY 型）に渡す必要があります。

Oracle C データ型のバインドおよび定義

これまでの章では、OCI のバインド操作と定義操作について説明しました。4-6 ページの「バインド」では、OCI バインド操作の基本について、4-15 ページの「定義」では、OCI 定義操作の基本について説明しています。名前付きデータ型および REF のバインドと定義の詳細は、第 5 章「バインドと定義」を参照してください。

バインドと定義の基本的な機能に関する章では、アプリケーションで SQL 文の入力（バインド）値として、または問合せに対する出力（定義）バッファとしてスカラー変数またはスカラーの配列を使用する方法を示しています。

名前付きデータ型と REF に関する項では、オブジェクトや参照をバインドまたは定義する方法を説明しています。第 10 章「OCI オブジェクト・リレーショナル・プログラミング」では、さらにオブジェクト参照の確保、オブジェクト・ナビゲーションおよび埋込みインスタンスのフェッチについて説明しています。

この項では、この章で説明するデータ型マッピングに従って、個々の属性値のバインドと定義の方法を説明します。

この章で定義する型の 1 つである OCINumber、OCIString などの変数は、一般にアプリケーションで宣言でき、適切なデータ型コードが指定されるかぎり、OCI バインドまたは定義操作で直接使用できます。次の表は、C マッピングとともにバインドおよび定義で利用できるデータ型と、バインドまたは定義コールの *dtv*（データ型コード）で指定する必要がある、OCI 外部データ型のリストです。

表 11-1 バインドおよび定義用のデータ型マッピング

データ型	C マッピング	OCI の外部データ型とコード
Oracle 数値	OCINumber	VARNUM (SQLT_VNU)
Oracle 日付	OCIDate	SQLT_ODT
BLOB	OCILobLocator *	SQLT_BLOB
CLOB、NCLOB	CILobLocator *	SQLTY_LOB
VARCHAR2、NVARCHAR2	OCIString *	SQLT_VST（注意：を参照）
RAW	OCIRaw *	SQLT_LVB（注意：を参照）
CHAR、NCHAR	OCIString *	SQLT_VST

表 11-1 バインドおよび定義用のデータ型マッピング（続き）

データ型	C マッピング	OCI の外部データ型とコード
OBJECT	struct *	名前付きデータ型 (SQLT_NTY)
REF	OCISRef *	REF (SQLT_REF)
VARRAY	OCIArray *	名前付きデータ型 (SQLT_NTY)
NESTED TABLE	OCITable *	名前付きデータ型 (SQLT_NTY)
DATETIME	OCIDateTime *	11-10 ページの「日時および時間隔 (OCIDateTime、OCIInterval)」を参照
INTERVAL	OCIInterval *	11-10 ページの「日時および時間隔 (OCIDateTime、OCIInterval)」を参照

注意： **OCIString *** 型の定義変数にデータをフェッチするには、最初に **OCIStringResize()** ルーチンを使用して文字列のサイズを設定する必要があります。そのためには、記述操作によって選択リスト・データの長さを取得することが必要な場合があります。同様に、**OCIRaw *** も最初に **OCIRawResize()** でサイズを決定する必要があります。

次の項では、C にマッピングしたデータ型の OCI アプリケーションでの使用例を示します。

関連項目： OCI 外部データ型の説明とデータ型コードのリストは、[第 3 章「データ型」](#)を参照してください。

バインドおよび定義の例

この項では、OCI のバインド操作と定義操作で **OCINumber** 型の変数を使用する例を示します。

注意： この項の例は、特定の OCI 作業を実行するのに使用されるコールのフローを説明します。ここでは拡張擬似コードを使用しています。実際の関数名を使用していますが、簡素化のためにパラメータとタイプキャストは一部のみ示します。ハンドルの割当てなど、その他の必要な OCI コールも省略しています。

この例では、次のような `person` というオブジェクト型が作成されていると仮定します。

```
CREATE TYPE person AS OBJECT
(name      varchar2(30),
salary    number);
```

この型を使用して、`person` 型の列を持つ `employees` (従業員) 表を作成します。

```
CREATE TABLE employees
(emp_id    number,
job_title  varchar2(30),
emp        person);
```

Object Type Translator (OTT) では、`person` に対して次の C 構造体と NULL インジケータ構造体が生成されます。

```
struct person
{
    OCIStr * name;
    OCIInt salary;};
typedef struct person person;

struct person_ind
{
    OCIInd _atomic;
    OCIInd name;
    OCIInd salary;};
typedef struct person_ind person_ind;
```

関連項目： OTT の詳細は、[第 14 章「Object Type Translator \(OTT\)」](#)を参照してください。

`employees` 表は値が入り、OCI アプリケーションは `person` の変数を宣言するとします。

```
person *my_person;
```

また、次のように、`SELECT` 文によってオブジェクトがその変数にフェッチされたとします。

```
text *mystmt = (text *) "SELECT person FROM employees
                        WHERE emp.name='Andrea'";
```

この場合は、適切な名前付きデータ型の OCI 定義コールを使用して、`my_person` をこの文の出力変数として定義する必要があります。詳細は、5-21 ページの「[拡張定義操作](#)」で説明されています。この文を実行すると、Andrea という名の `person` オブジェクトが、`my_person` 変数に取り出されます。

オブジェクトが `my_person` に取り出されると、OCI アプリケーションで `my_person` の名前や給与などの属性にアクセスできるようになります。

アプリケーションでは、別の従業員の給与を、次のように **Andrea** と同様に更新します。

```
text *updstmt = (text *) "UPDATE employees SET emp.salary = :newsal
                        WHERE emp.name = 'MONGO'";
```

`my_person->salary` に格納されている **Andrea** の給与は、プレースホルダ `:newsal` にバインドされ、`VARNUM` の外部データ型（データ型コード=6）をバインド操作に指定します。

```
OCIBindByName(..., ":newsal", ..., &my_person->salary, ..., 6, ...);
OCIStmtExecute(..., updstmt, ...);
```

文を実行すると、データベースにある **Mongo** の給与が、`my_person` に格納されているように、**Andrea** の給与と等しくなります。

その反対に、**Mongo** の給与をデータベースに問い合わせしてから給与に必要な割当てを行うことにより、アプリケーションで **Andrea** の給与を **Mongo** の給与と同額になるよう更新できます。

```
text *selstmt = (text *) "SELECT emp.salary FROM employees
                        WHERE emp.name = 'MONGO'";
```

```
OCINumber mongo_sal;
...
OCIDefineByPos(..., 1, ..., &mongo_sal, ..., 6, ...);
OCIStmtExecute(..., selstmt, ...);
OCINumberAssign(..., &mongo_sal, &my_person->salary);
```

この場合には、アプリケーションで **OCINumber** の型の出力変数を宣言し、それを定義のステップで使用します。この場合は、位置 1 の出力変数を定義し、適切なデータ型コード（`VARNUM` の場合は 6）を使用します。

`mongo_sal` という **OCINumber** に給与の値をフェッチし、OCI 関数 `OCINumberAssign()` を使用して、現在キャッシュ内にある **Andrea** オブジェクトのコピーに新しい給与を割り当てます。データベース内にあるデータを変更するには、変更内容をサーバーにフラッシュする必要があります。

給与更新の例

前の項では、Oracle データ型がバインド操作と定義操作に与える柔軟性に関して、その要点をいくつか説明しました。この項では、同じ操作を異なる方法でどのように実行するかを説明します。OCI アプリケーションにおける Oracle の新しいデータ型の様々な使用方法を示すことが目的です。

この項の例は、特定の OCI 作業を実行するのに使用されるコールのフローを説明します。ここでは拡張擬似コードを使用しています。実際の関数名を使用していますが、簡素化のためにパラメータとタイプキャストは一部のみ示します。ハンドルの割当てなど、その他の必要な OCI コールも省略しています。

シナリオ

これから示す例のシナリオは、次のとおりです。

1. ある病院の `employees` 表に、`BRUCE` という従業員が存在します。前の項の `person` 型および `employees` 表の作成文を参照してください。
2. Bruce の現在の職種は、`RADIOLOGIST` です。
3. Bruce は、`RADIOLOGY_CHIEF` に昇進したため、それに伴った昇給があります。
4. 次に示すように、病院の給与はドル単位の整数値とし、職種に応じて設定し、`salaries` という表に格納します。

```
CREATE TABLE salaries
(job_title  varchar2(20),
 salary    integer);
```

5. Bruce の給与を昇進にあわせて更新します。

前述の作業を実行するには、アプリケーションで `RADIOLOGY_CHIEF` に該当する給与を `salaries` 表から取り出し、Bruce の給与を更新します。別のステップで、Bruce の新しい職種と変更済みオブジェクトをデータベースに書き込みます。

次のように `person` 型の変数を宣言したとします。

```
person * my_person;
```

そして、Bruce に対応するオブジェクトをこの変数にフェッチしたとします。この場合に給与更新を実行するには、次の各項に示す 3 通りの方法があります。

方法 1 – フェッチ、変換、割当て

この例では次の方法をとります。

1. 整変数を使用して従来の OCI 定義を行い、データベースから新しい給与を取り出します。
2. 整数を `OCINumber` に変換します。
3. 新しい給与を Bruce に割り当てます。

```
#define INT_TYPE 3          /* datatype code for sword integer define */

text *getsal = (text *) "SELECT salary FROM salaries
                        WHERE job_title='RADIOLOGY_CHIEF'";

sword  new_sal;
OCINumber  orl_new_sal;
...
OCIDefineByPos(...,1,...,new_sal,...,INT_TYPE,...);
/* define int output */
```

```
OCIStmtExecute(..., getsal, ...);
/* get new salary as int */
OCINumberFromInt(..., new_sal, ..., &orl_new_sal);
/* convert salary to OCINumber */
OCINumberAssign(..., &orl_new_sal, &my_person->salary);
/* assign new salary */
```

方法 2 – フェッチ、割当て

この方法では、方法 1 のステップを 1 つ省略します。

1. **OCINumber** 型の出力変数を定義します。そのため値を取り出した後の変換は必要ありません。
2. 新しい給与を **Bruce** に割り当てます。

```
#define VARNUM_TYPE 6 /* datatype code for defining VARNUM */

text *getsal = (text *) "SELECT salary FROM salaries
                          WHERE job_title='RADIOLOGY_CHIEF'";

OCINumber orl_new_sal;
...
OCIDefineByPos(..., 1, ..., orl_new_sal, ..., VARNUM_TYPE, ...);
/* define OCINumber output */
OCIStmtExecute(..., getsal, ...); /* get new salary as OCINumber */
OCINumberAssign(..., &orl_new_sal, &my_person->salary);
/* assign new salary */
```

方法 3 – 直接フェッチ

この方法では、操作全体を 1 回の定義とフェッチで実行します。出力変数を介在させる必要はなく、データベースから取り出した値を、キャッシュに格納されているオブジェクトの給与属性に直接フェッチします。

1. **Bruce** はオブジェクト・キャッシュ内に確保されているため、その給与属性の位置を定義変数として使用し、その変数に直接実行し、フェッチを行います。

```
#define VARNUM_TYPE 6 /* datatype code for defining VARNUM */

text *getsal = (text *) "SELECT salary FROM salaries
                          WHERE job_title='RADIOLOGY_CHIEF'";
...
OCIDefineByPos(..., 1, ..., &my_person->salary, ..., VARNUM_TYPE, ...);
/* define bruce's salary in cache as output variable */
OCIStmtExecute(..., getsal, ...);
/* execute and fetch directly */
```

まとめと注意

この3つの例に示したように、C データ型を使用すると柔軟性のあるバインドと定義を実行できます。たとえば、整数をフェッチし、**OCINumber** に変換して操作できます。この方法では、**OCINumber** を中間的な変数として使用し、問合せの結果を格納できます。あるいは、目的のオブジェクトの **OCINumber** 属性にデータを直接フェッチすることもできます。

注意： OCI のこれらすべての例では、問合せの実行前に出力変数が定義されている場合、結果として得られるデータは、出力バッファに直接ブリフェッチされることを覚えておいてください。

この3つの例で、変更がデータベースに確実に永続的に書き込まれるためには、追加ステップが必要です。この追加ステップには、SQL の UPDATE コールと OCI トランザクションのコミット・コールを含めることができます。

これらの例ではすべて定義操作を処理していますが、同じような状況がバインドにも当てはまります。

同様に、これらの例では **OCINumber** 型のみを扱いましたが、この章でこれから説明する他の Oracle の C 型についても、同じような種々の操作を実行できます。

SQLT_NTY のバインドおよび定義の例

次の部分的なコードは、OCIBindObject() や OCIDefineObject() などの SQLT_NTY バインド・コールおよび SQLT_NTY 定義コールの使用方法を示しています。それぞれの例で、前に定義した SQL 文を処理します。

バインドの例

```
/*
** This example performs a SQL insert statement
*/
void insert(envhp, svchp, stmthp, errhp, insstmt, nrows)
OCIEnv *envhp;
OCISvcCtx *svchp;
OCIStmt *stmthp;
OCIError *errhp;
text *insstmt;
ub2 nrows;
{
    orttdo *addr_tdo = NULLP(orttdo);
    address addr;
    null_address naddrs;
    address *addr = &addr;
    null_address *naddr = &naddrs;
```

```

sword custno =300;
OCIBind *bndlp, *bnd2p;
ub2 i;

/* define the application request */
checkerr(errhp, OCISmtPrepare(stmthp, errhp, (text *) insstmt,
    (ub4) strlen((char *)insstmt),
    (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

/* bind the input variable */
checkerr(errhp, OCIBindByName(stmthp, &bndlp, errhp, (text *) ":custno",
    (sb4) -1, (dvoid *) &custno,
    (sb4) sizeof(sword), SQLENT_INT,
    (dvoid *) 0, (ub2 *)0, (ub2 *)0, (ub4) 0, (ub4 *) 0,
    (ub4) OCI_DEFAULT));

checkerr(errhp, OCIBindByName(stmthp, &bnd2p, errhp, (text *) ":addr",
    (sb4) -1, (dvoid *) 0,
    (sb4) 0, SQLENTY, (dvoid *) 0, (ub2 *)0, (ub2 *)0,
    (ub4) 0, (ub4 *) 0, (ub4) OCI_DEFAULT));

checkerr(errhp, OCISTypeByName(envhpx, errhp, svchpx, (const text *)
    SCHEMA, (ub4) strlen((char *)SCHEMA), (const text *)
    "ADDRESS_VALUE", (ub4) strlen((char *)"ADDRESS_VALUE"),
    OCI_DURATION_SESSION, &addr_tdo));

if(!addr_tdo)
{
    printf("Null tdo returned\n");
    goto done_insert;
}

checkerr(errhp, OCIBindObject(bnd2p, errhp, addr_tdo, (dvoid **) &addr,
    (ub4 *) 0, (dvoid **) &naddr, (ub4 *) 0));

```

定義の例

```

/*
** This example executes a SELECT statement from a table which includes
** an object.
*/

void selectval(envhp, svchp, stmthp, errhp)
OCIEnv *envhp;
OCISvcCtx *svchp;
OCISmt *stmthp;
OCIError *errhp;

```



```

{
    orttdo *addr_tdo = NULLP(orttdo);
    OCIDefine *defn1p, *defn2p;
    address *addr = (address *)NULL;
    sword custno =0;
    sb4 status;

    /* define the application request */
    checkerr(errhp, OCISTmtPrepare(stmthp, errhp, (text *) selvalstmt,
                                   (ub4) strlen((char *)selvalstmt),
                                   (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT));

    /* define the output variable */
    checkerr(errhp, OCIDefineByPos(stmthp, &defn1p, errhp, (ub4) 1, (dvoid *)
                                   &custno, (sb4) sizeof(sword), SQLT_INT, (dvoid *) 0, (ub2 *)0,
                                   (ub2 *)0, (ub4) OCI_DEFAULT));

    checkerr(errhp, OCIDefineByPos(stmthp, &defn2p, errhp, (ub4) 2, (dvoid *)
                                   0, (sb4) 0, SQLT_NTY, (dvoid *) 0, (ub2 *)0,
                                   (ub2 *)0, (ub4) OCI_DEFAULT));

    checkerr(errhp, OCITYPEByName(envhpx, errhp, svchpx, (const text *)
                                   SCHEMA, (ub4) strlen((char *)SCHEMA), (const text *)
                                   "ADDRESS_VALUE", (ub4) strlen((char *)"ADDRESS_VALUE"), OROODTSES,
                                   &addr_tdo));

    if(!addr_tdo)
    {
        printf("NULL tdo returned\n");
        goto done_selectval;
    }

    checkerr(errhp, OCIDefineObject(defn2p, errhp, addr_tdo, (dvoid **)
                                   &addr, (ub4 *) 0, (dvoid **) 0, (ub4 *) 0));

    checkerr(errhp, OCISTmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                                   (OCISnapshot *) NULL, (OCISnapshot *) NULL, (ub4) OCI_DEFAULT));

```

ダイレクト・パス・ロード

ダイレクト・パス・ロード関数は、データを外部ファイルから表とパーティションにロードするために使用します。

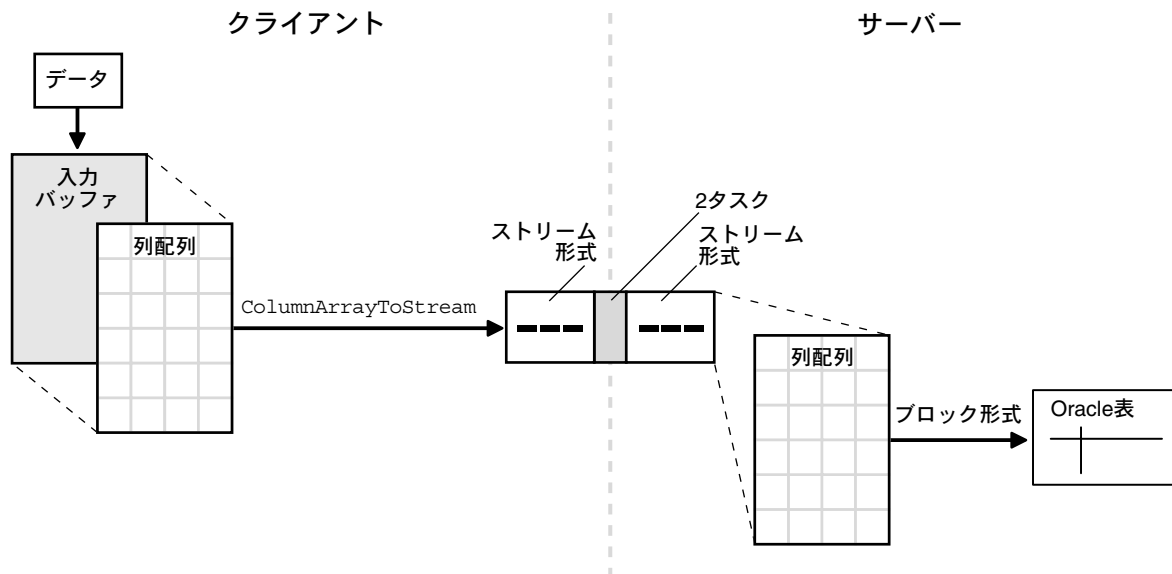
この章は、次の項目で構成されています。

- [ダイレクト・パス・ロードの概要](#)
- [オブジェクト型のダイレクト・パス・ロード](#)
- [ピース単位のダイレクト・パス・ロード](#)
- [ダイレクト・パス・コンテキスト・ハンドルとオブジェクト型の属性](#)

ダイレクト・パス・ロードの概要

ダイレクト・パス・ロード・インタフェースを使用すると、OCI アプリケーションから Oracle データベース・サーバーのダイレクト・パス・ロード・エンジンにアクセスし、Oracle SQL*Loader ユーティリティの関数を実行できます。この機能によって、データを外部ファイルから表またはパーティション表のパーティションのいずれかにロードできます。

図 12-1 ダイレクト・パス・ロード



OCI ダイレクト・パス・ロード・インタフェースには、複数行のデータが含まれたダイレクト・パス・ストリームをロードすることによって複数の行をロードする機能があります。

ダイレクト・パス API を使用するには、クライアント・アプリケーションで次のステップを実行します。

1. OCI 初期化を実行します。
2. ダイレクト・パス・コンテキスト・ハンドルを割り当てて、属性を設定します。
3. ロードするオブジェクト（表、パーティションまたはサブパーティション）の名前を指定します。
4. オブジェクトの列の外部データ型を記述します。
5. ダイレクト・パス・インタフェースを準備します。
6. 1 列以上の列配列を割り当てます。

7. 1つ以上のダイレクト・パス・ストリームを割り当てます。
8. 列配列内のエントリを、各列に対する入力データ値を指し示すように設定します。
9. 列配列をダイレクト・パス・ストリーム形式に変換します。
10. ダイレクト・パス・ストリームをロードします。
11. 発生したエラーをすべて取り出します。
12. ダイレクト・パス終了関数をコールします。
13. ハンドルおよびデータ構造を解放します。
14. サーバーから切断します。

ステップ 8～11 は、ロード対象のデータに従って繰り返すことができます。

ダイレクト・ロード操作では、ロードされているオブジェクトをロックしてそのオブジェクトに DML が実行されないようにする必要があります。オブジェクトのロード中、問合せはロック解除され許可されていることに注意してください。DML ロックのモード、および取得される DML ロックの種類は、OCI_DIRPATH_PARALLEL_LOAD の指定によって決まります。また、表全体をロードするのか、パーティションまたはサブパーティションをロードするのかによっても異なります。

関連項目： OCI_DIRPATH_PARALLEL_LOAD の詳細は、16-124 ページの [OCIDirPathPrepare\(\)](#) を参照してください。

- 表のロードの場合、OCI_DIRPATH_PARALLEL_LOAD オプションの設定とその結果は、次のとおりです。
 - FALSE の場合、表 DML X-Lock が取得されます。
 - TRUE の場合、表 DML S-Lock が取得されます。
- パーティションのロードの場合、OCI_DIRPATH_PARALLEL_LOAD オプションの設定とその結果は、次のとおりです。
 - FALSE の場合、表 DML SX-Lock とパーティション DML X-Lock が取得されます。
 - TRUE の場合、表 DML SS-Lock とパーティション DML S-Lock が取得されます。

ダイレクト・パス・ロードでサポートされるデータ型

ダイレクト・パス・ロード操作では、スカラー列に対して有効な外部データ型は次のとおりです。

- SQLT_CHR
- SQLT_DAT
- SQLT_INT
- SQLT_UIN
- SQLT_FLT
- SQLT_BIN
- SQLT_NUM
- SQLT_PDN
- SQLT_DATE
- SQLT_TIMESTAMP
- SQLT_TIMESTAMP_TZ
- SQLT_TIMESTAMP_LTZ
- SQLT_INTERVAL_YM
- SQLT_INTERVAL_DS

次の外部オブジェクトのデータ型がサポートされています。

- SQLT_NTY — 列オブジェクト (FINAL と NOT FINAL) および SQL 文字列の列
- SQLT_REF — REF 列 (FINAL と NOT FINAL)

次の表の型がサポートされています。

- NESTED TABLE
- オブジェクト表 (FINAL と NOT FINAL)

関連項目： 列のデータ型の設定または取出しの詳細は、A-72 ページの「[OCI_ATTR_DATA_TYPE](#)」を参照してください。データ型の詳細は、[表 3-2 「外部データ型とそのコード」](#)を参照してください。

ダイレクト・パス・ハンドル

ダイレクト・パス・ロードは、ダイレクト・パス配列の挿入操作に対応しています。ダイレクト・パス・ロード・インタフェースでは、次のハンドルを使用してロードされるオブジェクトおよび操作するデータの指定が記録されます。

- ダイレクト・パス・コンテキスト
- ダイレクト・パス関数コンテキスト
- ダイレクト・パス列配列
- ダイレクト・パス関数コンテキストの列配列
- ダイレクト・パス・ストリーム

関連項目： A-58 ページの「[ダイレクト・パス・ロード・ハンドル属性](#)」およびこの後のダイレクト・パス属性に関する説明を参照してください。

ダイレクト・パス・コンテキスト

このハンドルは、ロード対象の各オブジェクト（表またはパーティション表のパーティションのいずれか）に対して割り当てる必要があります。**OCIDirPathCtx** ハンドルは、**OCIDirPathFuncCtx**、**OCIDirPathColArray** および **OCIDirPathStream** ハンドルの親ハンドルであるため、**OCIDirPathCtx** ハンドルを解放すると、その子ハンドルも解放されます（ただし、親ハンドルの解放前に、子ハンドルを個別に解放するコーディングをお勧めします）。

ダイレクト・パス・コンテキストは、**OCIHandleAlloc()** を使用して割り当てます。

```
OCIEnv *envp;  
OCIDirPathCtx *dpctx;  
sword error;  
error = OCIHandleAlloc((dvoid *)envp, (dvoid **)&dpctx,  
                      OCI_HTYPE_DIRPATH_CTX, 0, (dvoid **)0);
```

ダイレクト・パス・コンテキストの親ハンドルは、常に環境ハンドルです。ダイレクト・パス・コンテキストは、**OCIHandleFree()** を使用して解放します。

```
error = OCIHandleFree(dpctx, OCI_HTYPE_DIRPATH_CTX);
```

OCI ダイレクト・パス関数コンテキスト

関連項目： サポートされるデータ型の詳細は、『Oracle9i アプリケーション開発者ガイド - オブジェクト・リレーショナル機能』を参照してください。

OCIDirPathFuncCtx 型のこのハンドルは、次の名前付き型と REF 列の記述に使用します。

- 列オブジェクト。このオブジェクト内の関数コンテキストでは、オブジェクト型が記述され、デフォルト・コンストラクタとして使用されて、オブジェクトとそのコンストラクタのオブジェクト属性を作成します。
- REF 列。この列内の関数コンテキストでは、行オブジェクトの参照元である単一のオブジェクト表（オプション） および行オブジェクトを識別する REF 引数が記述されます。
- SQL 文字列の列。この列内の関数コンテキストでは、列にロードする値を計算するための SQL 文字列とその引数が記述されます。

関数コンテキストの割当てを示すには、次の例のように **OCI_HTYPE_DIRPATH_FN_CTX** ハンドル型を **OCIHandleAlloc()** に渡します。

```
OCIDirPathCtx *dpctx;          /* direct path context */
OCIDirPathFuncCtx *dpfnctx; /* direct path function context */

OCIHandleAlloc((dvoid *)dpctx,
               (dvoid **)&dpfnctx,
               (ub4)OCI_HTYPE_DIRPATH_FN_CTX,
               (size_t)0, (dvoid **)0);
```

ダイレクト・パス関数コンテキストの親ハンドルは、常にダイレクト・パス・コンテキスト・ハンドルであることに注意してください。ダイレクト・パス関数コンテキスト・ハンドルは、次のコードで解放します。

```
error = OCIHandleFree(dpfnctx, OCI_HTYPE_DIRPATH_FN_CTX);
```

ダイレクト・パス列配列およびダイレクト・パス関数列配列

ダイレクト・パス・インタフェースに行配列を表示するには、ダイレクト・パス列配列およびダイレクト・パス関数列配列ハンドルを使用します。行は、列の値、列の長さおよび列フラグの 3 つの配列によって表します。列配列で使用するメソッドには、配列ハンドルの割当て、および配列のエントリに対応する値の設定や取得があります。

これらのハンドルは同じデータ構造体 **OCIDirPathColArray** を共有します。ただし、この 2 つの列配列ハンドルは、親ハンドルとハンドル型が異なります。

ダイレクト・パス列配列ハンドルは、**OCIHandleAlloc()** を使用して割り当てます。次のコード・フラグメントは、ダイレクト・パス列配列ハンドルの明示的な割当てを示します。

```
OCIDirPathCtx *dpctx;          /* direct path context */
OCIDirPathColArray *dpca; /* direct path column array */
sword error;
error = OCIHandleAlloc((dvoid *)dpctx, (dvoid **)&dpca,
                       OCI_HTYPE_DIRPATH_COLUMN_ARRAY,
                       (size_t)0, (dvoid **)0);
```


ダイレクト・パス列配列ハンドルは、OCIHandleFree() を使用して解放します。

```
error = OCIHandleFree(dpca, OCI_HTYPE_DIRPATH_COLUMN_ARRAY);
```

ダイレクト・パス関数列配列ハンドルも、ほとんど同じ方法で割り当てます。

```
OCIDirPathFuncCtx *dpfnctx; /* direct path fuction context */
OCIDirPathColArray *dpfnca; /* direct path function column array */
sword error;
error = OCIHandleAlloc((dvoid *)dpfnctx, (dvoid **)&dpfnca,
                      (ub4)OCI_HTYPE_DIRPATH_FN_COL_ARRAY,
                      (size_t)0, (dvoid **)0);
```

ダイレクト・パス関数列配列ハンドルは、OCIHandleFree() を使用して解放します。

```
error = OCIHandleFree(dpfnca, OCI_HTYPE_DIRPATH_FN_COL_ARRAY);
```

OCIDirPathColArray ハンドルの解放により、そのハンドルに関連付けられた列配列も解放されます。

ダイレクト・パス・ストリーム

変換操作 **OCIDirPathColArrayToStream()** およびロード操作 **OCIDirPathLoadStream()** には、このハンドルを使用します。

ダイレクト・パス・ストリーム・ハンドルは、クライアントが OCIHandleAlloc() を使用して割り当てます。**OCIDirPathStream** ハンドルの構造体は、フォーム（バッファ、バッファ長）内のペアとみなすことができます。

ダイレクト・パス・ストリームは、Oracle 表データの線形表現です。変換操作は、常にストリームの最後に追加されます。ロード操作は、常にストリームの最初に起動します。ストリームのロードが正常終了した後、OCIDirPathStreamReset() をコールしてストリームをリセットする必要があります。

次のコードは、OCIHandleAlloc() を使用して割り当てられるダイレクト・パス・ストリーム・ハンドルの例です。親ハンドルは、常に **OCIDirPathCtx** ハンドルです。

```
OCIDirPathCtx *dpctx; /* direct path context */
OCIDirPathStream *dpstr; /* direct path stream */
sword error;
error = OCIHandleAlloc((dvoid *)dpctx, (dvoid **)&dpstr,
                      OCI_HTYPE_DIRPATH_STREAM, (size_t)0, (dvoid **)0);
```

ダイレクト・パス・ストリーム・ハンドルは、OCIHandleFree() を使用して解放します。

```
error = OCIHandleFree(dpstr, OCI_HTYPE_DIRPATH_STREAM);
```

OCIDirPathStream ハンドルの解放によって、そのハンドルに関連付けられているストリーム・バッファも解放されます。

ダイレクト・パス・インタフェースの関数

この項にリストされた関数は、ダイレクト・パス・ロード・インタフェースとともに使用します。

関連項目： 各関数の詳細は、16-110 ページの「[ダイレクト・パス・ロード関数](#)」を参照してください。

ダイレクト・パス・コンテキストの操作は、[表 12-1「ダイレクト・パス・コンテキストの関数」](#)の関数で実行します。

表 12-1 [ダイレクト・パス・コンテキストの関数](#)

関数	用途
OCIDirPathAbort()	ダイレクト・パス処理を異常終了します。
OCIDirPathDataSave()	データのセーブポイントを実行します。
OCIDirPathFinish()	ロードされたデータをコミットします。
OCIDirPathFlushRow()	データベース・サーバーから行の一部分をフラッシュします。
OCIDirPathLoadStream()	ダイレクト・パス・ストリーム形式に変換されたデータをロードします。
OCIDirPathPrepare()	行の変換またはロードを行うために、ダイレクト・パス・インタフェースを準備します。

ダイレクト・パス列配列の操作は、[表 12-2「ダイレクト・パス列配列の関数」](#)の関数で実行します。

表 12-2 [ダイレクト・パス列配列の関数](#)

関数	用途
OCIDirPathColArrayEntryGet()	列配列内の特定のエントリを取得します。
OCIDirPathColArrayEntrySet()	列配列内の特定のエントリを特定の値に設定します。
OCIDirPathColArrayRowGet()	特定の行番号の基本行ポインタを取得します。
OCIDirPathColArrayReset()	行配列の状態をリセットします。
OCIDirPathColArrayToStream()	列配列形式からダイレクト・パス・ストリーム形式に変換します。

ダイレクト・パス・ストリームの操作は、ダイレクト・ストリームの状態をリセットする [OCIDirPathStreamReset\(\)](#) 関数で実行します。

ダイレクト・パス・ロード・インタフェースに関する制限と制約

ダイレクト・パス・ロード・インタフェースには、SQL*Loader と同じ次の制限があります。

- トリガーがサポートされていません。
- 参照整合性制約がサポートされていません。
- クラス化表がサポートされていません。
- リモート・オブジェクトのロードがサポートされていません。
- LONG は最後に指定してください。
- LOB、オブジェクトまたはコレクションを戻す SQL 文字列がサポートされていません。
- VARRAY 列のロードがサポートされていません。
- すべてのパーティション列は、LOB の前に指定する必要があります。これは、LOB をパーティションに書き込む前に、書き込み先のパーティションを確認する必要があるためです。

スカラー列に対するダイレクト・パス・ロードの例

ダイレクト・パス・ロードの例で使用されているデータ構造

例では、次のデータ構造が使用されています。

```
/* load control structure */
struct loadctl
{
    ub4          nrow_ctl;          /* number of rows in column array */
    ub2          ncol_ctl;          /* number of columns in column array */
    OCIEnv        *envhp_ctl;       /* environment handle */
    OCIServer     *srvhp_ctl;       /* server handle */
    OCIError      *errhp_ctl;       /* error handle */
    OCIError      *errhp2_ctl;      /* another error handle */
    OCISvcCtx     *svchp_ctl;       /* service context */
    OCISession    *authp_ctl;       /* authentication context */
    OCIPParam     *colLstDesc_ctl;  /* column list parameter handle */
    OCIDirPathCtx *dpctx_ctl;       /* direct path context */
    OCIDirPathColArray *dpca_ctl;   /* direct path column array handle */
    OCIDirPathStream *dpstr_ctl;    /* direct path stream handle */
    ub1          *buf_ctl;          /* pre-alloc'd buffer for out-of-line data */
    ub4          bufisz_ctl;        /* size of buf_ctl in bytes */
    ub4          bufoff_ctl;        /* offset into buf_ctl which is not in use */
    ub4          *otor_ctl;         /* Offset to Recnum mapping */
    ub1          *inbuf_ctl;        /* buffer for input records */
    struct pctx   pctx_ctl;         /* partial field context */
};
```

demo ディレクトリのヘッダー・ファイル `cdemodp.h` で、次の構造体を定義します。

```
#ifndef cdemodp_ORACLE
# define cdemodp_ORACLE

# include <oratypes.h>

# ifndef externdef
#  define externdef
# endif

/* External column attributes */
struct col
{
    text *name_col;                /* column name */
    ub2  id_col;                   /* column load id */
    ub2  exttyp_col;               /* external type */
    text *datemask_col;            /* datemask, if applicable */
    ub1  prec_col;                /* precision, if applicable */
    sb1  scale_col;               /* scale, if applicable */
    ub2  csid_col;                /* character set id */
    ub1  date_col;                /* is column a chrdate or date? 1=TRUE. 0=FALSE */
};

/* Input field descriptor
 * For this example (and simplicity),
 * fields are strictly positional.
 */
struct fld
{
    ub4  begpos_fld;               /* 1-based beginning position */
    ub4  endpos_fld;              /* 1-based ending position */
    ub4  maxlen_fld;              /* max length for out of line field */
    ub4  flag_fld;
#define FLD_INLINE 0x1
#define FLD_OUTOFFLINE 0x2
#define FLD_STRIP_LEAD_BLANK 0x4
#define FLD_STRIP_TRAIL_BLANK 0x8
};

struct tbl
{
    text *owner_tbl;              /* table owner */
    text *name_tbl;               /* table name */
    text *subname_tbl;            /* subname, if applicable */
    ub2  ncol_tbl;                /* number of columns in col_tbl */
    text *dflddatemask_tbl;       /* table level default date mask */
    struct col *col_tbl;          /* column attributes */
};
```

```

    struct fld  *fld_tbl;                                /* field descriptor */
    ub1         parallel_tbl;                            /* parallel: 1 for true */
    ub1         nolog_tbl;                               /* no logging: 1 for true */
    ub4         xfrsz_tbl;                               /* transfer buffer size in bytes */
};

struct sess                                /* options for a direct path load session */
{
    text        *username_sess;                    /* user */
    text        *password_sess;                    /* password */
    text        *inst_sess;                        /* remote instance name */
    text        *outfn_sess;                       /* output filename */
    ub4         maxreclen_sess;                    /* max size of input record in bytes */
};
#endif                                     /* cdemodp_ORACLE */

```

スカラー列に対するダイレクト・パス・ロードの例の概略

次のコードは、OCI ダイレクト・パス・インタフェースの使用方法的例です。ただし、一部省略されています。

`init_load` 関数は、`tblp` で記述された表上で、ダイレクト・パス API を使用してダイレクト・パス・ロードを実行します。`ctlp` によって指定された `loadctl` 構造体の環境およびサービス・コンテキストは、適切に初期化されています。サーバーへの接続は確立されています。

```

STATICF void
init_load(ctlp, tblp)
struct loadctl *ctlp;
struct tbl     *tblp;
{
    struct col  *colp;
    struct fld  *fldp;
    sword       ociret;                                /* return code from OCI calls */
    OCIDirPathCtx *dpctx;                              /* direct path context */
    OCIPParam   *colDesc;                              /* column parameter descriptor */
    ub1         parmtyp;
    ub1         *timestamp = (ub1 *)0;
    ub4         size;
    ub4         i;
    ub4         pos;
}

```

```
/* allocate and initialize a direct path context */
OCI_CHECK(ctlp->envhp_ctl, OCI_HTYPE_ENV, ociret, ctlp,
          OCIHandleAlloc((dvoid *)ctlp->envhp_ctl,
                          (dvoid **)&ctlp->dpctx_ctl,
                          (ub4)OCI_HTYPE_DIRPATH_CTX,
                          (size_t)0, (dvoid **)0));

dpctx = ctlp->dpctx_ctl;                                /* shorthand */

OCI_CHECK(ctlp->errhp_ctl, OCI_HTYPE_ERROR, ociret, ctlp,
          OCIAttrSet((dvoid *)dpctx, (ub4)OCI_HTYPE_DIRPATH_CTX,
                     (dvoid *)tblp->name_tbl,
                     (ub4)strlen((const char *)tblp->name_tbl),
                     (ub4)OCI_ATTR_NAME, ctlp->errhp_ctl));
...
```

OCI_ATTR_SUB_NAME、OCI_ATTR_SCHEMA_NAME など、その他の属性も設定します。属性を設定した後、ロードの準備をします。

```
OCI_CHECK(ctlp->errhp_ctl, OCI_HTYPE_ERROR, ociret, ctlp,
          OCIDirPathPrepare(dpctx, ctlp->svchp_ctl, ctlp->errhp_ctl));
```

列配列とストリーム・ハンドルの割当て

ダイレクト・パス・コンテキスト・ハンドルは、列配列ハンドルおよびストリーム・ハンドルの親ハンドルです。また、エラーは、ダイレクト・パス・コンテキストに関連付けられている環境ハンドルとともに戻されます。

```
OCI_CHECK(ctlp->envhp_ctl, OCI_HTYPE_ENV, ociret, ctlp,
          OCIHandleAlloc((dvoid *)ctlp->dpctx_ctl, (dvoid **)&ctlp->dpca_ctl,
                          (ub4)OCI_HTYPE_DIRPATH_COLUMN_ARRAY,
                          (size_t)0, (dvoid **)0));

OCI_CHECK(ctlp->envhp_ctl, OCI_HTYPE_ENV, ociret, ctlp,
          OCIHandleAlloc((dvoid *)ctlp->dpctx_ctl, (dvoid **)&ctlp->dpstr_ctl,
                          (ub4)OCI_HTYPE_DIRPATH_STREAM,
                          (size_t)0, (dvoid **)0));
```

行数と列数の取得

割り当てた列配列から行数および列数を取得します。

```
OCI_CHECK(ctlp->errhp_ctl, OCI_HTYPE_ERROR, ociret, ctlp,
          OCIAttrGet(ctlp->dpca_ctl, (ub4)OCI_HTYPE_DIRPATH_COLUMN_ARRAY,
                     &ctlp->nrow_ctl, 0, OCI_ATTR_NUM_ROWS,
                     ctlp->errhp_ctl));
```

```
OCI_CHECK(ctlp->errhp_ctl, OCI_HTYPE_ERROR, ociret, ctlp,
          OCIAAttrGet(ctlp->dpca_ctl, (ub4)OCI_HTYPE_DIRPATH_COLUMN_ARRAY,
                      &ctlp->ncol_ctl, 0, OCI_ATTR_NUM_COLS,
                      ctlp->errhp_ctl));
```

入力データ・フィールドの設定

入力データ・フィールドを対応するデータ列に設定します。

```
OCIDirPathColArrayEntrySet(ctlp->dpca_ctl, ctlp->errhp_ctl,
                           rowoff, colp->id_col,
                           cval, clen, cflg));
```

列配列の状態のリセット

前の変換を続ける必要がある場合、または行にデータを追加する場合は、列配列の状態をリセットします。

```
(void) OCIDirPathColArrayReset(ctlp->dpca_ctl, ctlp->errhp_ctl);
```

ストリームの状態のリセット

ストリームの状態をリセットして、新しいストリームを開始します。リセットしなかった場合は、ストリーム内のデータが既存のデータの末尾に追加されます。

```
(void) OCIDirPathStreamReset(ctlp->dpstr_ctl, ctlp->errhp_ctl);
```

データのストリーム形式への変換

データの入力後、列配列内のデータをストリーム形式に変換し、不正なレコードを削除します。

```
ocierr = OCIDirPathColArrayToStream(ctlp->dpca_ctl, ctlp->dpctx_ctl,
                                     ctlp->dpstr_ctl, ctlp->errhp_ctl,
                                     rowcnt, startoff);
```

ストリームのロード

ストリームを生成した列配列のオフセット情報とともに、ストリーム・ハンドルに対するストリーム内の位置は変わりません。ストリーム形式への変換が終わると、そのデータはストリームの末尾に追加されます。ストリームは、適切なタイミングでコール元がリセットする必要があります。エラーの発生時には、位置は次の行に移動します。ただし、最後の列でエラーが発生した場合は、ストリームの末尾に移動します。次の行がある場合は、次の OCIDirPathLoadStream() コールが開始されます。OCIDirPathLoadStream() がコールされ、ストリームの末尾に到達した場合は、OCI_NO_DATA が戻されます。

```
ocierr = OCIDirPathLoadStream(ctlp->dpctx_ctl, ctlp->dpstr_ctl,
                              ctlp->errhp_ctl);
```

ダイレクト・パス・ロードの終了

```
OCIDirPathFinish(ctlp->dpctx_ctl, ctlp->errhp_ctl);
```

ダイレクト・パス・ハンドルの解放

割り当てられているすべてのダイレクト・パス・ハンドルを解放します。親のダイレクト・パス・コンテキスト・ハンドルの解放前に、ダイレクト・パス列配列ハンドルとストリーム・ハンドルが解放されます。

```
ociret = OCIHandleFree((dvoid *)ctlp->dpca_ctl,  
                        OCI_HTYPE_DIRPATH_COLUMN_ARRAY);  
ociret = OCIHandleFree((dvoid *)ctlp->dpstr_ctl,  
                        OCI_HTYPE_DIRPATH_STREAM);  
ociret = OCIHandleFree((dvoid *)ctlp->dpctx_ctl,  
                        OCI_HTYPE_DIRPATH_CTX);
```

OCI のダイレクト・パス・ロードでの日付キャッシュの使用

表に格納するためにデータ型変換が必要な Oracle 日付およびタイムスタンプの値をロードする場合は、日付キャッシュ機能を使用するとパフォーマンスが向上します。

この機能は、同じ日付が入力値として繰返しロードされる場合に使用します。日付変換は（特に複数の日付列がロードされる場合）、非常に時間がかかり、ロード時間の中で大きな割合を占める場合があります。入力データに多数の同じ日付値がある場合は、この機能を使用すると実際に変換される日付の数が減少するため、パフォーマンスが向上します。ただし、日付キャッシュ機能によってパフォーマンスが向上するのは、多数の同じ日付が入力値として日付列にロードされる場合のみです（この章で使用する「日付」という用語は、日付およびタイムスタンプのすべてのデータ型を指します）。

日付キャッシュ・サイズを明示的に指定した場合、デフォルトでは日付キャッシュ機能は使用禁止になりません。この動作をオーバーライドするには、`OCI_ATTR_DIRPATH_DCACHE_DISABLE` を 1 に設定します。この設定以外の場合は、データ変換を回避するためにキャッシュが検索されます。ただし、キャッシュ・ミスが発生すると、変換のために時間がかかります。

`OCI_ATTR_DIRPATH_DCACHE_NUM` 属性、`OCI_ATTR_DIRPATH_DCACHE_MISSES` 属性および `OCI_ATTR_DIRPATH_DCACHE_HITS` 属性を問い合せて、ロードに必要なキャッシュ・サイズを調整します。

キャッシュ・ミスがなく、キャッシュ内の要素数がキャッシュ・サイズより小さい場合は、キャッシュ・サイズを小さくします。多数のキャッシュ・ミスが発生し、相対的にヒット数が少ない場合は、キャッシュ・サイズを大きくすることができます。キャッシュ・サイズを大きくしすぎると、ページングやメモリー消費量の増加など、別の問題が発生する可能性があります。キャッシュ・サイズを大きくしてもパフォーマンスが向上しない場合、この機能は使用しないでください。

日付キャッシュ機能を明示的かつ全体的に使用禁止にするには、日付キャッシュ・サイズを 0（ゼロ）に設定します。

この機能では、次の OCI ダイレクト・パス・コンテキスト属性を設定できます。

OCI_ATTR_DIRPATH_DCACHE_SIZE

この属性を 0（ゼロ）以外に設定する場合は、表に対する日付キャッシュ・サイズを要素数で設定します。たとえば、日付キャッシュ・サイズを 200 に設定すると、最大 200 個の一意の日付またはタイムスタンプ値をキャッシュに格納できます。OCI DirPathPrepare() をコールした後は、日付キャッシュ・サイズを変更できません。デフォルト値は 0（ゼロ）で、これは表に対する日付キャッシュが作成されないことを意味します。日付キャッシュは、データ型変換が必要な 1 つ以上の日付またはタイムスタンプ値がロードされ、この属性値が 0（ゼロ）以外の場合のみ、表に対して作成されます。

OCI_ATTR_DIRPATH_DCACHE_NUM

この属性を使用して、日付キャッシュ内の現在のエントリ数を問い合わせます。

OCI_ATTR_DIRPATH_DCACHE_MISSES

この属性を使用して、現在の日付キャッシュ・ミス数を問い合わせます。この数が多い場合は、日付キャッシュ・サイズを大きくするようにアプリケーションを調整することを検討してください。日付キャッシュ・サイズを大きくしてもこの数が大幅に減少しない場合は、日付キャッシュ機能を使用しないでください。日付キャッシュ・ミスが発生すると、ハッシングや参照のために時間を費やします。

OCI_ATTR_DIRPATH_DCACHE_HITS

この属性を使用して、日付キャッシュ・ヒット数を問い合わせます。日付キャッシュ機能の使用による効果があることを確認するには、この数が相対的に多くなる必要があります。

OCI_ATTR_DIRPATH_DCACHE_DISABLE

この属性を 1 に設定すると、サイズを超えた場合に日付キャッシュが使用禁止になります。OCI DirPathPrepare() をコールした後は、この属性を変更または設定できません。

デフォルト（= 0）では、オーバーフロー時にキャッシュが使用禁止になりません。キャッシュが使用禁止でない場合は、変換を回避するためにキャッシュが検索されますが、入力された日付値のエントリがオーバーフローすると日付キャッシュには追加されず、時間がかかる日付変換関数を使用して変換されます。多数の日付キャッシュ・ミスが発生すると、アプリケーションの処理速度は日付キャッシュを使用しない場合よりも遅くなる場合があります。

この属性を問い合わせると、オーバーフローが原因で日付キャッシュが使用禁止になっているかどうかを確認できます。

関連項目： A-58 ページ「ダイレクト・パス・コンテキスト・ハンドル (OCI DirPathCtx) の属性」

オブジェクト型のダイレクト・パス・ロード

この項では、ダイレクト・パス関数コンテキストによる多様な非スカラー型のロードについて説明します。

非スカラー型は、次のとおりです。

- NESTED TABLE
- オブジェクト表 (FINAL と NOT FINAL)
- 列オブジェクト (FINAL と NOT FINAL)
- REF 列 (FINAL と NOT FINAL)
- SQL 文字列の列

関連項目： Oracle のインストールで使用可能なダイレクト・パス・ロードのデモ・プログラムのリストは、[表 B-1 「OCI デモ・プログラム」](#) を参照してください。

NESTED TABLE のダイレクト・パス・ロード

NESTED TABLE は、個別の表に格納されます。ダイレクト・パス・ロード API を使用して、NESTED TABLE を SETID という外部キーで、その親の表から個別にロードし、2 つの表をリンクします。

注意：

- 現在、SETID はシステムで生成されないため、ユーザーによる指定が必要です。
 - 親と子の表を別々にロードするときに、子の表に行は挿入されているが、対応する親の行が親の表に挿入されていない場合、孤立した子が作成される可能性があります。また、親の表には親の行が挿入されているが、子の行が子の表に挿入されていない場合、子が欠落する可能性があります。
-
-

NESTED TABLE の列とその NESTED TABLE の記述

NESTED TABLE の列を持つ親の表のロードと、子の NESTED TABLE のロードとは、別のアクションです。

- NESTED TABLE の列を持つ親の表をロードする手順は、次のとおりです。

親の表とその列を通常どおり記述します。ただし、次の場合を除きます。

NESTED TABLE の列を記述する場合は、この列が、SETID が格納されている列です。データ・ファイルの SETID が文字の場合、その外部データ型は、SQLT_CHR です（バイナリの場合は、SQLT_BIN です）。

- NESTED TABLE（子）をロードする手順は、次のとおりです。

1. NESTED TABLE とその列を通常どおり記述します。
2. SETID 列は必須です。
 - a. ダミーの名前（setid など）を使用して、その OCI_ATTR_NAME を設定します。これは、API では、プログラマがそのシステム名を認識しているとはみなしていません。
 - b. 列が SETID 列であることを示すには、次のように、OCI_ATTR_DIRPATH_SID を使用して列属性を設定します。

```
ub1 flg = 1;
OCIAttrSet((dvoid *)colDesc, (ub4)OCI_DTYPE_PARAM, (dvoid *)&flg,
            (ub4)0, (ub4)OCI_ATTR_DIRPATH_SID, ctlp->errhp_ctl);
```

列オブジェクトのダイレクト・パス・ロード

列オブジェクトとは、オブジェクトとして定義される表の列のことです。現在は、すべての構成属性で構成されるデフォルト・コンストラクタのみがサポートされています。

列オブジェクトの記述

列オブジェクトとそのオブジェクト属性を記述するには、ダイレクト・パス関数コンテキストを使用します。列オブジェクトの記述には、そのオブジェクトのコンストラクタの設定が必要です。オブジェクト属性の記述は、スカラー列のリストの記述に類似しています。

列オブジェクトを記述する手順は、次のとおりです。

注意：

- ネストした列のオブジェクトがサポートされています。
 - 次に説明するステップは、表にロードするスカラー列のリストを記述するステップに類似しています。
-
-

1. パラメータ・ハンドルを `OCI_DTYPE_PARAM` で列オブジェクトに割り当てます。このパラメータ・ハンドルを使用して、列の外部属性を設定します。
2. 列名とその他の外部列の属性（最大データ・サイズ、精度、スケールなど）を設定します。
3. `OCI_ATTR_DATA_TYPE` で、外部型を `SQLT_NTY`（名前付き型）として設定します。
4. ダイレクト・パス関数コンテキスト・ハンドルを割り当てます。このコンテキストを使用して、列のオブジェクト型と属性を記述します。

```
OCIDirPathCtx *dpctx;          /* direct path context */
OCIDirPathFuncCtx *dpfnctx /* direct path function context */;
sword error;
error = OCIHandleAlloc((dvoid *)dpctx, (dvoid **)&dpfnctx,
                      OCI_HTYPE_DIRPATH_FN_CTX, (size_t)0, (dvoid **)0);
```

5. 関数コンテキストの `OCI_ATTR_NAME` で列のオブジェクト型の名前（Employee など）を設定します。

```
text *obj_type; /* column object's object type */
OCIAttrSet((dvoid *)dpfnctx, (ub4)OCI_HTYPE_DIRPATH_FN_CTX,
           (dvoid *)obj_type, (ub4)strlen((const char *)obj_type),
           (ub4)OCI_ATTR_NAME, ctlp->errhp_ctl);
```

6. `OCI_ATTR_DIRPATH_EXPR_TYPE` 式型を `OCI_DIRPATH_EXPR_OBJ_CONSTR` に設定します。これは、`OCI_ATTR_NAME` を持つ式セットが、デフォルトのオブジェクト・コンストラクタとして使用されることを意味します。

```
ub1 expr_type = OCI_DIRPATH_EXPR_OBJ_CONSTR;
OCIAttrSet((dvoid *)dpfnctx, (ub4)OCI_HTYPE_DIRPATH_FN_CTX,
           (dvoid *)&expr_type, (ub4)0, (ub4)OCI_ATTR_DIRPATH_EXPR_TYPE,
           ctlp->errhp_ctl);
```

7. `OCI_ATTR_NUM_COLS` を使用して、この列オブジェクトにロードする列またはオブジェクト属性の数を設定します。
8. 関数コンテキスト `OCIDirPathFuncCtx` の列/属性のパラメータ・リストを取得します。
9. オブジェクト属性ごとに、次の操作を行います。
 - `OCI_DTYPE_PARAM` で、オブジェクト属性の列記述子を取得します。
 - `OCI_ATTR_NAME` で属性の列名を設定します。
 - `OCI_ATTR_DATA_TYPE` で外部列の型（ダイレクト・パス API に渡されるデータの型）を設定します。
 - その他の外部列の属性（最大データ・サイズ、精度、スケールなど）を設定します。

- この属性の列が列オブジェクトの場合は、そのオブジェクト属性に対してステップ 3 ～ 10 を実行します。
- 列記述子へのハンドルを解放します。

10. ステップ 4 で作成された関数コンテキスト **OCIDirPathFuncCtx** を、親の列オブジェクトのパラメータ・ハンドルに **OCI_ATTR_DIRPATH_FN_CTX** を使用して設定します。

列配列の列オブジェクトへの割当て

列オブジェクトをロードすると、そのオブジェクトの属性データは、そのオブジェクト専用で作成された個別の列配列にロードされます。子の列配列は、ネストされているかどうかに関係なく、各列オブジェクトに割り当てられます。子の列配列にあるオブジェクト属性の各行は、その親列配列内の親の列オブジェクトにある対応する **NULL** 以外の行にマップされて戻されます。

列オブジェクトのダイレクト・パス関数コンテキスト・ハンドルと **OCI_HTYPE_DIRPATH_FN_COL_ARRAY** 列配列型を使用します。

列オブジェクトに子の列配列を割り当てるには、次のように行います。

```
OCIDirPathFuncCtx *dpfnctx;    /* direct path function context */
OCIDirPathColArray *dpfnca;    /* direct path function column array */

OCIHandleAlloc((dvoid *)dpfnctx,
               (dvoid **)&dpfnca, (ub4)OCI_HTYPE_DIRPATH_FN_COL_ARRAY,
               (size_t)0, (dvoid **)0);
```

列オブジェクト・データの列配列へのロード

列がスカラー列の場合は、その値のアドレスを **OCIDirPathColArrayEntrySet()** に渡して、その値を列配列に設定します。列がオブジェクトの場合は、かわりに、子の列配列ハンドルのアドレスを渡します。子の列配列にはオブジェクトの属性データが格納されます。

データを列オブジェクトにロードする手順は、次のとおりです。

(開始) 列オブジェクトごとに、次の操作を行います。

1. 列が NULL 以外の場合。
 - a. オブジェクト属性の列ごとに、次の操作を行います。

オブジェクト属性がネストされた列オブジェクトの場合は、(開始) に移動し、この全手順を繰り返し実行します。

`OCIDirPathColArrayEntrySet()` を使用して、子の列配列にデータを設定します。
 - b. 子の列配列ハンドルのアドレスを `OCIDirPathColArrayEntrySet()` に渡して、列オブジェクトのデータを列配列に設定します。
2. 列が NULL の場合。
 - データの NULL アドレス、0 (ゼロ) の長さおよび `OCI_DIRPATH_COL_NULL` フラグを `OCIDirPathColArrayEntrySet()` に渡して、列オブジェクトのデータを列配列に設定します。

SQL 文字列の列のダイレクト・パス・ロード

列の値は、SQL 文字列で計算できます。SQL 文字列は、スカラー列の型に使用できます。SQL 文字列は、オブジェクト型には使用できませんが、スカラー列の型のオブジェクト属性には使用できます。NESTED TABLE と LONG には使用できません。

SQL 式は、**OCIDirPathFuncCtx** を使用してダイレクト・パス API で表現します。式の `OCI_ATTR_NAME` の値が、その式の名前付きバインド変数のパラメータ・リストを含む SQL 文字列になります。

SQL 文字列の例を、次に示します。

```
substr(substr(:string, :offset, :length), :offset, :length)
```

この例では、次の点に注意してください。

- SQL 式はネストできます。
- バインド変数名は、その式内で繰り返し指定できます。

SQL 文字列の列の記述

1. パラメータ・ハンドルを `OCI_DTYPE_PARAM` で SQL 文字列の列に割り当てます。このパラメータ・ハンドルを使用して、列の外部属性を設定します。
2. 列名とその他の外部列の属性（最大データ・サイズ、精度、スケールなど）を設定します。
3. `OCI_ATTR_DATA_TYPE` を使用して、SQL 文字列の列の外部型を `SQLT_NTY` として設定します。
4. ダイレクト・パス関数コンテキスト・ハンドルを割り当てます。このコンテキストを使用して、SQL 文字列の引数を記述します。

```
OCIDirPathCtx *dpctx;          /* direct path context */
OCIDirPathFuncCtx *dpfnctx /* direct path function context */;
sword error;
error = OCIHandleAlloc((dvoid *)dpctx, (dvoid **)&dpfnctx,
                      OCI_HTYPE_DIRPATH_FN_CTX, (size_t)0, (dvoid **)0);
```

5. 列の SQL 文字列を関数コンテキストの `OCI_ATTR_NAME` に設定します。

```
text *sql_str; /* column's SQL string expression */
OCIAttrSet((dvoid *)dpfnctx, (ub4)OCI_HTYPE_DIRPATH_FN_CTX,
           (dvoid *)sql_str, (ub4)strlen((const char *)sql_str),
           (ub4)OCI_ATTR_NAME, ctlp->errhp_ctl);
```

6. `OCI_ATTR_DIRPATH_EXPR_TYPE` 式型を `OCI_DIRPATH_EXPR_SQL` として設定します。これは、`OCI_ATTR_NAME` を持つ式セットが、値を導出するための SQL 文字列として使用されることを意味します。

```
ub1 expr_type = OCI_DIRPATH_EXPR_SQL;
OCIAttrSet((dvoid *)dpfnctx, (ub4)OCI_HTYPE_DIRPATH_FN_CTX,
           (dvoid *)&expr_type, (ub4)0, (ub4)OCI_ATTR_DIRPATH_EXPR_TYPE,
           ctlp->errhp_ctl);
```

7. SQL 文字列に渡す引数の数を `OCI_ATTR_NUM_COLS` を使用して設定します。
8. 関数コンテキストの列 / 属性のパラメータ・リストを取得します。
9. SQL 文字列の引数ごとに、次の操作を行います。
 - `OCI_DTYPE_PARAM` で、オブジェクト属性の列記述子を取得します。
 - `OCI_ATTR_NAME` で属性の列名を設定します。

SQL 文字列の引数の定義順序は重要ではありません。SQL 文字列で使用されている順序と一致させる必要はありません。

SQL 文字列の引数には、次のネーミング規則が適用されます。

- 引数名は、SQL 文字列で使用されているバインド変数名と内容が一致している必要がありますが、大 / 小文字区別を一致させる必要はありません。たとえば、SQL 文字列が `substr(:INPUT_STRING,3,5)` の場合は、引数名を `input_string` にしてもかまいません。
 - 1 つの引数を SQL 文字列内で繰り返し使用する場合は、宣言した後、1 つの引数としてのみカウントしてください。
 - `OCI_ATTR_DATA_TYPE` で外部列の型（ダイレクト・パス API に渡されるデータの型）を設定します。
 - その他の外部列の属性（最大データ・サイズ、精度、スケールなど）を設定します。
 - 列記述子へのハンドルを解放します。
10. ステップ 4 で作成された関数コンテキスト `OCIDirPathFuncCtx` を、`OCI_ATTR_DIRPATH_FN_CTX` を使用して親の列オブジェクトのパラメータ・ハンドルに設定します。

列配列の SQL 文字列の列への割当て

SQL 文字列の列をロードすると、その引数のデータは、その SQL 文字列の列専用で作成された個別の列配列にロードされます。子の列配列は、SQL 文字列の列ごとに割り当てられます。子の列配列にある引数の各行は、その親列配列内の親の SQL 文字列の列にある対応する NULL 以外の行にマップされて戻されます。

SQL 文字列の列に子の列配列を割り当てるには、次のようにします。

```
OCIDirPathFuncCtx *dpfnctx; /* direct path function context */
OCIDirPathColArray *dpfnca; /* direct path function column array */

OCIHandleAlloc((dvoid *)dpfnctx,
               (dvoid **)&dpfnca, (ub4)OCI_HTYPE_DIRPATH_FN_COL_ARRAY,
               (size_t)0, (dvoid **)0);
```


SQL 文字列データの列配列へのロード

列がスカラー列の場合は、その値のアドレスを `OCIDirPathColArrayEntrySet()` に渡して、その値を列配列に設定します。列が SQL 文字列型の場合は、かわりに、子の列配列ハンドルのアドレスを渡します。子の列配列には、SQL 文字列の引数データが格納されます。

データを SQL 文字列の列にロードする手順は、次のとおりです。

SQL 文字列の列ごとに、次の操作を行います。

1. 列が NULL 以外の場合。
 - a. 関数の引数の列ごとに、次の操作を行います。

`OCIDirPathColArrayEntrySet()` を使用して、子の列配列にデータを設定します。
 - b. SQL 文字列の列のデータを列配列に設定するには、その子の列配列ハンドルのアドレスを `OCIDirPathColArrayEntrySet()` に渡します。
2. 列が NULL の場合。

データの NULL アドレス、0 (ゼロ) の長さおよび `OCI_DIRPATH_COL_NULL` フラグを `OCIDirPathColArrayEntrySet()` に渡して、SQL 文字列の列データを列配列に設定します。

この処理は、列オブジェクトの処理に類似しています。

REF 列のダイレクト・パス・ロード

REF 型とは、オブジェクト表の行オブジェクトへのポインタ、つまり参照です。

REF 列の記述

REF 列への引数の記述は、表にロードする列リストの記述に類似しています。

注意： REF 列は、表のトップレベル列に設定したり、オブジェクト属性として列オブジェクトにネストできます。

1. OCI_DTYPE_PARAM で、REF 列のパラメータ・ハンドルを取得します。このパラメータ・ハンドルを使用して、列の外部属性を設定します。
2. 列名とその他の外部列の属性（最大データ・サイズ、精度、スケールなど）を設定します。
3. OCI_ATTR_DATA_TYPE を使用して、REF 列の外部型を SQLT_REF として設定します。
4. ダイレクト・パス関数コンテキスト・ハンドルを割り当てます。このコンテキストを使用して、REF 列の引数を記述します。

```
OCIDirPathCtx *dpctx;          /* direct path context */
OCIDirPathFuncCtx *dpfnctx /* direct path function context */;
sword error;
error = OCIHandleAlloc((dvoid *)dpctx, (dvoid **)&dpfnctx,
                      OCI_HTYPE_DIRPATH_FN_CTX, (size_t)0, (dvoid **)0);
```

5. オプション: REF 列の表名を関数コンテキストの OCI_ATTR_NAME に設定します。詳細は、次のステップを参照してください。

```
text *ref_tbl;                /* column's reference table */
OCIAttrSet((dvoid *)dpfnctx, (ub4)OCI_HTYPE_DIRPATH_FN_CTX,
           (dvoid *)ref_tbl, (ub4)strlen((const char *)ref_tbl),
           (ub4)OCI_ATTR_NAME, ctlp->errhp_ctl);
```

6. オプション: OCI_ATTR_DIRPATH_EXPR_TYPE 式型を OCI_DIRPATH_EXPR_REF_TBLNAME に設定します。この設定は、ステップ 5 が完了している場合にのみ実行します。これは、OCI_ATTR_NAME を持つ式セットが、行オブジェクトを参照するためのオブジェクト表として使用されることを意味します。

このパラメータは、オプションです。このパラメータの動作は、REF の型によって異なります。

a. 有効範囲なしの REF 列（有効範囲なし、システム OID ベース）

このパラメータが設定されていない場合、この REF 列には、有効範囲なしの REF 列の定義により、参照表の名前が引数としてデータ行ごとに含まれている必要があります。

パラメータが設定されている場合、この REF 列がロードの継続時間中に参照できるのは、指定されたオブジェクト表の行オブジェクトのみです。この REF 列に、参照表の名前を引数として含めることはできません（ダイレクト・パス API は、このパラメータをショート・カットとしてユーザーに提供しています。ユーザーは、このパラメータをロードの継続時間中に同じ参照オブジェクト表を参照する有効範囲なしの REF 列にロードします）。

b. 有効範囲付 REF 列（有効範囲付、システム OID ベースおよび主キー・ベース）

このパラメータが設定されていない場合、ダイレクト・パス API は、スキーマで指定されている参照表を使用します。

このパラメータが設定されている場合、参照表の名前は、この有効範囲付 REF 列のスキーマに指定されているオブジェクト表と一致している必要があります。表名が一致していない場合は、エラーが発生します。

このパラメータが設定されているかどうかに関係なく、この参照表の名前がデータ行に存在しているかどうかは API にとって問題ではありません。名前がデータ行にある場合、その名前はスキーマに指定されている表名と一致している必要があります。名前がデータ行にない場合、API はスキーマに指定されている参照表を使用します。

7. 行オブジェクトの参照に使用する REF 引数の数を、OCI_ATTR_NUM_COLS で設定します。

必要な引数の数は、REF 列の型によって異なります。この数は、前述のステップ 6 で導出されます。

a. 有効範囲なしの REF 列（有効範囲なし、システム OID ベースの REF 列）

OCI_DIRPATH_EXPR_REF_TBLNAME が使用されている場合は、1 つの引数が導出されます。参照表の名前の引数はなく、OID 値の引数が 1 つです。

OCI_DIRPATH_EXPR_REF_TBLNAME が使用されていない場合は、2 つの引数が導出されます。参照表の名前の引数が 1 つと OID 値の引数が 1 つです。

b. 有効範囲付 REF 列（有効範囲付、システム OID ベースおよび主キー・ベース）

OCI_DIRPATH_EXPR_REF_TBLNAME を使用しているかどうかにかかわらず、オブジェクト ID を構成している列数が N の場合、受入れ可能な数は N または N+1 です。参照表の名前がデータ行にない場合、最小値は N です。参照表の名前がデータ行にある場合、その最小値は N+1 です。

注意： REF がシステム OID ベースの場合、N は 1 になります。REF が主キー・ベースの場合、N は主キーを構成するコンポーネント列の数になります。参照表の名前がデータ行にある場合は、N に 1 を加算します。

注意： エラー・メッセージを簡素化するために、N または N+1 以外の数の REF 引数を渡すと、期待値 N とは異なる数の引数が検出されたというエラー・メッセージが戻され、検出された引数の数が表示されます。メッセージでは N+1 について言及されていませんが、N+1 は受入れ可能です（参照表の名前が不要な場合でも同様です）。したがって、エラー・メッセージは表示されません。

8. 関数コンテキストの列 / 属性のパラメータ・リストを取得します。

9. REF の引数または属性ごとに、次の操作を行います。

- a. OCI_DTYPE_PARAM で、REF 引数の列記述子を取得します。
- b. OCI_ATTR_NAME で属性の列名を設定します。

REF 引数の順序は重要です。

参照表の名前が指定されている場合は、その名前を最初の REF 引数に指定します。

オブジェクト ID は、システム生成または主キー・ベースに関係なく、次の REF 引数に指定します。

REF 引数には次のネーミング規則が適用されます。

参照表の名前は表の列ではないため、列名に `ref-tbl` などのダミーの名前を使用できます。

システム生成の OID 列に対しては、列名に `sys-OID` などのダミーの名前を使用できます。

主キー・ベースのオブジェクト ID に対しては、ロード対象のすべての主キー列をリストします。OID に対してダミーの名前を作成する必要はありません。コンポーネント列名は、指定されている場合（次のショート・カットの注意を参照）、任意の順序で指定できます。

ショート・カットを使用する場合は、オブジェクト ID に属性列名を設定しないでください。

ショート・カット システム OID ベースの REF 列をロードする場合は、列名に名前を設定しないでください。名前は API によって判断されます。ただし、外部データ型などの他の列属性は、プログラマが設定する必要があります。

主キー REF 列をロードしており、その主キーが複数の列で構成されている場合、ショート・カットはそれぞれの列名に設定されません。ただし、外部データ型などの他の列属性は、プログラマが設定する必要があります。

注意： コンポーネント列名が NULL の場合は、API コードによって、主キーに定義された位置または順序で列名が決定されます。したがって、名前以外の列属性を設定する場合は、その属性がコンポーネントの列に適切な順序で設定されていることを確認してください。

- c. OCI_ATTR_DATA_TYPE で外部列の型（ダイレクト・パス API に渡されるデータの型）を設定します。
- d. その他の外部列の属性（最大データ・サイズ、精度、スケールなど）を設定します。

- e. 列記述子へのハンドルを解放します。
- f. ステップ 4 で作成された関数コンテキスト **OCIDirPathFuncCtx** を、**OCI_ATTR_DIRPATH_FN_CTX** を使用して親の列オブジェクトのパラメータ・ハンドルに設定します。

列配列の REF 列への割当て

REF 列に子の列配列を割り当てるには、次のようにします。

```
OCIDirPathFuncCtx *dpfnctx; /* direct path function context */
OCIDirPathColArray *dpfnca; /* direct path function column array */

OCIHandleAlloc((dvoid *)dpfnctx,
               (dvoid **)&dpfnca, (ub4)OCI_HTYPE_DIRPATH_FN_COL_ARRAY,
               (size_t)0, (dvoid **)0);
```

REF データの列配列へのロード

列がスカラー列の場合は、その値のアドレスを **OCIDirPathColArrayEntrySet()** に渡して、その値を列配列に設定します。列が REF の場合は、かわりに、子の列配列ハンドルのアドレスを渡します。子の列配列には、REF の引数のデータが格納されます。

データを REF 列にロードする手順は、次のとおりです。

REF 列ごとに、次の操作を行います。

1. 列が NULL 以外の場合。
 - a. REF 引数の列ごとに、次の操作を行います。

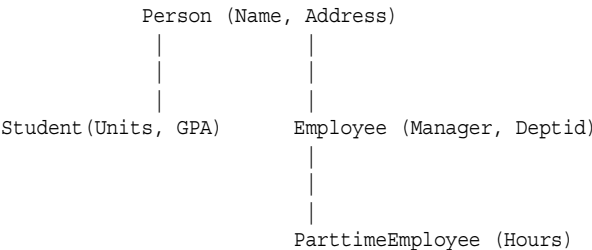
OCIDirPathColArrayEntrySet() を使用して、子の列配列にデータを設定します。
 - b. REF 列のデータを列配列に設定するには、子の列配列ハンドルのアドレスを **OCIDirPathColArrayEntrySet()** に渡します。
2. 列が NULL の場合。

データの NULL アドレス、0（ゼロ）の長さおよび **OCI_DIRPATH_COL_NULL** フラグを **OCIDirPathColArrayEntrySet()** に渡して、REF 列のデータを列配列に設定します。

NOT FINAL オブジェクトと REF 列

継承の階層の例を次に示します。この例の `Person` は、階層の最上位にあります。その下には、`Employee` と `Student` という 2 つのサブタイプがあります。`ParttimeEmployee` は `Employee` のサブタイプです。したがって、`Person` の列に格納できる型は、次の図のとおりです。

継承階層の図



`Person` 型の列を含む表をロードする場合、実際の型セットには、NOT FINAL 型 `Person` とその 3 つのサブタイプである `Student`、`Employee` および `ParttimeEmployee` を含めることができます。このロードの継続時間中、ダイレクト・パス API がサポートするのは、この NOT FINAL 列の固定的な派生型のロードのみです。したがって、API では、ロード対象となる型、その型の属性およびその型の作成に使用する関数などを認識する必要があります。

注意：

- ロードの継続時間中に、表の NOT FINAL 列に格納できるのは、1 つの固定的な派生型のみです。
 - 派生型を記述およびロードするときは、ロード対象となるその型の全属性を指定する必要があります。サブタイプは、この型とその親の全属性に固有なすべてのオブジェクト属性のフラット表現とみなしてください。したがって、ロード対象の属性列すべてを記述してカウントする必要があります。
 - たとえば、全列を `ParttimeEmployee` にロードする場合、ロード対象のオブジェクト属性は、`Name`、`Address`、`Manager`、`Deptid` および `Hours` の 5 つです。
-

ロード対象の固定的な派生型の記述

NOT FINAL または置換可能オブジェクトと固定的な派生型の REF 列を記述する手順は、次のとおりです。

注意： 固定的な派生型の NOT FINAL 列を記述するステップは、同じ型の FINAL 列を記述するステップと類似しています。

X 型 (X はオブジェクトまたは REF) の NOT FINAL 列を記述するには、この型の FINAL 列を記述している前述の各項を参照してください。派生型 (スーパータイプまたはサブタイプの場合があります) は、ロードの継続時間中固定されているため、NOT FINAL 列を記述するためのクライアント側インタフェースは、FINAL 列の場合と同じです。

サブタイプは、この型とその親の全属性に固有なすべてのオブジェクト属性のフラット表現とみなすことができます。したがって、ロード対象の属性列すべてを記述してカウントする必要があります。

列配列の割当て

同じ型の FINAL 列の場合と同じです。

データの列配列へのロード

同じ型の FINAL 列の場合と同じです。

オブジェクト表のダイレクト・パス・ロード

オブジェクト表とは、すべての行がオブジェクト (つまり、行オブジェクト) である表です。表の各列は、オブジェクト属性です。

オブジェクト表の記述

オブジェクト表の記述は、非オブジェクト表の記述と非常に類似しています。各オブジェクト属性が表の列です。唯一の相違点は、OID がシステム生成、ユーザー生成または主キー・ベースの場合に、その OID の記述が必要になることです。

オブジェクト表を記述する手順は、次のとおりです。

オブジェクト属性列ごとに、次の操作を行います。

それぞれの型 (NUMBER、REF など) に基づいて、各オブジェクト属性の列を必要に応じて記述します。

オブジェクト表が OID の場合は、次の操作を行います。

1. オブジェクト ID がシステム生成の場合。
何もする必要はありません。システムがすべての行オブジェクトの OID を生成します。
2. オブジェクト ID がユーザー生成の場合。
 - a. 仮の名前 (cust_oid など) を使用して OID の列名を表現します。
 - b. OCI_ATTR_DIRPATH_OID で OID の列属性を設定します。
3. オブジェクト ID が主キー・ベースの場合。
 - a. OID を構成しているすべての主キー列のロードが必要です。
 - b. OCI_ATTR_DIRPATH_OID を設定しないでください。ダミーの名前を持つ OID 列は作成されていません。

列配列のオブジェクト表への割当て

これは、非オブジェクト表に列配列を割り当てる方法と同じです。

```
OCIDirPathCtx *dpctx;      /* direct path context */
OCIDirPathColArray *dpca; /* direct path column array */
sword error;
error = OCIHandleAlloc((dvoid *)dpctx, (dvoid **)&dpca,
                      OCI_HTYPE_DIRPATH_COLUMN_ARRAY, 0, (dvoid **)0);
```

データの列配列へのロード

これは、非オブジェクト表にデータをロードする方法と同じです。

NOT FINAL オブジェクト表のダイレクト・パス・ロード

NOT FINAL オブジェクト表は継承をサポートしますが、FINAL オブジェクト表は継承をサポートできません。

NOT FINAL オブジェクト表の記述

固定的な派生型の NOT FINAL オブジェクト表の記述は、FINAL オブジェクト表の記述と非常に類似しています。

固定的な派生型の NOT FINAL オブジェクト表を記述する手順は、次のとおりです。

1. OCI_ATTR_DIRPATH_OBJ_CONSTR を使用して、ダイレクト・パス・コンテキストにオブジェクト表のオブジェクト型を設定します。これは、スーパータイプまたは派生型に関係なく、ロードの継続時間中、この表へのロードにはオブジェクト型がデフォルトのオブジェクト・コンストラクタとして使用されることを意味します。

```
text *obj_type;          /* the object type to load into this NOT FINAL */
                          /* object table */
OCIAttrSet((dvoid *)dpctx,
            (ub4)OCI_HTYPE_DIRPATH_CTX,
            (dvoid *) obj_type,
            (ub4)strlen((const char *) obj_type),
            (ub4)OCI_ATTR_DIRPATH_OBJ_CONSTR, errhp);
```

2. ロード対象のオブジェクト属性の列に、それぞれのデータ型に従って記述します。必要に応じて、オブジェクト ID を記述します。これは、FINAL オブジェクト表を記述する方法と同じです。

列配列の NOT FINAL オブジェクト表への割当て

これは、FINAL オブジェクト表を割り当てる方法と同じです。

ピース単位のダイレクト・パス・ロード

一度にメモリー内に格納できないデータのロードをサポートするために、ピース単位のロードを使用します。

ダイレクト・パス API では、すでに LONG と LOB の段階的なロードがサポートされています。これには、次の一連のステップを順に実行します。

1. 最初のピースを OCIDirPathColArrayEntrySet() を使用して列配列に設定し、OCI_DIRPATH_COL_PARTIAL フラグに渡して、この列の全データのロードが完了していないことを示します。
2. 列配列をストリームに変換します。
3. ストリームをロードします。
4. そのデータの次のピースを列配列に設定します。このピースが未完了の場合は、パーシャル・フラグを設定してステップ 2 に戻ります。完了している場合は、OCI_DIRPATH_COL_COMPLETE フラグを設定して次の列へ続行します。

このアプローチは、本質的には列オブジェクトの大規模な属性や SQL 文字列型の大規模な引数を扱う方法と同じです。

注意： コレクションは、このようにピース単位でロードしません。
NESTED TABLE は、トップレベル表と同じように、個別にロードします。
NESTED TABLE は段階的にロードでき、またピース単位でロードされた列を含めることができます。したがって、コレクションを格納している列には、OCI_DIRPATH_COL_PARTIAL フラグを設定しないでください。

オブジェクト型のピース単位のロード

オブジェクトは、そのオブジェクトが格納されている親の表から個別の列配列にロードします。したがって、オブジェクトをピース単位でロードする必要がある場合は、子の列配列に要素を設定して、ピース化された要素を含める必要があります。

通常の手順は、次のとおりです。

1. ピース化された要素に、OCI_DIRPATH_COL_PARTIAL フラグを設定します。
2. 子の列配列ハンドルを親の列配列に設定し、そのエントリも OCI_DIRPATH_COL_PARTIAL フラグでマークします。
3. この時点で、親の列配列をストリームに変換します。この結果、子の列配列もストリームに変換されます。
4. 次に、ストリームをロードします。
5. ステップ 1 に戻り、その要素に対する残りのデータのロードを、完了するまで続行します。

ピース単位のロードには、いくつかのルールがあります。

- パーシャル要素は、任意のレベルで一度に 1 つしか存在できません。1 つのパーシャル要素が完了とマークされた後は、そのレベルの別の要素をパーシャル要素に設定できません。
- 要素がパーシャルで、かつトップレベルでない場合は、制御階層上のその要素のすべての親にもパーシャルのマークを付ける必要があります。
- ネスト・レベルが複数の場合は、対象データをストリームに変換できるレベルまで移動する必要があります。これがトップレベル表になります。

ダイレクト・パス・コンテキスト・ハンドルとオブジェクト型の属性

次の説明では、付録 A にリストされているハンドルと属性の詳細を補足します。

ダイレクト・パス・コンテキストの属性

OCI_ATTR_DIRPATH_OBJ_CONSTR

オブジェクト型が NOT FINAL オブジェクト表にロードされることを示します。

```
text *obj_type;          /* the object type to load into this NOT FINAL */
                          /* object table */
OCIAttrSet((dvoid *)dpctx,
            (ub4)OCI_HTYPE_DIRPATH_CTX,
            (dvoid *) obj_type,
            (ub4)strlen((const char *) obj_type),
            (ub4)OCI_ATTR_DIRPATH_OBJ_CONSTR, errhp);
```

ダイレクト・パス関数コンテキストと属性

関数コンテキスト・ハンドルの属性の概要を、次に示します。

関連項目： A-58 ページ「[ダイレクト・パス・コンテキスト・ハンドル \(OCIDirPathCtx\) の属性](#)」

OCI_ATTR_DIRPATH_OBJ_CONSTR

オブジェクト型が置換可能なオブジェクト表にロードされることを示します。

```
text *obj_type; /* stores an object type name */
OCIAttrSet((dvoid *)dpctx,
            (ub4)OCI_HTYPE_DIRPATH_CTX,
            (dvoid *) obj_type,
            (ub4)strlen((const char *) obj_type),
            (ub4)OCI_ATTR_DIRPATH_OBJ_CONSTR, errhp);
```

OCI_ATTR_NAME

関数コンテキストの作成時に、非スカラー列を記述する式に相当する OCI_ATTR_NAME を設定します。次に、式の型を示す OCI 属性を設定します。次のように、様々な式の型があります。

1. 列オブジェクト

- a. 必要な式は、オブジェクト型名です。オブジェクト型はデフォルトのオブジェクト・コンストラクタとして使用されます。
- b. OCI_ATTR_DIRPATH_EXPR_TYPE 式型を OCI_DIRPATH_EXPR_OBJ_CONSTR に設定し、この式がオブジェクト型名であることを示します。

2. REF 列

- a. このオプションの式は、参照表名です。この表は、REF 列が行オブジェクトを参照する参照元のオブジェクト表です。
- b. OCI_ATTR_DIRPATH_EXPR_TYPE 式型を OCI_DIRPATH_EXPR_REF_TBLNAME に設定し、この式が参照オブジェクト表であることを示します。
- c. このパラメータの動作は、設定の有無に関係なく、REF の型によって異なります。
- 有効範囲なしの REF 列（有効範囲なし、システム OID ベース）

このパラメータが設定されていない場合、この REF 列には、有効範囲なしの REF 列の定義により、参照表の名前が引数としてデータ行ごとに含まれている必要があります。

パラメータが設定されている場合、この REF 列がロードの継続時間中に参照できるのは、指定されたオブジェクト表の行オブジェクトのみです。この REF 列に、参照表の名前を引数として含めることはできません（ダイレクト・パス API は、このパラメータをショート・カットとしてユーザーに提供しています。ユーザーは、このパラメータをロードの継続時間中に同じ参照オブジェクト表を参照する有効範囲なしの REF 列にロードします）。

- 有効範囲付 REF 列（有効範囲付、システム OID ベースおよび主キー・ベース）

このパラメータが設定されていない場合、ダイレクト・パス API は、スキーマで指定されている参照表を使用します。

このパラメータが設定されている場合、参照表の名前は、この有効範囲付 REF 列のスキーマに指定されているオブジェクト表と一致している必要があります。表名が一致していない場合は、エラーが発生します。

このパラメータが設定されているかどうかに関係なく、この参照表の名前がデータ行に存在しているかどうかは API にとって問題ではありません。名前がデータ行にある場合、その名前はスキーマに指定されている表名と一致している必要があります。名前がデータ行にない場合、API はスキーマに定義されている参照表を使用します。

3. SQL 文字列の列

この必須式には、列に格納される値を導出する SQL 文字列が含まれています。

OCI_ATTR_DIRPATH_EXPR_TYPE 式型を OCI_DIRPATH_EXPR_SQL に設定し、この式が SQL 文字列であることを示します。

OCI_ATTR_DIRPATH_EXPR_TYPE

この属性を使用して、非スカラー列の関数コンテキストに対して、OCI_ATTR_NAME で指定した式の型を示します。

OCI_ATTR_NAME が設定されている場合、OCI_ATTR_DIRPATH_EXPR_TYPE は必須です。

OCI_ATTR_DIRPATH_EXPR_TYPE に使用可能な値は、次のとおりです。

1. OCI_DIRPATH_EXPR_OBJ_CONSTR

- 式がオブジェクト型名であること、この式が列オブジェクトに対するデフォルトのオブジェクト・コンストラクタとして使用されることを示します。
- 列オブジェクトには必須です。

2. OCI_DIRPATH_EXPR_REF_TBLNAME

- 式が参照オブジェクト表の名前であることを示します。この表は、REF 列が行オブジェクトを参照する参照元のオブジェクト表です。
- REF 列の場合は、オプションです。

3. OCI_DIRPATH_EXPR_SQL

- 式が SQL 文字列であること、この SQL 文字列が列に格納されている値を導出するために実行されることを示します。
- SQL 文字列の列の場合は、必須です。

次の疑似コードは、前述のルールを例示したものです。

```
OCIDirPathFuncCtx *dpfnctx; /* function context for this non-scalar column */

ub1 exprtype; /* expression type */

if (column type is an object) then exprtype = OCI_DIRPATH_EXPR_OBJ_CONSTR;

if (column_type is a REF && function context name exists) then exprtype =
OCI_DIRPATH_EXPR_REF_TBLNAME;

if (column_type is a SQL string) then exprtype = OCI_DIRPATH_EXPR_SQL;
```

```
OCIAttrSet((dvoid *) (dpfnctx), (ub4)OCI_HTYPE_DIRPATH_FN_CTX, (dvoid *)
&exprtype, (ub4) 0, (ub4)OCI_ATTR_DIRPATH_EXPR_TYPE, ct1p->errhp, ct1);
```

OCI_ATTR_NUM_COLS

この属性は、非スカラー列用にロードまたは処理する属性や引数の数を記述します。このパラメータは、列リストが取り出される前に設定する必要があります。

1. 列オブジェクト

この列オブジェクト用にロードするオブジェクト属性列の数。

2. SQL 文字列の列

- a. SQL 文字列に渡す引数の数。
- b. 引数が関数内で繰り返して使用される場合も、1 つとしてカウントしてください。

3. REF 列

- a. REF 列が指し示す必要のある行オブジェクトを識別する REF 引数の数。
- b. 必要な引数の数は、REF 列の型によって異なります。
- 有効範囲なしの REF 列（有効範囲なし、システム OID ベースの REF 列）

OCI_DIRPATH_EXPR_REF_TBLNAME を使用している場合。参照表の名前ではなく、OID 値が 1 つです（OID 値のみがデータ行にある）。

OCI_DIRPATH_EXPR_REF_TBLNAME を使用していない場合。参照表の名前が 1 つ、OID 値が 1 つです（参照表名と OID 値の両方がデータ行にある）。

- 有効範囲付 REF 列（有効範囲付、システム OID ベースおよび主キー・ベース）

OCI_DIRPATH_EXPR_REF_TBLNAME を使用しているかどうかにかかわらず、オブジェクト ID を構成している列数が N の場合、受入れ可能な数は、N または N+1 です。参照表の名前がデータ行にない場合、最小値は N です。参照表の名前がデータ行にある場合は、N+1 を使用します。

REF がシステム OID ベースの場合、N は 1 になります。REF が主キー・ベースの場合、N は主キーを構成するコンポーネント列の数になります。参照表の名前がデータ行にある場合は、N に 1 を加算します。

注意： エラー・メッセージを簡素化するために、N または N+1 以外の数の REF 引数を渡すと、期待値 N とは異なる数の引数が検出されたというエラー・メッセージが戻され、検出された引数の数が表示されます。メッセージでは N+1 について言及されていませんが、N+1 は受入れ可能です（参照表の名前が不要な場合でも同様です）。したがって、エラー・メッセージは表示されません。

OCI_ATTR_NUM_ROWS

OCI_HTYPE_DIRPATH_FN_CTX（関数コンテキスト）に対して使用される場合、この属性は、取出し専用のため、ユーザーによる設定はできません。この属性は、OCIAttrGet() にのみ使用でき、OCIAttrSet() には使用できません。OCIAttrGet() を使用してコールすると、それまでにロードされた行数が戻されます。

ただし、属性 OCI_ATTR_NUM_ROWS は、OCI_HTYPE_DIRPATH_CTX（表レベルのコンテキスト）に対して使用する場合は設定可能で、ユーザーによる取出しが可能です。

OCIAttrSet() を OCI_ATTR_NUM_ROWS と OCI_HTYPE_DIRPATH_CTX を使用してコールすると、表レベルの列配列に割り当てる行数が設定されます。設定されない場合、ダイレクト・パス API コードは、レコード最大サイズと転送バッファのサイズに基づいて適切な数を導出します。割り当てた行数を確認するには、表レベルの列配列については、OCI_ATTR_NUM_ROWS を OCI_HTYPE_DIRPATH_COLUMN_ARRAY で使用し、関数列配列については、OCI_HTYPE_DIRPATH_FN_COL_ARRAY を使用して OCIAttrGet() をコールします。

OCIAttrGet() を OCI_ATTR_NUM_ROWS と OCI_HTYPE_DIRPATH_CTX でコールすると、それまでにロードされた行数が戻されます。

この属性は、ユーザーが関数コンテキストに設定することはできません。プログラマが OCIAttrSet() で OCI_ATTR_NUM_ROWS をコールして関数列配列に必要な行数を指定することは許可されません。これは、コールすると、すべての関数列配列が表レベルの列配列と同じ行数を所有することになるためです。この属性を設定できるのは、表レベルのコンテキストのみで、関数コンテキストには設定できません。

ダイレクト・パス列パラメータの属性

オブジェクト、SQL 文字列または REF 列を記述すると、その列属性の 1 つが関数コンテキストになります。

列がオブジェクトの場合、その関数コンテキストには、そのオブジェクト型とオブジェクト属性が記述されます。SQL 文字列の場合は、コール対象の式が記述されます。REF の場合は、その参照表の名前と行オブジェクトの識別子が記述されます。

関数コンテキストを列属性として設定すると、OCIAttrSet() では OCI_ATTR_DIRPATH_FN_CTX が使用されます。

```
OCIParam *colDesc;           /* column parameter descriptor */
OCIDirPathFuncCtx *dpfnctx;  /* direct path function context */
OCIError *errhp;             /* error handle */

OCIAttrSet((dvoid *)colDesc, (ub4)OCI_DTYPE_PARAM,
           (dvoid *) (dpfnctx), (ub4)0,
           (ub4)OCI_ATTR_DIRPATH_FN_CTX, errhp);
```

列パラメータ・コンテキスト・ハンドルの属性について次に説明します。

関連項目： A-68 ページ「[ダイレクト・パス列パラメータ属性](#)」

OCI_ATTR_NAME

次に、NESTED TABLE、オブジェクト表、SQL 文字列の列および REF 列をロードするときのネーミング規則について説明します。

通常、オブジェクト表のシステム生成オブジェクト ID (OID) 列や NESTED TABLE の SETID (SID) 列など、プログラマにとって不明なシステム名を持つシステム列にデータをロードする場合、あるいは列がデータベース表の列を持たない引数 (SQL 文字列や REF 引数など) である場合は、ダミーの名前を使用します。

列はデータベース表の列であっても、ダミーの名前を使用した場合は、その列がデータベースで識別されない名前の場合でも関数が識別できるように、列属性を設定する必要があります。

ネーミング規則は、次のとおりです。

1. 子の NESTED TABLE の SETID (SID) 列

SETID 列は必須です。ダミーの名前を使用して OCI_ATTR_NAME を設定します。これは、API では、ユーザーがシステム名を認識していないとみなしているためです。次に、これが SID 列であることを示すために、OCI_ATTR_DIRPATH_SID を使用して列属性を設定します。

2. オブジェクト表のオブジェクト ID (OID) 列

次の場合、オブジェクト ID は必須です。

a. オブジェクト ID がシステム生成の場合。

列名にダミーの名前 (たとえば、cust_oid) を使用します。

OCI_ATTR_DIRPATH_OID で列属性を設定します。その結果、ダミーの名前を持つ列が複数ある場合にも、システム生成の OID を表す列を識別できます。

b. オブジェクト ID が主キー・ベースの場合。

列名にはダミーの名前を使用できません。したがって、OCI_ATTR_DIRPATH_OID を使用してその列属性を設定する必要はありません。

3. SQL 文字列の引数

OCI_ATTR_NAME で属性の列名を設定します。

SQL 文字列の引数の順序は、重要ではありません。SQL 文字列で使用されている順序と一致させる必要はありません。

SQL 文字列の引数には、次のネーミング規則が適用されます。

- a. 引数名は、SQL 文字列で使用されているバインド変数名と内容が一致している必要がありますが、大 / 小文字区別を一致させる必要はありません。たとえば、SQL 文字列が `substr(:INPUT_STRING, 3, 5)` の場合は、引数名を `input_string` にできます。
- b. 1 つの引数が SQL 文字列で繰り返し使用される場合、それを一度宣言した後は、1 つの引数としてカウントできます。

4. REF 引数

- a. OCI_ATTR_NAME で属性の列名を設定します。

REF 引数の順序は重要です。

- 参照表の名前が指定されている場合は、その名前を最初の REF 引数に指定します。
 - オブジェクト ID は、システム生成または主キー・ベースに関係なく、次の REF 引数に指定します。
- b. REF 引数には次のネーミング規則が適用されます。

- 参照表名の引数の場合は、その列名に `ref-tbl` などの仮の名前を使用します。
- システム生成 OID の引数に対しては、列名に `sys-OID` などの仮の名前を使用します。

注意：この列は、引数として使用され、ロード対象の列としては使用されないため、OCI_ATTR_DIRPATH_OID で設定しないでください。

- 主キー・ベースのオブジェクト ID に対しては、ロード対象のすべての主キー列をリストします。OID に対してダミーの名前を作成する必要はありません。コンポーネント列名は、指定されている場合（次のショート・カットのステップを参照）、任意の順序で指定できます。
- c. ショート・カットを使用する場合は、オブジェクト ID に属性列名を設定しないでください。
- **ショート・カット** システム OID ベースの REF 列をロードする場合は、列名に名前を設定しないでください。名前は API によって判断されます。ただし、外部データ型など他の列属性は、プログラマが設定する必要があります。
 - 主キー REF 列をロードしており、その主キーが複数の列で構成されている場合、ショート・カットはそれぞれの列名に設定されません。ただし、外部データ型など他の列属性は、ユーザーが設定する必要があります。

注意： コンポーネント列名が NULL の場合は、API コードによって、主キーに定義された位置または順序で列名が決定されます。したがって、名前以外の列属性を設定する場合は、その属性がコンポーネントの列に適切な順序で設定されていることを確認してください。

OCI_ATTR_DIRPATH_SID

列が NESTED TABLE の SETID 列であることを示します。NESTED TABLE にロードする場合は必須です。

```
ub1 flg = 1;
OCIAttrSet((dvoid *)colDesc, (ub4)OCI_DTYPE_PARAM, (dvoid *)&flg, (ub4)0,
            (ub4)OCI_ATTR_DIRPATH_SID, ctlp->errhp_ctl);
```

OCI_ATTR_DIRPATH_OID

列が NESTED TABLE のオブジェクト ID 列であることを示します。

```
ub1 flg = 1;
OCIAttrSet((dvoid *)colDesc, (ub4)OCI_DTYPE_PARAM, (dvoid *)&flg, (ub4)0,
            (ub4)OCI_ATTR_DIRPATH_OID, ctlp->errhp_ctl);
```

非スカラー列のダイレクト・パス関数列配列ハンドル

関連項目： A-66 ページ「[ダイレクト・パスおよびダイレクト・パス関数列配列ハンドル \(OCIDirPathColArray\) 属性](#)」

列がオブジェクト、SQL 文字列または REF の場合は、OCI_HTYPE_DIRPATH_FN_COL_ARRAY ハンドル型を使用します。構造体 **OCIDirPathColArray** は、スカラー列と非スカラー列両方の場合で同じです。

関数コンテキストに子の列配列を割り当てるには、次のようにします。

```
OCIDirPathFuncCtx *dpfnctx; /* direct path function context */
OCIDirPathColArray *dpfnca; /* direct path function column array */

OCIHandleAlloc((dvoid *)dpfnctx,
                (dvoid **)&dpfnca,
                (ub4)OCI_HTYPE_DIRPATH_FN_COL_ARRAY,
                (size_t)0, (dvoid **)0);
```

OCI_ATTR_NUM_ROWS 属性

OCI_HTYPE_DIRPATH_FN_COL_ARRAY（関数列配列）に対して使用される場合、この属性は、取出し専用のため、ユーザーによる設定はできません。関数 `OCIAttrGet()` でコールすると、関数列配列に割り当てられた行数が戻されます。

オブジェクトのキャッシュおよびナビゲーション

この章では、オブジェクトを Oracle データベース・サーバーで操作するための OCI の機能を紹介します。また、OCI のオブジェクト・ナビゲーション関数コールについても説明します。この章は、次の項目で構成されています。

- [オブジェクト・キャッシュおよびメモリー管理](#)
- [オブジェクト・ナビゲーション](#)
- [OCI ナビゲーション関数](#)
- [型の変更とオブジェクト・キャッシュ](#)

オブジェクト・キャッシュおよびメモリ管理

オブジェクト・キャッシュは、オブジェクトの参照とメモリ管理をサポートするクライアント側のメモリ・バッファです。これは OCI アプリケーションでフェッチしたオブジェクト・インスタンスを格納し、追跡するためのものです。オブジェクト・キャッシュには、メモリ管理が用意されています。

アプリケーションで SQL の SELECT 文または OCI の PIN オペレーションによってオブジェクトをフェッチすると、オブジェクトのコピーがオブジェクト・キャッシュに格納されます。SELECT 文で直接フェッチされたオブジェクトは、値によってフェッチされる、確保できない参照不可能なオブジェクトです。確保できるのは参照可能オブジェクトのみです。

オブジェクトを確保するとき、適切なバージョンのオブジェクトがすでにキャッシュ内に存在していれば、サーバーからオブジェクトをフェッチする必要はありません。

OCI を使用して REF を参照解除し、オブジェクトを取り出す各クライアント・プログラムでは、オブジェクト・キャッシュが使用されます。クライアント側のオブジェクト・キャッシュは、オブジェクト・モードで初期化されたすべての OCI 環境ハンドルで割り当てられます。1 つのプロセスの複数のスレッドで、同一の OCI 環境ハンドルを共有することによって同一のクライアント側キャッシュを共有できます。

オブジェクト・キャッシュ内には接続ごとに、各参照可能オブジェクトのコピーがそれぞれ 1 つあります。1 つの REF を複数回参照解除するか、等価な複数の REF を参照解除した場合は、オブジェクトの同一コピーが戻されます。

キャッシュ内にあるオブジェクトのコピーを変更した場合、変更内容をその他のプロセスに見えるようにするには、変更内容をサーバーにフラッシュする必要があります。必要のなくなったオブジェクトは、確保解除つまり解放できます。そしてキャッシュからスワップ・アウトでき、それによって、オブジェクトが使用していたメモリ領域を解放できます。

キャッシュにロードされたデータベース・オブジェクトは、C 言語構造体に透過的にマップされます。オブジェクト・キャッシュでは、キャッシュ内のすべてのオブジェクト・コピーと、それぞれに対応するデータベース内のオブジェクトとの間の関連付けが、メンテナンスされます。トランザクションをコミットすると、キャッシュ内のオブジェクト・コピーに対して行った変更がデータベースに自動的に伝播されます。

オブジェクト・キャッシュではオブジェクト・コピーの内容は管理されず、オブジェクト・コピーが自動的にリフレッシュされることもありません。アプリケーションでは、オブジェクト・コピーの内容が正しく、一貫性があることを確認する必要があります。たとえば、アプリケーションでオブジェクト・コピーに挿入、更新または削除のマークを付けた後、トランザクションが異常終了したとします。この場合、オブジェクト・キャッシュではオブジェクト・コピーのマークが解除されるのみであり、コピーの削除や無効化は行われません。アプリケーションで *recent* または *latest* を確保するか、オブジェクトのコピーを次のトランザクションでリフレッシュします。*any* を確保すると、以前異常終了したトランザクションから、変更をコミットしていない同じオブジェクト・コピーを取得する場合があります。

関連項目： 確保オプションの詳細は、13-7 ページの「[オブジェクト・コピーの確保](#)」を参照してください。

注意： オブジェクト・キャッシュは、接続によって論理的にパーティション化されます。つまり、サービス・コンテキストです。これにより、すべてのオブジェクト型（TDO）と表定義をフェッチするために、それらを使用する各サービス・コンテキストに OCI コードが書き込まれます。プログラムは、1 つのサービス・コンテキストについてすべての TDO と表をフェッチし、それらを他のサービス・コンテキストにも使用するもので、キャッシュが少ないためにオブジェクトがエージ・アウトすることがないかぎり、要求に応じて動作します。ただし、TDO と表が一度キャッシュからエージ・アウトすると、予期しない動作が起こる可能性があります。この場合、内部エラーが発生します。

オブジェクト・キャッシュは、*mode* が OCI_OBJECT の OCIEnvCreate() によって OCI 環境が初期化されると作成されます。

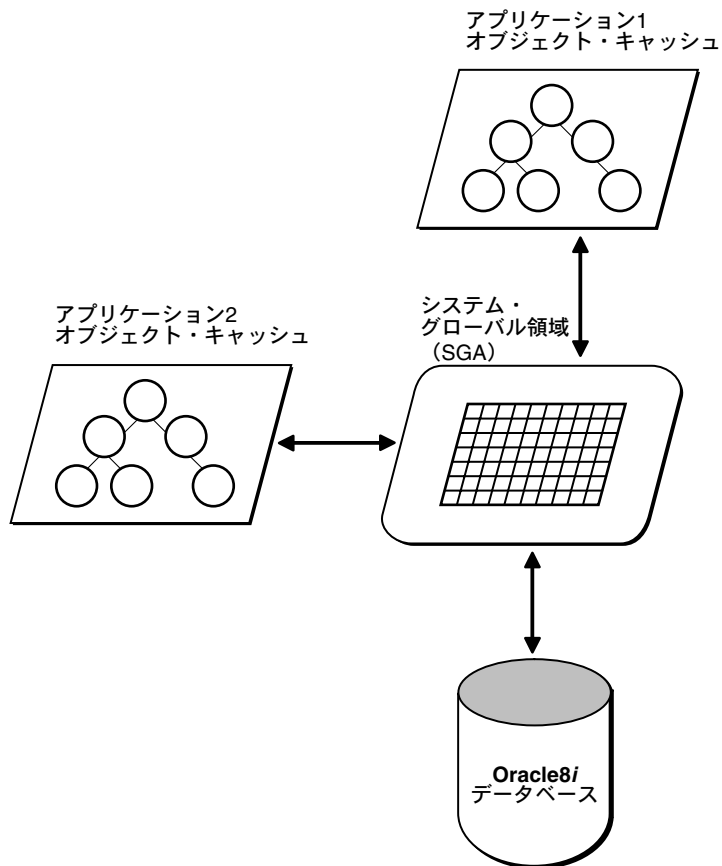
オブジェクト・キャッシュでは、REF をオブジェクトにマッピングするための高速参照表がメンテナンスされています。アプリケーションが REF を参照解除するときに、対応するオブジェクトがオブジェクト・キャッシュにキャッシュされていない場合、オブジェクト・キャッシュは、サーバーに対して、オブジェクトをデータベースからフェッチしてオブジェクト・キャッシュにロードする要求を自動的に送信します。

同じ REF の後続の参照解除は、ローカル・キャッシュへのアクセスとなり、ネットワーク・ラウンドトリップが発生しないため、高速になります。アプリケーションによるキャッシュ内オブジェクトへのアクセスを、オブジェクト・キャッシュに通知するために、アプリケーションは、オブジェクトを確保します。アプリケーションによるオブジェクトの処理が終了すると、オブジェクトの確保解除が行われます。オブジェクト・キャッシュでは、キャッシュ内の各オブジェクトの確保カウントがメンテナンスされています。このカウントは確保コール時に増分され、確保解除コール時に減分されます。確保カウントが 0（ゼロ）の場合は、そのオブジェクトがアプリケーションでは不要になったことを意味します。

オブジェクト・キャッシュでは、最低使用頻度（LRU）アルゴリズムを使用して、キャッシュのサイズが管理されます。LRU アルゴリズムは、キャッシュが最大サイズに達すると、候補オブジェクトを解放します。候補オブジェクトとは、確保カウントが 0（ゼロ）のオブジェクトのことです。

同じサーバーに対して稼働中の各アプリケーション・プロセスには、[図 13-1「オブジェクト・キャッシュ」](#)で示すような専用のオブジェクト・キャッシュがあります。

図 13-1 オブジェクト・キャッシュ



オブジェクト・キャッシュは、メモリーに現在あるオブジェクトの追跡、そのオブジェクトへの参照のメンテナンス、自動オブジェクト・スワッピングの管理、オブジェクトのメタ属性の追跡を行います。

キャッシュの一貫性

オブジェクト・キャッシュ内のオブジェクト・コピーと、データベース内で対応するオブジェクトとの間における値の一貫性は、自動的にメンテナンスされません。つまり、アプリケーションでオブジェクト・コピーを変更すると、その変更内容はデータベース内の対応するオブジェクトに自動的に適用されず、その逆も同様です。オブジェクト・キャッシュでは、変更されたオブジェクト・コピーをデータベースにフラッシュする操作や、古いオブジェクト・コピーをデータベースの最新値にリフレッシュする操作が可能であり、プログラムである程度の一貫性をメンテナンスできます。

注意： Oracle は、サーバーのバッファ・キャッシュまたはデータベースとの自動キャッシュ一貫性をサポートしていません。自動キャッシュ一貫性とは、対応するオブジェクトがサーバーのバッファ・キャッシュで変更されると、ローカル・オブジェクト・コピーがオブジェクト・キャッシュによってリフレッシュされるメカニズムを指します。このメカニズムは、オブジェクト・キャッシュが、サーバー内の対応するオブジェクトに直接アクセスする前に、ローカル・オブジェクト・キャッシュに行われた変更をバッファ・キャッシュにフラッシュしたとき実行されます。直接的なアクセスには、SQL、トリガーまたはストアード・プロシージャを使用した、サーバーのオブジェクトの読み込みや変更が含まれます。

オブジェクト・キャッシュ・パラメータ

オブジェクト・キャッシュには、環境ハンドルの属性である、2つの重要なパラメータが関連付けられています。

- `OCI_ATTR_CACHE_MAX_SIZE` キャッシュの最大サイズ
- `OCI_ATTR_CACHE_OPT_SIZE` キャッシュの最適サイズ

これらのパラメータは、キャッシュのメモリー使用量レベルを示し、削除してもよいオブジェクトをいつ自動的にキャッシュから削除してメモリーを解放するかを判断するのに役立ちます。

現在キャッシュにあるオブジェクトのメモリー使用量が最大キャッシュ・サイズに達するかそれを超えると、キャッシュでは、確保カウントが0（ゼロ）の、マークされていないオブジェクトの自動的な解放（またはエージ・アウト）が始まります。キャッシュのメモリー使用量が最適サイズに達するか、解放対象オブジェクトがなくなるまで、オブジェクトの解放が続きます。キャッシュが指定した最大キャッシュ・サイズを超えることがあります。

`OCI_ATTR_CACHE_MAX_SIZE` は、`OCI_ATTR_CACHE_OPT_SIZE` のパーセントで指定します。オブジェクト・キャッシュの最大サイズ（バイト単位）は、次のように `OCI_ATTR_CACHE_OPT_SIZE` を `OCI_ATTR_CACHE_MAX_SIZE` のパーセントで増分して計算します。

```
maximum_cache_size = optimal_size + optimal_size * max_size_percentage / 100
```

または

```
maximum_cache_size = OCI_ATTR_CACHE_OPT_SIZE + OCI_ATTR_CACHE_OPT_SIZE *  
OCI_ATTR_CACHE_MAX_SIZE / 100
```

OCI_ATTR_CACHE_MAX_SIZE の値は、OCI_ATTR_CACHE_OPT_SIZE の 110% に設定します。OCI_ATTR_CACHE_OPT_SIZE のデフォルト値は 8MB です。

環境ハンドルのキャッシュ・サイズ属性の設定には OCIAttrSet () コールを、取出しには OCIAttrGet () 関数を使用します。

関連項目： 詳細は、A-3 ページの「[環境ハンドル属性](#)」を参照してください。

オブジェクト・キャッシュ操作

この項では、オブジェクト・キャッシュで使用されるオブジェクト・コピーを操作する重要な関数について説明します。

関連項目： OCI のナビゲーション関数およびキャッシュまたはオブジェクトの管理関数は、13-20 ページの「[OCI ナビゲーション関数](#)」にすべてリストされています。

確保と確保解除

オブジェクト・コピーの確保によって、アプリケーションは、そのコピーへの REF を参照解除して、キャッシュ内のオブジェクト・コピーにアクセスできるようになります。

オブジェクトを確保解除すると、オブジェクトが現在使用中でないことがキャッシュに対して示されます。不要になったオブジェクトは確保解除してキャッシュからの暗黙的な解放の対象にし、メモリを解放してください。

解放

オブジェクト・コピーを解放すると、オブジェクト・コピーはキャッシュから削除され、使用されていたメモリが解放されます。

マークとマーク解除

オブジェクトをマークすることで、キャッシュ内のオブジェクト・コピーが更新されており、オブジェクト・コピーをフラッシュするときにサーバー内の対応するオブジェクトを更新する必要があることが、キャッシュに通知されます。

オブジェクトをマーク解除すると、オブジェクトが更新されたという指示が削除されます。

フラッシュ

オブジェクトをフラッシュすると、キャッシュ内のマークされたオブジェクト・コピーに対して行ったローカルな変更が、サーバー内の対応するオブジェクトに書き込まれます。それによって、オブジェクト・キャッシュ内のコピーのマークが解除されます。

リフレッシュ

キャッシュ内のオブジェクト・コピーをリフレッシュすると、そのコピーは、サーバー内の対応するオブジェクトの最新値に置き換わります。

注意： トップレベルのオブジェクト・メモリーへのポインタは、リフレッシュ後も有効です。2 次レベル・メモリーへのポインタ（文字列テキスト・ポインタ、コレクションなど）は、リフレッシュ後に無効になる場合があります。

オブジェクト・コピーのロードと削除

この項では、確保、確保解除および解放の各関数を説明します。

オブジェクト・コピーの確保

オブジェクト・キャッシュ内の REF をアプリケーションで参照解除する必要がある場合は、`OCIObjectPin()` をコールします。このコールによって REF が参照解除され、キャッシュ内のオブジェクト・コピーが確保されます。オブジェクト・コピーは、確保されているかぎり、アプリケーションからアクセス可能であることが保証されます。`OCIObjectPin()` のバリエーションの 1 つに、`OCIObjectArrayPin()` があります。これは REF の配列を取得して REF を参照解除し、オブジェクト・コピーを確保します。`OCIObjectPin()` と `OCIObjectArrayPin()` の両方で、PIN オプションである *any*、*recent* または *latest* を受け付けます。PIN オプションのデータ型は **OCIPinOpt** です。

- *any* (`OCI_PIN_ANY`) オプションが指定される場合で、キャッシュ内にオブジェクト・コピーがすでにある場合は、それをオブジェクト・キャッシュでただちに戻します。キャッシュ内にオブジェクト・コピーがない場合、オブジェクト・キャッシュでは最新のオブジェクト・コピーをデータベースからロードして、そのオブジェクト・コピーが戻されます。*any* オプションは、製品、販売員、ベンダー、販売領域、部品、事務所などの、読取り専用、情報、事実またはメタのオブジェクトに適しています。これらのオブジェクトは、通常は頻繁に変更されることはなく、変更されてもその変更内容はアプリケーションに影響しません。
- *latest* (`OCI_PIN_LATEST`) オプションが指定されている場合、オブジェクト・キャッシュでは、最新のオブジェクト・コピーがデータベースからキャッシュにロードされます。そのオブジェクト・コピーがキャッシュ内でロックされていないかぎり、そのオブジェクト・コピーは戻されます。この場合、マークされたオブジェクト・コピーがただちに戻されます。オブジェクトがすでにキャッシュ内にあり、ロックされていない場合は、最新のオブジェクト・コピーがロードされ、既存のコピーが上書きされます。*latest*

オブションは、発注情報、バグ、明細項目、銀行口座、株価などの操作オブジェクトに適しています。これらのオブジェクトは通常頻繁に変更され、プログラムではこれらのオブジェクトにできるだけ最新の状態でアクセスが試行されます。

- *recent* (OCI_PIN_RECENT) オブションが指定される場合、2つの可能性が考えられます。
 - 同じトランザクションにおいて、*latest* または *recent* オブションを使用して、オブジェクト・コピーが以前に確保されている場合は、*recent* オブションは *any* オブションと同等になります。
 - 前述の条件が当てはまらない場合は、*recent* オブションは *latest* オブションと同等になります。

プログラムでオブジェクトが確保される場合、プログラムでは *session* または *transaction* という2つの可能な値の内の1つが確保継続時間に指定されます。継続時間のデータ型は **OCIDuration** です。

- 確保継続時間が *session* (OCI_DURATION_SESSION) の場合、セッションを終了するまで（つまり接続の終了まで）、またはプログラムで (OCIObjectUnpin()) をコールして明示的に確保解除されるまで、オブジェクト・コピーは確保されたままとなります。
- 確保継続時間が *transaction* (OCI_DURATION_TRANS) の場合、トランザクションの終了まで、または明示的に確保解除されるまで、オブジェクト・コピーは確保されたままとなります。

オブジェクト・コピーをデータベースからキャッシュにロードするとき、実際にはキャッシュで次の命令が実行されます。

```
SELECT VALUE(t) FROM t WHERE REF(t) = :r
```

このコード例の *t* はオブジェクトが格納されているオブジェクト表であり、*r* は REF です。フェッチされた値はキャッシュ内のオブジェクト・コピーの値になります。

オブジェクト・キャッシュでは、個々の **SELECT** 文が効率的に実行され、コミット読みトランザクションで各オブジェクト・コピーがキャッシュにロードされます。オブジェクト・コピー相互の読み一貫性は保証されません。

直列可能トランザクションでは、*recent* または *latest* で確保されたオブジェクト・コピーは相互の読み一貫性があります。これは、これらのオブジェクト・コピーをロードする **SELECT** 文は、同じデータベース・スナップショットに基づいて実行されるためです。

オブジェクト・キャッシュのモデルは、Oracle トランザクション・モデルに対して独立しています。オブジェクト・キャッシュの動作は、トランザクション・モデルによって変わることはありません。ただし、同じプログラムを異なる複数のトランザクション・モデル（コミット読みとシリアル化可能など）のもとで実行した場合、オブジェクト・キャッシュを介してサーバーから取り出したオブジェクトは、異なることがあります。

オブジェクト・コピーの確保解除

プログラムで使用しないオブジェクト・コピーは、確保解除できます。確保解除したオブジェクト・コピーは、解放できます。キャッシュのメモリが残り少なくなった場合は、オブジェクト・コピーを完全に確保解除すると同時にマーク解除し、キャッシュ内のそのコピーを暗黙的解放の対象にする必要があります。オブジェクト・コピーを完全に確保解除するには、N 回確保した場合は N 回確保解除する必要があります。

確保解除済みでマーク未解除のオブジェクト・コピーは、フラッシュするか、ユーザーが明示的にマーク解除しないと暗黙的解放の対象になりません。ただし、オブジェクト・キャッシュのメモリが残り少なくなると、オブジェクト・キャッシュのオブジェクト・コピーは暗黙的に解放されるので、確保解除したオブジェクト・コピーを必ずしも解放する必要はありません。暗黙的に解放されていないオブジェクト・コピーをプログラムで (any または recent オプションを指定して) 再度確保した場合、同一のオブジェクト・コピーを入手することになります。

アプリケーションでオブジェクト・コピーを確保解除するには、OCIObjectUnpin() または OCIObjectPinCountReset() をコールします。さらに、プログラムでは、特定の接続に対してキャッシュ内のすべてのオブジェクト・コピーを完全に確保解除するために、OCICacheUnpin() をコールできます。

オブジェクト・コピーの解放

オブジェクト・コピーの解放とは、オブジェクト・キャッシュからオブジェクト・コピーを削除し、そのコピーが使用していたメモリを解放することです。キャッシュのメモリを解放するには、次の 2 つの方法があります。

1. 明示的解放 — プログラムで OCIObjectFree() をコールして、キャッシュからオブジェクト・コピーを明示的に解放または削除します。この関数は、マークまたは確保されたオブジェクト・コピーのいずれかを強制的に解放するオプションがあります。OCICacheFree() をコールしてキャッシュ内のすべてのオブジェクトを解放することもできます。
2. 暗黙的解放 — キャッシュのメモリが残り少なくなった場合は、確保解除済みのオブジェクト・コピーとマーク解除済みのオブジェクト・コピーの両方が、暗黙的に解放されます。マークされた、確保解除済みのオブジェクトは、オブジェクト・コピーをフラッシュまたはマーク解除した場合にのみ、暗黙的解放の対象になります。

関連項目： 詳細は、13-5 ページの「[オブジェクト・キャッシュ・パラメータ](#)」を参照してください。

メモリ管理を目的とする場合は、不要になったオブジェクトをアプリケーションで確保解除することが重要です。確保解除したオブジェクトはキャッシュから削除される対象になるので、必要となるときにキャッシュのメモリをスムーズに解放できます。

OCI では、クライアント側キャッシュにある参照されていないオブジェクトを解放する関数は提供していません。

オブジェクト・コピーの変更

この項では、オブジェクト・コピーをマークする関数とマークを解除する関数を説明します。

オブジェクト・コピーのマーク

オブジェクト・コピーはキャッシュ内でローカルに作成、更新または削除できます。(OCIObjectNew() をコールすることによって) キャッシュ内にオブジェクト・コピーを作成すると、オブジェクト・キャッシュでオブジェクト・コピーに挿入のマークが付けられます。そのため、このオブジェクト・コピーをフラッシュすると、オブジェクトがサーバーに挿入されます。

オブジェクト・コピーがキャッシュ内で更新された場合、ユーザーは、オブジェクト・コピーに更新のマークを付ける (OCIObjectMarkUpdate() をコールする) ことによって、オブジェクト・キャッシュに通知する必要があります。オブジェクト・コピーをフラッシュすると、サーバー内の対応するオブジェクトが、オブジェクト・コピーの値に更新されます。

オブジェクト・コピーが削除された場合、そのオブジェクト・コピーは (OCIObjectMarkDelete() をコールすることにより) オブジェクト・キャッシュ内で削除のマークが付けられます。オブジェクト・コピーをフラッシュすると、サーバー内の対応するオブジェクトが削除されます。マークされたオブジェクト・コピーのメモリは、そのコピーをフラッシュして確保解除しないかぎり解放されません。プログラムで削除マーク付きのオブジェクトを確保すると、参照先のない参照を参照解除した場合と同様のエラーになります。

1 つのオブジェクト・コピーを複数回変更した場合、そのコピーをフラッシュしたときにサーバーのオブジェクトに適用されるのは、最後の変更結果です。たとえば、オブジェクト・コピーを更新した後で削除した場合、オブジェクト・コピーをフラッシュすると、サーバーのオブジェクトは単に削除されます。同様に、オブジェクト・コピーの属性を複数回更新した場合、オブジェクト・コピーをフラッシュしたときにサーバーで更新されるのは、属性の最終的な値です。

プログラムでは、オブジェクト・コピーがオブジェクト・キャッシュにロードされている場合のみ、オブジェクト・コピーを更新済みまたは削除済みとしてマークすることができます。

オブジェクト・コピーのマーク解除

オブジェクト・キャッシュ内のマークされたオブジェクト・コピーは、マーク解除できます。マークされたオブジェクト・コピーをマーク解除すると、オブジェクト・コピーに対して行った変更内容はサーバーにフラッシュされません。オブジェクト・キャッシュのオブジェクト・コピーに対してすでに行ったローカルな変更内容は、取り消されません。

プログラムでオブジェクトをマーク解除するには、OCIObjectUnmark() をコールします。さらに、OCICacheUnmark() をコールして、特定の接続に対して、キャッシュ内のすべてのオブジェクト・コピーをマーク解除することもできます。

オブジェクト・コピーとサーバーとの同期化

この項では、キャッシュとサーバーの同期化操作（フラッシュ、リフレッシュ）を説明します。

変更をサーバーにフラッシュ

キャッシュ内のマークされたオブジェクト・コピーに対するローカルな変更は、オブジェクト・コピーをフラッシュしたときにサーバーに書き込まれます。プログラムで単一のオブジェクト・コピーをフラッシュするには、`OCIObjectFlush()` をコールします。または、`OCICacheFlush()` をコールして、キャッシュ内のすべてのマークされたオブジェクト・コピーをフラッシュするか、リストに選択されているマークされたオブジェクト・コピーをフラッシュできます。`OCICacheFlush()` は、特定のサービス・コンテキストに関連付けられたオブジェクトをフラッシュします。17-9 ページの [OCICacheFlush\(\)](#) を参照してください。

オブジェクト・コピーをフラッシュすると、そのオブジェクト・コピーはマーク解除されません（フラッシュ後にオブジェクトはサーバーでロックされます。そのため、キャッシュ内のオブジェクト・コピーはロック状態とマークされます）。

注意： `OCICacheFlush()` 操作では、複数のオブジェクトがフラッシュされる場合でも、サーバー・ラウンドトリップは 1 回のみです。

ある型の使用済みオブジェクトのみをフラッシュする場合、`OCICacheFlush()` コールのオプションの引数であるコールバック関数を使用して行うことが可能です。目的のオブジェクトのみを戻すコールバックを定義します。この場合でも、フラッシュに対し、サーバー・ラウンドトリップが 1 回のみ発生します。

`OCICacheFlush()` のデフォルト・モードでは、オブジェクトは使用済みとマークされるようにフラッシュされます。このフラッシュ操作のパフォーマンスは、環境ハンドル内で `OCI_ATTR_CACHE_ARRAYFLUSH` 属性を設定すると、非常に向上します。

関連項目： A-3 ページの「[OCI_ATTR_CACHE_ARRAYFLUSH](#)」を参照してください。

ただし、`OCI_ATTR_CACHE_ARRAYFLUSH` モードは、オブジェクトがフラッシュされる順序が重要でない場合にのみ使用することができます。このモード中は、使用済みオブジェクトはまとめてグループ化され、サーバーによる表の更新が効率的に行われるような方法でサーバーに送信されます。このモードが有効なときは、オブジェクトが使用済みとしてマークされる順序の維持は保証されません。

オブジェクト・コピーのリフレッシュ

オブジェクト・コピーをリフレッシュすると、サーバー内の対応するオブジェクトの最新値が、オブジェクト・コピーに再ロードされます。最新値は、その他のコミット済みトランザクションでの変更や、同じトランザクションで（オブジェクト・キャッシュを介さずに）直接サーバー内で行った変更を含む場合があります。プログラムでは、SQL DML またはトリガー、ストアド・プロシージャを使用して、サーバー内のオブジェクトを直接変更できます。

プログラムで、マークされたオブジェクト・コピーをリフレッシュするには、まずオブジェクト・コピーをマーク解除する必要があります。確保解除したオブジェクト・コピーは、リフレッシュすると（つまりキャッシュ全体をリフレッシュすると）単に解放されます。

プログラムで単一のオブジェクト・コピーをリフレッシュするには、OCIObjectRefresh() をコールします。または、OCICacheRefresh() をコールして、キャッシュ内のすべてのオブジェクト・コピー、トランザクションでロードしたすべてのオブジェクト・コピー（つまり、recent または latest オプションを指定して確保したオブジェクト・コピー）、またはリストに選択されたオブジェクト・コピーのいずれかをリフレッシュできます。

オブジェクトをサーバーにフラッシュすると、トリガーを起動してサーバー内のさらに多くのオブジェクトを変更できます。その場合、オブジェクト・キャッシュ内にある（トリガーによって変更したオブジェクトと）同一のオブジェクトは最新の状態ではなくなり、リフレッシュしないとロックまたはフラッシュできません。

各種のメタ属性フラグおよびオブジェクト継続時間は、表 13-1 で説明するとおり、リフレッシュされた後で変更されます。

表 13-1 リフレッシュ後のオブジェクト属性

オブジェクト属性	リフレッシュ後の状態
既存	適切な値に設定
確保済み	変更なし
フラッシュ済み	リセット
割当て時間	変更なし
確保継続時間	変更なし

リフレッシュ実行時、オブジェクト・キャッシュでは新しいデータがオブジェクト・コピーのトップレベル・メモリにロードされます。つまりトップレベル・メモリは再使用されます。オブジェクト・コピーのトップレベル・メモリには、オブジェクトのインライン属性が入っています。その反対に、アウト・ライン属性はサイズ変更ができるので、オブジェクト・コピーのアウト・ライン属性のメモリは解放して再配置できます。

関連項目： オブジェクト・メモリーの詳細は、13-17 ページの「[インスタンスのメモリー・レイアウト](#)」を参照してください。

オブジェクトのロック

この項では、オブジェクトのロックに関連する OCI 関数について説明します。

ロック・オプション

オブジェクトを確保するときに、そのオブジェクトをロック・オプションでロックするかどうかを指定できます。オブジェクトをロックすると、サーバー側のロックが取得されます。これによって、他のユーザーによるオブジェクトへの変更を防止します。このロックは、トランザクションのコミットまたはロールバック時に解放されます。次の3つのロック・オプションがあります。

- ロック・オプション `OCI_LOCK_NONE` ーロックなしでオブジェクトを確保するようにキャッシュに指示します。
- ロック・オプション `OCI_LOCK_X` ーロックを取得した後でのみオブジェクトを確保するようにキャッシュに指示します。オブジェクトが現在別のユーザーによってロックされている場合、このオプションによる確保コールは、ロックが取得できるまで待機してから、コール元に戻ります。これは、`SELECT FOR UPDATE` 文の実行に相当します。
- ロック・オプション `OCI_LOCK_X_NOWAIT` ーロックを取得した後でのみオブジェクトを確保するようにキャッシュに指示します。`OCI_LOCK_X` オプションとは異なり、`OCI_LOCK_X_NOWAIT` オプションによる確保コールは、オブジェクトが現在別のユーザーによってロックされている場合、待機しません。これは、`SELECT FOR UPDATE WITH NOWAIT` 文の実行に相当します。

更新するためのオブジェクトのロック

プログラムでは、オプションで `OCIObjectLock()` をコールして、更新するオブジェクトをロックできます。このコールは、データベースのオブジェクトに対する行ロックの実行をオブジェクト・キャッシュに指示します。これは、次のようにして実行します。

```
SELECT NULL FROM t WHERE REF(t) = :r FOR UPDATE
```

このコード例の `t` は、ロックするオブジェクトを格納しているオブジェクト表であり、`r` はオブジェクトを識別する `REF` です。`OCIObjectLock()` をコールすると、オブジェクト・キャッシュ内のオブジェクト・コピーがロック状態とマークされます。

オブジェクトのグラフまたはオブジェクトの集合をロックするには、オブジェクトごとに `OCIObjectLock()` のコールが必要なため、数回のコールが必要です。配列確保の `OCIObjectArrayPin()` コールを使用すると、パフォーマンスが向上します。

オブジェクトをロックすることにより、キャッシュのオブジェクトが最新のものであることをアプリケーションで保証します。オブジェクトがアプリケーションでロックされている間は、その他のトランザクションはそのオブジェクトを変更できません。

トランザクションの最後には、すべてのロックがサーバーによって自動的に解放されます。オブジェクト・コピーのロック状態インジケータはリセットされます。

NOWAIT オプションを使用したロック

ロックしようとしているオブジェクトが、現在別のユーザーによってロックされている場合があります。この場合、アプリケーションはブロックされます。

オブジェクトをロックする際にブロッキングを避けるためには、アプリケーションで `OCIObjectLock()` コールのかわりに `OCIObjectLockNoWait()` コールを使用できます。この関数では、別のユーザーによるロックのためオブジェクトをロックできない場合は、すぐにエラーが戻されます。

NOWAIT オプションは、`OCI_LOCK_X_NOWAIT` の値をロック・オプション・パラメータとして渡すことで、コールを確保するためにも利用できます。

コミット時ロックの実装

OCI アプリケーションにコミット時ロックを実装するには、2つのオプションを使用します。

コミット時ロックのオプション 1

第1のコミット時ロック・オプションは、シリアル化可能なレベルでトランザクションを実行する OCI アプリケーションのためのオプションです。

OCI では、オブジェクト・キャッシュ内のオブジェクトをロックせずに参照解除して確保し、キャッシュ内でそのオブジェクトを（ロックせずに）変更し、使用済みのオブジェクトをデータベースにフラッシュするコールをサポートしています。

トランザクションの開始以降、使用済みオブジェクトが、別のコミット済みトランザクションによって変更された場合、フラッシュの間に、シリアライズ不能トランザクションのエラーが戻されます。トランザクションの開始以降、別のトランザクションで変更された使用済みオブジェクトがない場合は、変更内容はデータベースに正常に書き込まれます。

注意： `OCITransCommit()` は、最初使用済みオブジェクトをデータベースにフラッシュしてから、トランザクションをコミットします。

この機能により、コミット時ロック・モデルが効果的に実装されます。

コミット時ロックのオプション 2

アプリケーションでは、オブジェクトの変更検出モードを使用可能に切り替えられます。このためには、環境ハンドルの `OCI_ATTR_OBJECT_DETECTCHANGE` 属性の値に `TRUE` を設定します。

このモードがアクティブのときは、別のオブジェクトがコミットしたトランザクションによってサーバー内で変更されているオブジェクトをフラッシュしようとする、アプリケーションに `ORA-08179` エラー「並行性チェックに失敗しました」が戻されます。この後、アプリケーションではこのエラーを適切な方法で処理できます。

オブジェクト・アプリケーションでのコミットおよびロールバック

トランザクションをコミットすると (`OCITransCommit()`)、すべてのマークされたオブジェクトがサーバーにフラッシュされます。トランザクション継続時間を指定して確保したオブジェクト・コピーは、確保解除されます。

トランザクションをロールバックすると、すべてのマークされたオブジェクトがマーク解除されます。トランザクション継続時間を指定して確保したオブジェクト・コピーは、確保解除されます。

オブジェクト継続時間

メモリに空き領域を維持するため、オブジェクト・キャッシュでは、可能なかぎりオブジェクトのメモリの再利用を試みます。オブジェクト・キャッシュでは、オブジェクトの継続時間（割当て時間）またはオブジェクトの確保継続時間が期限切れとなると、オブジェクトのメモリを再使用します。割当て時間は、`OCIObjectNew()` でオブジェクトを作成するときに設定されます。確保継続時間は、`OCIObjectPin()` でオブジェクトを確保するときに設定されます。継続時間値のデータ型は **OCIDuration** です。

注意： オブジェクトの確保継続時間は、オブジェクトの割当て時間より長くは設定できません。

オブジェクトの割当て時間が終わりに達すると、オブジェクトは自動的に削除され、メモリが再使用可能になります。確保継続時間はオブジェクトのメモリを再使用できる時期を示し、キャッシュがいっぱいになるとメモリは再使用されます。

OCI では、次の 2 つの事前定義済みの継続時間をサポートします。

1. トランザクション (`OCI_DURATION_TRANS`)
2. セッション (`OCI_DURATION_SESSION`)

トランザクション継続時間は、トランザクションが終了（コミットまたは異常終了）した時点で期限切れになります。セッション継続時間は、セッションまたは接続が終了した時点で期限切れになります。

アプリケーションでは、`OCIObjectUnpin()` を使用してオブジェクトを明示的に確保解除できます。個々のオブジェクトの明示的な確保解除を最小限にするには、関数 `OCICacheUnpin()` を使用して、オブジェクト・キャッシュ内で現在確保済みのすべてのオブジェクトを確保解除します。デフォルトでは、すべてのオブジェクトは、確保継続時間終了時に確保を解除されます。

継続時間例

表 13-2 は、アプリケーションで、異なる継続時間を使用する方法を示しています。このアプリケーションでは、1 つの接続と 3 つのトランザクションで、4 つのオブジェクトを作成または確保しています。最初の列はデータベースで実行されるアクションを示し、2 番目の列はそのアクションを実行する関数を示しています。残りの列はアプリケーションの各ポイントでの各種オブジェクトの状態を示しています。

たとえば、オブジェクト 1 は、T2 で期間を接続継続時間として作成されてから、接続が終了する T19 まで存在します。オブジェクト 2 は、T6 でフェッチされた後、期間をトランザクション継続時間として T7 で確保され、トランザクションがコミットされる T9 まで確保されたままとなります。

表 13-2 割当て時間および確保継続時間の例

時間	アプリケーションの アクション	関数	オブジェクト 1	オブジェクト 2	オブジェクト 3	オブジェクト 4
T ₁	接続を確立する	—	—	—	—	—
T ₂	オブジェクト 1 を作成 する - 割当て時間 = 接続	OCIObjectNew()	既存	—	—	—
T ₅	トランザクション 1 を 開始する	OCITransStart()	既存	—	—	—
T ₆	SQL - REF をオブジェク ト 2 にフェッチする	—	既存	—	—	—
T ₇	オブジェクト 2 を確保 する - 確保継続時間 = トランザクション	OCIObjectPin()	既存	確保済み	—	—
T ₈	アプリケーション・ データを処理する	—	既存	確保済み	—	—
T ₉	トランザクション 1 を コミットする	OCITransCommit()	既存	確保解除	—	—
T ₁₀	トランザクション 2 を 開始する	OCITransStart()	既存	—	—	—
T ₁₁	オブジェクト 3 を作成 する - 割当て時間 = トランザクション	OCIObjectNew()	既存	—	既存	—
T ₁₂	SQL - REF をオブジェク ト 4 にフェッチする	—	既存	—	既存	—
T ₁₃	オブジェクト 4 を確保す る - 確保継続時間 = 接続	OCIObjectPin()	既存	—	既存	確保済み

表 13-2 割当て時間および確保継続時間の例（続き）

時間	アプリケーションのアクション	関数	オブジェクト 1	オブジェクト 2	オブジェクト 3	オブジェクト 4
T ₁₄	トランザクション 2 をコミットする	OCITransCommit()	既存	—	削除済み	確保済み
T ₁₅	セッション 1 を終了する	OCIDurationEnd()	既存	—	—	確保済み
T ₁₆	トランザクション 3 を開始する	OCITransStart()	既存	—	—	確保済み
T ₁₇	アプリケーション・データを処理する	—	既存	—	—	確保済み
T ₁₈	トランザクション 3 をコミットする	OCITransCommit()	既存	—	—	確保済み
T ₁₉	接続を終了する	—	削除済み	—	—	確保解除

関連項目：

- これらの関数に渡すことができるパラメータ値の詳細は、[第 17 章「OCI のナビゲーション関数と型関数」](#)の OCIObjectNew() および OCIObjectPin() の説明を参照してください。
- 割当て時間が期限切れになる前にオブジェクト・メモリを解放する場合の詳細は、10-32 ページの「[オブジェクトの作成](#)」を参照してください。

インスタンスのメモリ・レイアウト

メモリーのインスタンスは、インスタンスのトップレベル・メモリー・チャンク、NULL インジケータ構造体のトップレベル・メモリー、およびオプションとして多数の 2 次メモリー・チャンクで構成されています。たとえば、次のような DEPARTMENT という行の型を想定します。

```
CREATE TYPE department AS OBJECT
( dep_name      varchar2(20),
  budget        number,
  manager       person,           /* person is an object type */
  employees     person_array );  /* varray of person objects */
```

この C 表現は次のようになります。

```
struct department
{
    OCIStrng * dep_name;
    OCINumber budget;
    struct person manager;
    OCIArray * employees;
};
typedef struct department department;
```

DEPARTMENT の各インスタンスには、dep_name、budget、manager、employees などのトップレベル属性が含まれたトップレベル・メモリー・チャンクがあります。dep_name 属性と employees 属性は、実際にはそれ自体が追加メモリー（2 次メモリー・チャンク）へのポインタです。2 次メモリーは、埋込みインスタンスの実データを入れるために使用されます（employees の VARRAY や dep_name の文字列など）。

NULL インジケータ構造体のトップレベル・メモリーには、インスタンスのトップレベル・メモリー・チャンクにある属性の NULL 状態が入っています。この例では、NULL インジケータ構造体のトップレベル・メモリーに、属性 dep_name、budget、manager の NULL 状態と属性 employees のアトミック NULL 状態が含まれています。

オブジェクト・ナビゲーション

この項では、OCI アプリケーションでオブジェクト・キャッシュ内のオブジェクトのグラフをナビゲートする方法を説明します。

単純なオブジェクト・ナビゲーション

これまでの項に示した例では、アプリケーションで取り出したオブジェクトは単純なオブジェクトであり、その属性はすべてスカラー値でした。アプリケーションで、別のオブジェクトへの REF である属性を持つオブジェクトを取り出す場合は、OCI コールを使用してオブジェクト・グラフを横断し、参照対象のインスタンスにアクセスできます。

たとえば、次の宣言でデータベースに新しい型を定義したとします。

```
CREATE TYPE person_t AS OBJECT
(
    name          VARCHAR2(30),
    mother        REF person_t,
    father        REF person_t);
```

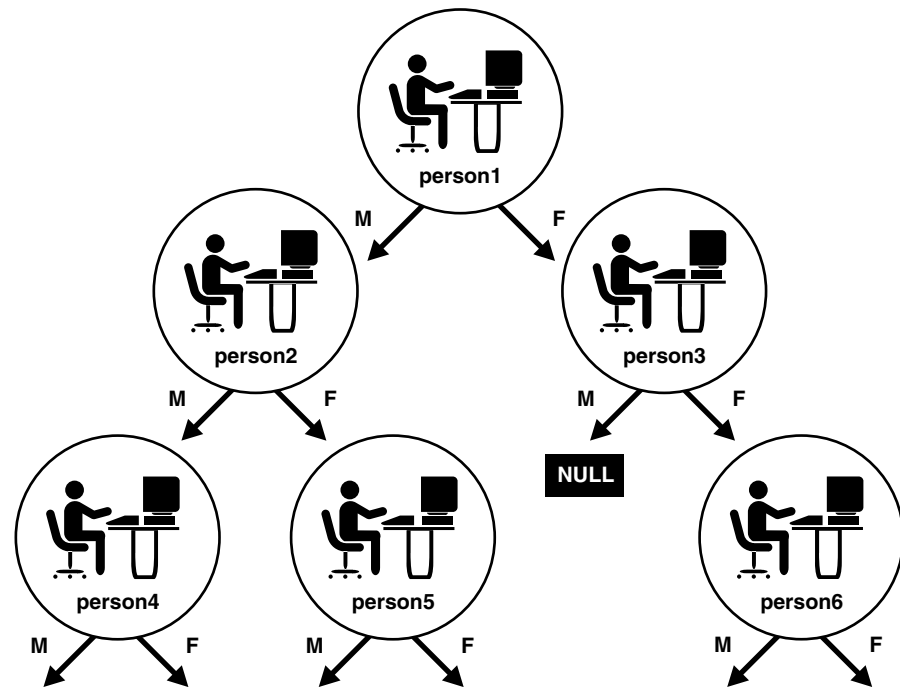
person_t オブジェクトのオブジェクト表は、次の文を使用して作成します。

```
CREATE TABLE person_table OF person_t;
```

これで `person_t` 型のインスタンスをこの型指定表に格納できます。`person_t` の各インスタンスには、同じ表に格納される他の 2 つのオブジェクトへの参照が含まれています。NULL 参照は、親に関する情報がないことを表します。

オブジェクト・グラフは、オブジェクト・インスタンス間の REF リンクを視覚的に表現したものです。たとえば、次のページの図 13-2 「`person_t` インスタンスのオブジェクト・グラフ」に、1 オブジェクトから別のオブジェクトへのリンクを示す `person_t` インスタンスのオブジェクト・グラフを示します。円はオブジェクトを表し、矢印は他のオブジェクトへの参照を表しています。

図 13-2 `person_t` インスタンスのオブジェクト・グラフ



この場合、各オブジェクトは、同一オブジェクトの他の 2 つのインスタンスへのリンクを持ちます。これ以外のケースも考えられます。オブジェクトがその他のオブジェクト型へのリンクを持つ場合もあります。その他の型のグラフも可能です。たとえば、オブジェクトの集合をリンク・リストとして実装する場合は、オブジェクト・グラフを単純な連鎖とみなすことができます。この場合、各オブジェクトは、リンク・リストの前にあるオブジェクトまたは次にあるオブジェクト（あるいはその両方）を参照します。

この章の前半で説明した方法を使用して、`person_t` インスタンスへの参照を取り出し、次にそのインスタンスを確保できます。OCI には、1 つのオブジェクトから別のオブジェクトへ参照をたどることでオブジェクト・グラフを横断できる機能があります。

たとえば、アプリケーションで前述のグラフの `person1` インスタンスをフェッチし、`pers_1` として確保するとします。確保されると、アプリケーションでは、`person1` の `mother` インスタンスにアクセスし、2 回目の PIN オペレーションで `pers_2` に確保することができます。

```
OCIObjectPin(env, err, pers_1->mother, OCI_PIN_ANY, OCI_DURATION_TRANS,
             OCI_LOCK_X, (OCIComplexObject *) 0, &pers_2);
```

この場合、2 つ目のインスタンスを取り出すための OCI のフェッチ操作は必要ありません。

アプリケーションでは、次に、`person1` の `father` インスタンスを確保するか、`person2` の参照リンクで操作できます。

注意： NULL または参照先がない REF を確保しようとすると、`OCIObjectPin()` コールがエラーになります。

OCI ナビゲーション関数

この項では、OCI ナビゲーション関数をまとめて簡単に説明します。関数は一般的な機能別にグループ化してあります。

関連項目： これらの関数の詳細は、[第 17 章「OCI のナビゲーション関数と型関数」](#)を参照してください。

これらの関数の使用方法については、この章の初めの項で説明しています。

ナビゲーション関数は、機能の種類によって異なる接頭辞を付けるネーミング計画に従っています。

`OCICache*` () — キャッシュ操作の関数

`OCIObject*` () — 個々のオブジェクトを操作する関数

確保 / 確保解除 / 解放関数

オブジェクトの確保、確保解除または解放を行うには、次の関数を使用します。

関数	用途
<code>OCICacheFree()</code>	キャッシュの全インスタンスを解放します。
<code>OCICacheUnpin()</code>	キャッシュまたは接続で永続オブジェクトを確保解除します。
<code>OCIObjectArrayPin()</code>	参照の配列を確保します。
<code>OCIObjectFree()</code>	スタンドアロン・インスタンスを解放し、確保解除します。
<code>OCIObjectPin()</code>	オブジェクトを確保します。
<code>OCIObjectPinCountReset()</code>	オブジェクトを確保解除して確保カウントを 0（ゼロ）にします。
<code>OCIObjectPinTable()</code>	継続時間を指定して表オブジェクトを確保します。
<code>OCIObjectUnpin()</code>	オブジェクトを確保解除します。

フラッシュおよびリフレッシュ関数

変更したオブジェクトをサーバーにフラッシュするには、次の関数を使用します。

関数	用途
<code>OCICacheFlush()</code>	キャッシュ内の変更済み永続オブジェクトをサーバーにフラッシュします。
<code>OCIObjectFlush()</code>	単一の変更済み永続オブジェクトをサーバーにフラッシュします。
<code>OCICacheRefresh()</code>	キャッシュ内の確保済み永続オブジェクトをリフレッシュします。
<code>OCIObjectRefresh()</code>	単一の永続オブジェクトをリフレッシュします。

マークおよびマーク解除関数

アプリケーションでメタ属性の 1 つを変更してオブジェクトのマークまたはマーク解除を行うには、次の関数を使用します。

関数	用途
OCIObjectMarkDelByRef()	REF を指定してオブジェクトを削除済みにマークします。
OCIObjectMarkUpd()	オブジェクトに更新済み / 使用済みのマークを付けます。
OCIObjectMarkDel()	オブジェクトに削除済みのマークを付けます。または値インスタンスを削除します。
OCICacheUnmark()	キャッシュ内の全オブジェクトをマーク解除します。
OCIObjectUnmark()	指定のオブジェクトを更新済みにマークします。
OCIObjectUnmarkByRef()	REF を指定してオブジェクトを更新済みにマークします。

オブジェクトのメタ属性アクセッサ関数

アプリケーションでオブジェクトのメタ属性にアクセスするには、次の関数を使用します。

関数	用途
OCIObjectExists()	インスタンスの存在状況を取得します。
OCIObjectFlushStatus()	インスタンスのフラッシュ状態を取得します。
OCIObjectGetInd()	インスタンスの NULL インジケータ構造体を取得します。
OCIObjectIsDirtied()	オブジェクトに更新済みのマークが付いているかどうか調べます。
OCIObjectIsLocked()	オブジェクトはロック状態かどうか調べます。

その他の関数

次の関数は、OCI アプリケーションにその他のオブジェクト機能を提供します。

関数	用途
<code>OCIObjectCopy()</code>	1 つのインスタンスを別のインスタンスにコピーします。
<code>OCIObjectGetObjectRef()</code>	指定のオブジェクトへの参照を戻します。
<code>OCIObjectGetTypeRef()</code>	インスタンスの TDO への参照を取得します。
<code>OCIObjectLock()</code>	永続オブジェクトをロックします。
<code>OCIObjectLockNoWait()</code>	NOWAIT モードでオブジェクトをロックします。
<code>OCIObjectNew()</code>	新規インスタンスを作成します。

型の変更とオブジェクト・キャッシュ

型情報を型名に基づいて要求すると、OCI では、その型の最新バージョンに対応する型記述子オブジェクト (TDO) を戻します。サーバーとオブジェクト・キャッシュ間は同期化されていません。したがって、オブジェクト・キャッシュ内の TDO は、更新されていない可能性があります。

オブジェクトの確保時に、イメージのバージョンが TDO のバージョンと異なっている場合があります。この場合は、エラーが発生します。アプリケーションを強制終了するか、TDO をリフレッシュしてオブジェクトを再度確保します。アプリケーションを続行すると、アプリケーションが失敗する可能性があります。これは、イメージと TDO のバージョンが同じであっても、アプリケーションで定義されているオブジェクト構造体 (つまり、C 構造体) に新しい型のバージョンとの互換性があるという保証ができないためです。サーバー内の型の属性が削除されている場合は、特にこの可能性があります。

したがって、型の構造体を変更した場合は、変更された型のヘッダー・ファイルを再生成し、アプリケーションを変更し、再コンパイルおよび再リンクしてから、プログラムを再実行する必要があります。

関連項目： 10-41 ページ [「型の変更」](#)

Object Type Translator (OTT)

この章では、Object Type Translator (OTT) について説明します。OTT を使用すると、データベース・オブジェクト・タイプや名前付きコレクション型を C 構造体にマップして、OCI アプリケーションおよび Pro*C/C++ アプリケーションで使用できます。この章は、次の項目で構成されています。

- OTT の概要
- Object Type Translator の使用方法
- OTT のコマンドライン
- Intype ファイル
- OTT データ型マッピング
- Outtype ファイル
- OCI アプリケーションでの OTT の使用方法
- OTT リファレンス

関連項目： Pro*C/C++ 固有の詳細は、『Pro*C/C++ Precompiler プログラムーズ・ガイド』を参照してください。

OTT の概要

Object Type Translator (OTT) は、Oracle サーバーでユーザー定義型を使用する C 言語アプリケーションを開発するのに役立ちます。

SQL の CREATE TYPE 文を使用すると、オブジェクト型を作成できます。これらの型の定義をデータベースに格納しておき、データベースの表を作成するときに使用できます。これらの表を作成すると、OCI または Pro*C/C++ のプログラマは、表に格納されたオブジェクトにアクセスできます。

オブジェクト・データにアクセスするアプリケーションでは、データをホスト言語形式で表現する必要があります。これは、オブジェクト型を C 構造体として表現することによって実現できます。プログラマがデータベース・オブジェクト・タイプを表す構造体宣言を、手入力でコーディングすることは可能です。しかし、多くの型がある場合、この作業は時間がかかり、エラーを生む原因になることも少なくありません。OTT を使用して適切な構造体宣言を自動的に生成することにより、このステップを単純化できます。Pro*C/C++ の場合、OTT で生成したヘッダー・ファイルをアプリケーションに組み込むだけで済みます。OCI のアプリケーションでは、OTT で生成された初期化関数をコールします。

OTT は、格納済みのデータ型を表す構造体を作成するのみでなく、オブジェクト型またはそのフィールドが NULL かどうかを示すパラレル・インジケータ構造体も生成します。

Object Type Translator の使用方法

Object Type Translator (OTT) は、オブジェクト型と名前付きコレクション型のデータベース定義を、OCI または Pro*C/C++ アプリケーションに組み込むことができる C 構造体宣言に変換します。

OTT を明示的に起動し、データベース型を C 表現に変換する必要があります。

ほとんどのオペレーティング・システムでは、コマンドラインから OTT を起動します。

intype ファイルを入力として取得し、outtype ファイル、1 つ以上の C のヘッダー・ファイルおよびオプションの実装ファイルを生成します。OTT を起動するコマンドの例に示します。

```
ott userid=scott/tiger intype=demo.in.typ outtype=demo.out.typ code=c \
hfile=demo.h initfile=demo.c
```

このコマンドで、OTT をユーザー名 'scott' とパスワード 'tiger' でデータベースに接続し、intype ファイル (demo.in.typ) の指示に従ってデータベース型を C 構造体に変換します。その結果生成された構造体は、code パラメータで指定したホスト言語 (C 言語) 用としてヘッダー・ファイル (demo.h) に出力されます。outtype ファイル (demo.out.typ) では、その変換についての情報を受け取ります。

実装ファイル (demo.c) には、変換済みのユーザー定義型の情報を含む型バージョン表を初期化する関数を格納します。

各パラメータについては、この章の後の項で詳しく説明します。

demoin.typ ファイルの例

```
CASE=LOWER
TYPE employee
```

demoout.typ ファイルの例

```
CASE = LOWER
TYPE SCOTT.EMPLOYEE AS employee
  VERSION = "$8.0"
  HFILE = demo.h
```

この例の demoin.typ ファイルには、TYPE が前に付く変換対象の型 (TYPE employee など) があります。outtype ファイルの構造体は、intype ファイルに似ており、OTT で取得した情報が追加されます。

OTT による変換が終了すると、ヘッダー・ファイルには、intype ファイルで指定した各型の C 構造体表現と、各型に対応した NULL インジケータ構造体が入っています。たとえば、intype ファイルにリストされた employee 型が次のように定義されたとします。

```
CREATE TYPE employee AS OBJECT
(
  name          VARCHAR2(30),
  empno         NUMBER,
  deptno        NUMBER,
  hiredate      DATE,
  salary        NUMBER
);
```

OTT で生成されたヘッダー・ファイル (demo.h) には、その他の項目の中に、次の宣言が含まれます。

```
struct employee
{
  OCIStr * name;
  OCINumber empno;
  OCINumber deptno;
  OCIDate hiredate;
  OCINumber salary;
};
typedef struct emp_type emp_type;

struct employee_ind
{
  OCIInd _atomic;
  OCIInd name;
  OCIInd empno;
  OCIInd deptno;
  OCIInd hiredate;
```

```
OCIInd salary;
};
typedef struct employee_ind employee_ind;
```

このコマンドで生成されたサンプルの実装ファイル `demov.c` には、次の宣言が含まれます。

```
#ifndef OCI_ORACLE
#include <oci.h>
#endif

sword demov(OCIEnv *env, OCIError *err)
{
    sword status = OCITypeVTInit(env, err);
    if (status == OCI_SUCCESS)
        status = OCITypeVTInsert(env, err,
            "SCOTT", 5,
            "EMPLOYEE", 8,
            "$8.0", 4);
    return status;
}
```

`intype` ファイルのパラメータで、生成済み構造体の命名方法を制御します。この例では、構造体名の `employee` がデータベース型名の `EMPLOYEE` と一致しています。構造体名は小文字です。これは、`intype` ファイル内に `CASE=lower` の行があるためです。

構造体宣言（`OCIString` や `OCIInd` など）のデータ型は、特殊なデータ型です。

関連項目： これらの型の詳細は、14-10 ページの「[OTT データ型マッピング](#)」を参照してください。

次の各項では、OTT の使用方法をそれぞれの局面から説明します。

- [データベースでの型の作成](#)
- [OTT の起動](#)
- [OTT のコマンドライン](#)
- [Intype ファイル](#)
- [OTT データ型マッピング](#)
- [NULL インジケータ構造体](#)
- [Outtype ファイル](#)

この章の残りの各項では、OCI での OTT の使用方法を説明します。その後に続く参照の項では、コマンドライン構文、パラメータ、`intype` ファイルの構造、ネストした `#include` ファイルの生成、スキーマ名の使用方法、デフォルトの名前マッピングおよび制限事項を説明します。

データベースでの型の作成

OTT を使用するときの最初のステップは、オブジェクト型または名前付きコレクション型を作成してデータベースに格納することです。そのためには、SQL の `CREATE TYPE` 文を使用します。

関連項目： `CREATE TYPE` 文の詳細は、『Oracle9i SQL リファレンス』を参照してください。

OTT の起動

OTT を起動するステップは、次のとおりです。OTT パラメータは、コマンドラインで、または構成ファイルをコールしたファイルで指定できます。一部のパラメータは、`intype` ファイルでも指定できます。

1 つのパラメータが複数箇所に指定されている場合は、コマンドラインのパラメータ値が `intype` ファイルの値より優先され、`intype` ファイルの値はユーザー定義の構成ファイルの値より優先され、ユーザー定義の構成ファイルの値は、デフォルトの構成ファイルの値より優先されます。

グローバル・オプション、つまり、コマンドライン上のオプションまたはすべての `TYPE` 文の前にある `intype` ファイルの先頭にあるオプションの場合は、コマンドラインの値が `intype` ファイルの値より優先されます。(`intype` ファイルでグローバルに指定できるオプションは、`CASE`、`CODE`、`INITFILE` および `INITFUNC` です。 `HFILE` は指定できません)。 `intype` ファイルに `TYPE` 指定で記述されているオプションは、特定の型のみに適用され、型に通常適用されるコマンドラインのオプションより優先されます。したがって、`TYPE person HFILE=p.h` と入力すると、この値は `person` のみに適用され、コマンドラインの `HFILE` より優先されます。この文はコマンドライン・パラメータとはみなされません。

コマンドライン

コマンドラインに設定されたパラメータ（オプションとも呼ばれます）は、他で設定されたパラメータを上書きします。

関連項目： 詳細は、14-6 ページの「[OTT のコマンドライン](#)」を参照してください。

構成ファイル

構成ファイルは、OTT パラメータが入っているテキスト・ファイルです。ファイル内の非空白行には、1 つのオプションと、それに対応付けられた 1 つ以上の値が入っています。1 行に 2 つ以上のパラメータを指定した場合は、最初のパラメータのみが使用されます。構成ファイルの非空白行では、空白は使用できません。

構成ファイルは、コマンドラインで名前を付けることができます。さらに、デフォルトの構成ファイルは常に読み込まれます。このデフォルトの構成ファイルは常に存在している必要がありますが、空でもかまいません。デフォルトの構成ファイルの名前は `ottcfg.cfg` で

あり、構成ファイルの位置はシステム固有の設定です。たとえば、Solaris オペレーティング環境でのファイル指定は、\$ORACLE_HOME/precomp/admin/ottcfg.cfg です。詳細は、使用しているプラットフォームのマニュアルを参照してください。

INTYPE ファイル

intype ファイルには、OTT が変換するユーザー定義型のリストが指定されています。

パラメータの CASE、HFILE、INITFUNC および INITFILE は、intype ファイルに含めることができます。

関連項目： 詳細は、14-8 ページの「[Intype ファイル](#)」を参照してください。

OTT のコマンドライン

ほとんどのプラットフォームでは、コマンドラインから OTT を起動します。入出力ファイルやデータベース接続情報などを指定できます。プラットフォームでの OTT の起動方法については、使用しているプラットフォーム固有のマニュアルを参照してください。

OTT コマンドライン起動の例

コマンドラインから OTT を起動する例を次に示します。

```
ott userid=bren/bigkitty intype=demo.in typ outtype=demo.out typ code=c \  
hfile=demo.h initfile=demo.c
```

注意： 等号 (=) の両側には空白を挿入しないでください。

次の各項では、この例で使用しているコマンドラインの要素を説明します。

関連項目： 様々な OTT コマンドライン・オプションの詳細は、14-25 ページの「[OTT リファレンス](#)」を参照してください。

OTT

OTT を起動します。必ずコマンドラインの先頭に置きます。

USERID

OTT が使用するデータベース接続情報を指定します。

例の OTT は、ユーザー名 'bren' とパスワード 'bigkitty' で接続を試みています。

INTYPE

使用する `intype` ファイルの名前を指定します。

例では、`intype` ファイルの名前が `demo.in.typ` に指定されています。

OUTTYPE

`outtype` ファイルの名前を指定します。OTT で C ヘッダー・ファイルを生成すると、変換された型の情報が `outtype` ファイルに書き込まれます。このファイルには、変換された各型のエントリが、バージョン文字列および C 表現を書き込んだヘッダー・ファイルとともに含まれています。

14-6 ページの「[OTT コマンドライン起動の例](#)」では、`outtype` ファイルの名前が `demo.out.typ` に指定されています。

注意： `outtype` のキーワードで指定されたファイルがすでに存在している場合、そのファイルは、OTT が実行されると上書きされます。
`outtype` ファイル名と `intype` ファイル名が同じ場合は、`outtype` ファイル内の情報で `intype` ファイルが上書きされます。

CODE

変換の目標言語を指定します。次のオプションがあります。

- `C` (ANSI_C と等価)
- `ANSI_C` (ANSI C 対応)
- `KR_C` (Kernighan & Ritchie C 対応)

現在はデフォルト・オプションがないため、このパラメータは必須です。

構造体の宣言は、両方の C 言語において同一です。INITFILE ファイルに定義されている初期化関数のスタイルは、`KR_C` が使用されているかどうかによって異なります。INITFILE オプションが使用されていない場合、3 つのオプションはすべて等価です。

HFILE

生成された構造体を書き込む C ヘッダー・ファイルの名前を指定します。

14-6 ページの「[OTT コマンドライン起動の例](#)」では、生成される構造体は `demo.h` と呼ばれるファイルに格納されます。

注意： hfile キーワードで指定されたファイルがすでに存在している場合、そのファイルは、次の例外を除いて、OTT が実行されると上書きされます。OTT によって生成されたファイルの内容が、そのファイルの前の内容と同一の場合、OTT はファイルへの実際の書込みは行いません。これにより、ファイルの変更時間を節約でき、UNIX の make および他のプラットフォームの類似した機能で、不必要な再コンパイルが実行されません。

INITFILE

型の初期化関数が格納される C ソース・ファイルの名前を指定します。

注意： initfile キーワードで指定されたファイルがすでに存在している場合、そのファイルは、次の例外を除いて、OTT が実行されると上書きされます。OTT によって生成されたファイルの内容が、そのファイルの前の内容と同一の場合、OTT はファイルへの実際の書込みは行いません。これにより、ファイルの変更時間を節約でき、UNIX の make および他のプラットフォームの類似した機能で、不必要な再コンパイルが実行されません。

Intype ファイル

intype ファイルは、OTT の実行時に変換するデータベース型を OTT に指示し、生成された構造体のネーミングも制御します。intype ファイルは、ユーザー作成ファイルでも、前回 OTT を起動したときの outtype ファイルでもかまいません。intype パラメータが使用されていない場合は、OTT の接続先のスキーマにあるすべての型が変換されます。

簡単なユーザー作成 intype ファイルの例を次に示します。

```
CASE=LOWER
TYPE employee
    TRANSLATE SALARY$ AS salary
        DEPTNO AS department
TYPE ADDRESS
TYPE item
TYPE "Person"
TYPE PURCHASE_ORDER AS p_o
```

第 1 行の CASE キーワードは、生成された C 識別子を小文字にすることを指示しています。ただし、この CASE オプションは、intype ファイルに明示的に記述されていない識別子にのみ適用されます。そのため、常には employee は employee、ADDRESS は ADDRESS という C 構造体になります。これらの構造体のメンバーは、小文字で名前が付けられます。

関連項目： CASE オプションの詳細は、14-30 ページの「CASE」を参照してください。

TYPE キーワードで始まる行は、データベース内の変換する型を指定しています。この例では、EMPLOYEE、ADDRESS、ITEM、PERSON および PURCHASE_ORDER の各型が相当します。

TRANSLATE...AS キーワードは、オブジェクト型の C 構造体への変換時に、オブジェクト属性の名前変更が必要なことを指定しています。この例では、employee 型の SALARY\$ 属性が salary に変換されます。

最終行の AS キーワードは、オブジェクト型を構造体に変換するとき、名前を変更することを指定しています。この例では、purchase_order というデータベース型を p_o という構造体に変換します。

AS キーワードが、型または属性の名前の変換に使用されていない場合は、その型または属性のデータベース名が C 識別子名として使用されます。ただし、CASE オプションは有効のため、有効な C 識別子文字にマップできない文字は、アンダースコアで置換されます。型または属性名を変換する理由は、次のとおりです。

- 名前に、アルファベット、数字またはアンダースコア以外の文字が含まれている
- 名前が C キーワードと競合する
- 型名が、同じ有効範囲内の別の識別子と競合する。たとえば、プログラムで、異なるスキーマにある同じ名前の 2 つの型を使用する場合に発生します。
- プログラマが別の名前に変更する

OTT では、intype ファイルにリストされていない他の型の変換が必要になる場合があります。これは、OTT では変換前に intype ファイル内の型の依存性が分析され、必要に応じてその他の型が変換されるためです。たとえば、ADDRESS 型が intype ファイルにリストされていなくても、Person 型が ADDRESS 型の属性を所有している場合、Person 型の定義は必須のため、OTT は、ADDRESS を変換します。

TRANSITIVE パラメータの値として FALSE を指定すると、OTT は、intype ファイルに指定されていない型は生成しません。

通常の大 / 小文字の区別がない SQL 識別子は、intype ファイルでは、大 / 小文字のどのような組合せのつづりでもかまいません。引用符は付けません。

CREATE TYPE "Person" などの大 / 小文字を区別して作成した SQL 識別子を参照するには、TYPE "Person" などの引用符を使用します。SQL 識別子は、宣言の際に引用した場合は、大 / 小文字が区別されます。引用符は、TYPE "CASE" などの OTT 予約語である SQL 識別子の参照にも使用できます。このために名前に引用符を付けるときは、その SQL 識別子が CREATE TYPE Case のように大 / 小文字を区別せずに作成されている場合、その名前は大文字にする必要があります。OTT 予約語が SQL 識別子の名前参照用に使われているが、引用符が付けられていない場合、OTT は intype ファイルに構文エラーをレポートします。

関連項目： intype ファイルの構造体の指定と使用可能なオプションの詳細は、14-33 ページの「[Intype ファイルの構造体](#)」を参照してください。

OTT データ型マッピング

OTT でデータベース型から生成した C 構造体には、オブジェクト型の属性ごとに 1 つずつ対応する要素が含まれています。属性のデータ型は、Oracle のオブジェクト・データ型で利用される型にマップされます。Oracle のデータ型には、事前定義済みの基本型の集合が組み込まれており、オブジェクト型やコレクションなどのユーザー定義型の作成をサポートします。

また、Oracle には、オブジェクト型属性を C 構造体で表現するための事前定義済みの型の集合も含まれています。たとえば、次のようなオブジェクト型定義と、OTT で生成した対応する構造体宣言があるとしています。

```
CREATE TYPE employee AS OBJECT
(
  name      VARCHAR2(30),
  empno     NUMBER,
  deptno    NUMBER,
  hiredate  DATE,
  salary$   NUMBER);
```

CASE=LOWER で、型または属性名の明示的なマッピングがないと仮定すると、OTT 出力は次のようになります。

```
struct employee
{
  OCIStrng * name;
  OCINumber empno;
  OCINumber department;
  OCIDate   hiredate;
  OCINumber salary_;
};
typedef struct emp_type emp_type;
struct employee_ind
{
  OCInd _atomic;
  OCInd name;
  OCInd empno;
  OCInd department;
  OCInd hiredate;
  OCInd salary_;
}
typedef struct employee_ind employee_ind;
```

関連項目： インジケータの構造体 (struct employee_ind) は、14-15 ページの「[NULL インジケータ構造体](#)」で説明します。

ここでは、構造体宣言のデータ型、**OCIStrng**、**OCINumber**、**OCIDate** および **OCIInd** を使用してオブジェクト型の属性のデータ型をマップしています。たとえば、empno 属性の NUMBER データ型は、**OCINumber** データ型にマップされます。また、これらのデータ型は、バインド変数および定義変数の型としても使用できます。

オブジェクト・データ型の C へのマッピング

この項では、Oracle のオブジェクト属性型と OTT で生成される C の型とのマッピングを説明します。次の 14-12 ページの「**OTT 型マッピングの例**」では、様々なマッピングの例を示します。次の表には、属性として使用できる型から、OTT で生成されるオブジェクト・データ型へのマッピングをリストします。

表 14-1 オブジェクト型属性のオブジェクト・データ型マッピング

オブジェクト属性の型	C マッピング
BFILE	OCIBFileLocator*
BLOB	OCILobLocator * または OCIBlobLocator *
CHAR(N)、CHARACTER(N)、NCHAR(N)	OCIStrng *
CLOB、NCLOB	OCILobLocator * または OCIClobLocator *
DATE	OCIDate
ANSI DATE	OCIDateTime *
TIMESTAMP、TIMESTAMP WITH TIME ZONE、 TIMESTAMP WITH LOCAL TIME ZONE	OCIDateTime *
INTERVAL YEAR TO MONTH、INTERVAL DAY TO SECOND	OCIInterval *
DEC、DEC(N)、DEC(N,N)	OCINumber
DECIMAL、DECIMAL(N)、DECIMAL(N,N)	OCINumber
FLOAT、FLOAT(N)、DOUBLE PRECISION	OCINumber
INT、INTEGER、SMALLINT	OCINumber
ネストしたオブジェクト型	ネストしたオブジェクト型の C の 名前
NESTED TABLE	OCITable *
NUMBER、NUMBER(N)、NUMBER(N,N)	OCINumber
NUMERIC、NUMERIC(N)、NUMERIC(N,N)	OCINumber

表 14-1 オブジェクト型属性のオブジェクト・データ型マッピング (続き)

オブジェクト属性の型	C マッピング
RAW(N)	OCIRaw *
REAL	OCINumber
REF	OCISRef *
VARCHAR(N)	OCISString *
VARCHAR2(N)、NVARCHAR2(N)	OCISString *
VARRAY	OCIArray *

注意： REF、VARRAY および NESTED TABLE 型の場合、OTT では、`typedef` が生成されます。この `typedef` で宣言された型は、構造体宣言でデータ・メンバーの型として使用されます。次の項「[OTT 型マッピングの例](#)」を参照してください。

オブジェクト型に REF またはコレクション型の属性が含まれている場合は、最初に REF またはコレクション型の `typedef` が生成されます。次に、オブジェクト型に対応する構造体宣言が生成されます。構造体には、REF またはコレクション型へのポインタを型に持つ要素が含まれます。

あるオブジェクト型に、別のオブジェクト型を持つ属性が含まれている場合、OTT ではネストした型が最初に生成されます (TRANSITIVE=TRUE の場合)。次に、オブジェクト型属性が、ネストしたオブジェクト型である型のネストした構造体にマッピングされます。

OTT がオブジェクト以外のデータベース属性の型をマップする Oracle の C データ型は構造体で、**OCIDate** 以外は不透明です。

OTT 型マッピングの例

次の例では、OTT で作成される各種の型のマッピングを示します。

次のようなデータベース型があるとします。

```
CREATE TYPE my_varray AS VARRAY(5) OF integer;

CREATE TYPE object_type AS OBJECT
(object_name    VARCHAR2(20));

CREATE TYPE my_table AS TABLE OF object_type;

CREATE TYPE other_type AS OBJECT (object_number NUMBER);
```



```

CREATE TYPE many_types AS OBJECT
( the_varchar  VARCHAR2(30),
  the_char     CHAR(3),
  the_blob     BLOB,
  the_clob     CLOB,
  the_object   object_type,
  another_ref  REF other_type,
  the_ref      REF many_types,
  the_varray   my_varray,
  the_table    my_table,
  the_date     DATE,
  the_num      NUMBER,
  the_raw      RAW(255));

```

次のような内容の `intype` ファイルがあるとします。

```

CASE = LOWER
TYPE many_types

```

OTT によって次の C 構造体が生成されます。

注意： ここでは構造体についての補足説明として、コメントを付けています。これらのコメントは、実際の OTT 出力の一部ではありません。

```

#ifndef MYFILENAME_ORACLE
#define MYFILENAME_ORACLE

#ifndef OCI_ORACLE
#include <oci.h>
#endif

typedef OCISRef many_types_ref;
typedef OCISRef object_type_ref;
typedef OCIArray my_varray;           /* used in many_types */
typedef OCISTable my_table;          /* used in many_types */
typedef OCISRef other_type_ref;
struct object_type                    /* used in many_types */
{
    OCISString * object_name;
};
typedef struct object_type object_type;

```

```
struct object_type_ind          /*indicator struct for*/
{
    OCIInd _atomic;              /*object_types*/
    OCIInd object_name;
};
typedef struct object_type_ind object_type_ind;

struct many_types
{
    OCIStr *      the_varchar;
    OCIStr *      the_char;
    OCIBlobLocator * the_blob;
    OCIClobLocator * the_clob;
    struct object_type the_object;
    other_type_ref * another_ref;
    many_types_ref * the_ref;
    my_varray *    the_varray;
    my_table *     the_table;
    OCIDate        the_date;
    OCINumber      the_num;
    OCIRaw *       the_raw;
};
typedef struct many_types many_types;

struct many_types_ind          /*indicator struct for*/
{
    OCIInd _atomic;              /*many_types*/
    OCIInd the_varchar;
    OCIInd the_char;
    OCIInd the_blob;
    OCIInd the_clob;
    struct object_type_ind the_object;          /*nested*/
    OCIInd another_ref;
    OCIInd the_ref;
    OCIInd the_varray;
    OCIInd the_table;
    OCIInd the_date;
    OCIInd the_num;
    OCIInd the_raw;
};
typedef struct many_types_ind many_types_ind;

#endif
```

intype ファイルでは変換対象として 1 つの項目しか指定していませんが、2 つのオブジェクト型と 2 つの名前付きコレクション型が変換されていることに注意してください。これは、OTT パラメータ (14-31 ページの「[TRANSITIVE](#)」) のデフォルト値が TRUE のためです。

前述の項で説明しているとおり、TRANSITIVE=TRUE の場合、OTT は、リストされている型の変換を完了するために、変換対象の型の属性に使用されている型をすべて自動的に変換します。

この自動変換は、オブジェクト型属性のポインタまたは参照によってのみアクセスされる型には適用されません。たとえば、many_types 型には属性として another_ref REF other_type がありますが、struct other_type は生成されません。

この例では、VARRAY、NESTED TABLE および REF の各型を宣言するために、typedef がどのように使用されているかも示しています。

typedef は、始めに使用されます。

```
typedef OCISRef many_types_ref;
typedef OCISRef object_type_ref;
typedef OCIArray my_varray;
typedef OCISTable my_table;
typedef OCISRef other_type_ref;
```

構造体 many_types では、次のように VARRAY、NESTED TABLE および REF の各属性が宣言されています。

```
struct many_types
{
    ...
    other_type_ref *   another_ref;
    many_types_ref *   the_ref;
    my_varray *        the_varray;
    my_table *         the_table;
    ...
}
```

NULL インジケータ構造体

OTT でデータベース・オブジェクト・タイプを表現する C 構造体が生成されるたびに、対応する NULL インジケータ構造体も生成されます。C 構造体にオブジェクト型を選択するとき、パラレル構造体の中に NULL インジケータ情報を選択できます。

たとえば、前の項目の例では次の NULL インジケータ構造体が生成されます。

```
struct many_types_ind
{
    OCISInd _atomic;
    OCISInd the_varchar;
    OCISInd the_char;
    OCISInd the_blob;
    OCISInd the_clob;
    struct object_type_ind the_object;
    OCISInd another_ref;
    OCISInd the_ref;
```

```
OCIInd the_varray;  
OCIInd the_table;  
OCIInd the_date;  
OCIInd the_num;  
OCIInd the_raw;  
};  
typedef struct many_types_ind many_types_ind;
```

NULL インジケータ構造体のレイアウトは重要です。構造体の第1 要素（_atomic）は、アトミック NULL インジケータです。この値は、オブジェクト型全体の NULL 状態を示します。このアトミック NULL インジケータの後に、OTT で生成した、オブジェクト型を表現する構造体の各要素に対応するインジケータ要素が続きます。

あるオブジェクト型の定義の一部として別のオブジェクト型が含まれている場合（この例では object_type 属性）、その属性のインジケータ・エントリは、ネストしたオブジェクト型（TRANSITIVE=TRUE の場合）に対応している NULL インジケータ構造体（object_type_ind）です。

VARRAY と NESTED TABLE には、各要素の NULL 情報が含まれています。

NULL インジケータ構造体のその他の要素のデータ型は、すべて **OCIInd** です。

関連項目： アトミック NULL の詳細は、10-30 ページの「[NULL かどうか](#)」を参照してください。

OTT による型の継承のサポート

オブジェクトの型の継承をサポートするために、OTT は、新しい属性を宣言する前に、継承される属性を、特別な名前 **_super** を持つカプセル化された構造体内で宣言することで、オブジェクトのサブタイプを表す C 構造体を生成します。これによって、スーパータイプを継承するオブジェクトのサブタイプについては、その構造体の第1 要素が **_super** と命名され、サブタイプの各属性に対応する要素がその後に続きます。この **_super** と命名された要素の型がスーパータイプの名前です。

たとえば、サブタイプ **Student_t** と **Employee_t** を持つ **Person_t** 型は、次のように作成されます。

```
CREATE TYPE Person_t AS OBJECT  
( ssn      NUMBER,  
  name     VARCHAR2(30),  
  address  VARCHAR2(100)) NOT FINAL;  
  
CREATE TYPE Student_t UNDER Person_t  
( deptid  NUMBER,  
  major   VARCHAR2(30)) NOT FINAL;
```

```
CREATE TYPE Employee_t UNDER Person_t
( empid NUMBER,
  mgr   VARCHAR2(30));
```

次のような内容の `intype` ファイルがあるとします。

```
CASE=SAME
TYPE EMPLOYEE_T
TYPE STUDENT_T
TYPE PERSON_T
```

OTT では、`Person_t`、`Student_t` および `Employee_t` に対する C 構造体とそれぞれの NULL インジケータ構造体が次のように生成されます。

```
#ifndef MYFILENAME_ORACLE
#define MYFILENAME_ORACLE

#ifndef OCI_ORACLE
#include <oci.h>
#endif

typedef OCISRef EMPLOYEE_T_ref;
typedef OCISRef STUDENT_T_ref;
typedef OCISRef PERSON_T_ref;

struct PERSON_T
{
    OCINumber SSN;
    OCIStrng * NAME;
    OCIStrng * ADDRESS;
};
typedef struct PERSON_T PERSON_T;

struct PERSON_T_ind
{
    OCIInd_atomic;
    OCIInd SSN;
    OCIInd NAME;
    OCIInd ADDRESS;
};
typedef struct PERSON_T_ind PERSON_T_ind;

struct EMPLOYEE_T
{
    PERSON_T _super;
    OCINumber EMPID;
    OCIStrng * MGR;
};
```

```
typedef struct EMPLOYEE_T EMPLOYEE_T;

struct EMPLOYEE_T_ind
{
    PERSON_T _super;
    OCIInd EMPID;
    OCIInd MGR;
};
typedef struct EMPLOYEE_T_ind EMPLOYEE_T_ind;

struct STUDENT_T
{
    PERSON_T _super;
    OCINumber DEPTID;
    OCIStr * MAJOR;
};
typedef struct STUDENT_T STUDENT_T;

struct STUDENT_T_ind
{
    PERSON_T _super;
    OCIInd DEPTID;
    OCIInd MAJOR;
};
typedef struct STUDENT_T_ind STUDENT_T_ind;

#endif
```

前述の C マッピング変換によって、サブタイプのインスタンスから C のスーパータイプのインスタンスへのアップキャストが簡単かつ適切に行われます。たとえば、次のように行います。

```
STUDENT_T *stu_ptr = some_ptr;           /* some STUDENT_T instance */
PERSON_T *pers_ptr = (PERSON_T *)stu_ptr; /* up-casting */
```

NULL インジケータ構造体も同じように生成されます。スーパータイプ `Person_t` の NULL インジケータ構造体の第 1 要素は、`_atomic` です。サブタイプ `Employee_t` と `Student_t` の NULL インジケータ構造体の第 1 要素は、`_super` です（サブタイプに対するアトミック要素は生成されません）。

置換可能なオブジェクト属性

NOT FINAL 型の属性（つまり置換可能な属性）の場合、埋込み属性は、ポインタとして表現されます。

次の Book_t 型を想定します。

```
CREATE TYPE Book_t AS OBJECT
( title  VARCHAR2(30),
  author  Person_t      /* substitutable */);
```

OTT で生成された対応する C 構造体には、Person_t へのポインタが含まれています。

```
struct Book_t
{
    OCISString *title;
    Person_t   *author;    /* pointer to Person_t struct */
}
```

この型に対応する NULL インジケータ構造体は、次のとおりです。

```
struct Book_t_ind
{
    OCIInd _atomic;
    OCIInd title;
    OCIInd author;
}
```

author 属性に対応する NULL インジケータ構造体は、author オブジェクト自体から取得できます。OCIObjectGetInd() を参照してください。

FINAL として定義されている型は、サブタイプを持つことができません。したがって、FINAL 型の属性は、置換可能ではありません。この場合、マッピングは前の状態のままです。つまり、属性の構造体はインラインです。ここで、型を変更し、NOT FINAL として定義すると、マッピングの変更が必要になります。OTT を再実行することで、新しいマッピングが生成されます。

Outtype ファイル

outtype ファイルは、OTT コマンドラインで名前が付けられます。OTT で C ヘッダー・ファイルが生成されると、変換の結果が outtype ファイルに書き込まれます。このファイルには、変換された各型のエントリが、バージョン文字列および C 表現を書き込んだヘッダー・ファイルとともに含まれています。

OTT の 1 回の実行で生成された outtype ファイルは、それ以降の OTT の起動では intype ファイルとして使用できます。

たとえば、この章の前半の例でを使用した単純な intype ファイルを想定します。

```
CASE=LOWER
TYPE employee
    TRANSLATE SALARY$ AS salary
        DEPTNO AS department
TYPE ADDRESS
TYPE item
TYPE "Person"
TYPE PURCHASE_ORDER AS p_o
```

この例では、OTT で生成する C 識別子の大 / 小文字の区別を指定し、変換する型をリストして指定しています。そのうち 2 つの型については、ネーミング規則が指定されています。

OTT 実行後の outtype ファイルは、次のようになります。

```
CASE = LOWER
TYPE EMPLOYEE AS employee
    VERSION = "$8.0"
    HFILE = demo.h
    TRANSLATE SALARY$ AS salary
        DEPTNO AS department
TYPE ADDRESS AS ADDRESS
    VERSION = "$8.0"
    HFILE = demo.h
TYPE ITEM AS item
    VERSION = "$8.0"
    HFILE = demo.h
TYPE "Person" AS Person
    VERSION = "$8.0"
    HFILE = demo.h
TYPE PURCHASE_ORDER AS p_o
    VERSION = "$8.0"
    HFILE = demo.h
```


outtype ファイルの内容を調べると、intype 指定に含まれていなかった型がリストされていることに気づきます。たとえば、intype ファイルでは、次の内容による person 型の変換のみを指定したとします。

```
CASE = LOWER  
TYPE PERSON
```

この person 型の定義に address 型の属性が含まれている場合、outtype ファイルには、PERSON と ADDRESS の両方のエントリが含まれます。person 型を完全に変換するには、最初に address を変換する必要があります。

パラメータ TRANSITIVE が TRUE に設定されている（デフォルトの）場合、OTT は、変換を実行する前に intype ファイル内の型の依存性を分析してから、必要に応じて他の型を変換します。

OCI アプリケーションでの OTT の使用方法

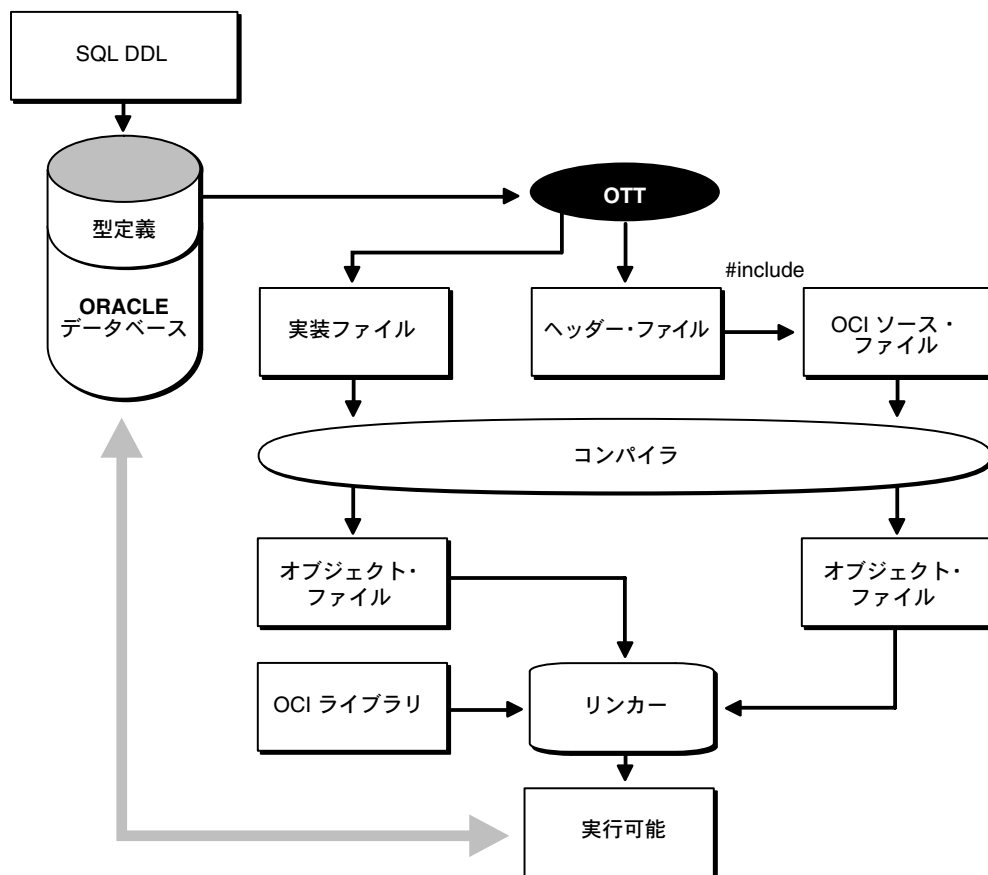
OTT で生成した C ヘッダー・ファイルと実装ファイルは、Oracle サーバー内のオブジェクトにアクセスする OCI アプリケーションで使用できます。ヘッダー・ファイルは、`#include` 文で OCI コードに取り込みます。

OCI アプリケーションでは、ヘッダー・ファイルを組み込んだ後、ホスト言語形式のオブジェクト・データにアクセスし、操作できます。

図 14-1 「OCI アプリケーションでの OTT の使用方法」は、最も単純なアプリケーションの OCI で OTT を使用する場合のステップを示しています。

1. SQL を使用してデータベースに型定義を作成します。
2. OTT で、オブジェクト型と名前付きコレクション型の C 表現を含むヘッダー・ファイルを生成します。INITFILE オプションで命名された実装ファイルも生成します。
3. アプリケーションを記述します。OCI アプリケーションでユーザーが記述したコードで、INITFUNC 関数を宣言してコールします。
4. ヘッダー・ファイルを OCI ソース・コード・ファイルに組み込みます。
5. OTT で生成された実装ファイルを含めて、OCI アプリケーションがコンパイルされ、OCI ライブラリにリンクされます。
6. OCI 実行可能ファイルを Oracle サーバーに対して実行します。

図 14-1 OCI アプリケーションでの OTT の使用方法



OCI でのオブジェクトへのアクセスおよび操作

アプリケーション内部では、OCI プログラムでバインド操作と定義操作を実行できます。そのためには、OTT で生成したヘッダー・ファイルに示されている型で宣言したプログラム変数を使用します。

たとえば、アプリケーションで SQL の `SELECT` 文を使用して、オブジェクトへの `REF` をフェッチし、適切な OCI 関数を使用してそのオブジェクトを確保します。オブジェクトを確保した後、その他の OCI 関数を使用してそのオブジェクトの属性データにアクセスし、操作できます。

OCI には、オブジェクト型の属性と名前付きコレクション型で操作するために特別に設計された、一連のデータ型マッピング関数および操作関数が組み込まれています。

使用可能な関数の例を次に示します。

- `OCIStringSize()` は、**OCIString** 文字列のサイズを取得します。
- `OCINumberAdd()` は、2 つの **OCINumber** の数値を加算します。
- `OCILobIsEqual()` は、2 つの LOB ロケータが等しいかどうかを比較します。
- `OCIRawPtr()` は、**OCIRaw** のロー・データ型へのポインタを取得します。
- `OCICollAppend()` は、コレクション型 (**OCIArray** または **OCITable**) に要素を追加します。
- `OCITableFirst()` は、NESTED TABLE (**OCITable**) の最初の既存要素の索引を戻します。
- `OCIRefIsNull()` は、`REF` (**OCIRef**) が `NULL` かどうかをテストします。

これらの関数の詳細は、このマニュアルの他の章を参照してください。

初期化関数のコール

OTT は、必要に応じて C 初期化関数を生成します。初期化関数では、プログラムで使用されている各オブジェクト型について、どのバージョンの型が使用されているかを環境に通知します。INITFUNC オプションで OTT を起動する場合は、初期化関数の名前を指定するか、初期化関数が含まれている実装ファイル (INITFILE) の名前に基づいてデフォルト名を選択できます。

初期化関数には、環境ハンドル・ポインタとエラー・ハンドル・ポインタの 2 つの引数があります。一般的に、使用する初期化関数は 1 つですが、必ずしもそうである必要はありません。プログラムにコンパイル済みの個別のピースがあり、異なる型を要求する場合は、各ピースに対して初期化関数が含まれた 1 つの初期化ファイルをそれぞれ要求して、OTT を個別に実行できます。

たとえば、`OCIEnvCreate()` をコールして、明示的な OCI オブジェクト・コールによって環境ハンドルを作成した後は、初期化関数も明示的にコールする必要があります。明示的に作成した環境ハンドルそれぞれに対して、必ずすべての初期化関数をコールする必要があります。

ます。これによって、各ハンドルは、プログラム全体で使用しているすべての Oracle データ型にアクセスできます。

EXEC SQL CONTEXT USE や EXEC SQL CONNECT などの埋込み SQL 文を使用して環境ハンドルを暗黙的に作成する場合、ハンドルは暗黙的に初期化されるため、初期化関数をコールする必要はありません。これは、Pro*C/C++ を OCI アプリケーションと結合している場合にのみ適用します。

次に、初期化関数の例を示します。

intype ファイルの `ex2c.typ` に次の内容が含まれているとします。

```
TYPE BREN.PERSON
TYPE BREN.ADDRESS
```

さらに、次のコマンドラインが含まれているとします。

```
ott userid=bren/bigkitty intype=ex2c outtype=ex2co hfile=ex2ch.h initfile=ex2cv.c
```

OTT では、次のファイル `ex2cv.c` が生成されます。

```
#ifndef OCI_ORACLE
#include <oci.h>
#endif

sword ex2cv(OCIEnv *env, OCIError *err)
{
    sword status = OCITypeVTInit(env, err);
    if (status == OCI_SUCCESS)
        status = OCITypeVTInsert(env, err,
            "BREN", 5,
            "PERSON", 6,
            "$8.0", 4);
    if (status == OCI_SUCCESS)
        status = OCITypeVTInsert(env, err,
            "BREN", 5,
            "ADDRESS", 7,
            "$8.0", 4);
    return status;
}
```

関数 `ex2cv` によって、型バージョン表が作成され、`BREN.PERSON` 型と `BREN.ADDRESS` 型が挿入されます。

プログラムで明示的に環境ハンドルを作成する場合、明示的に作成するハンドルごとに初期化関数をコールする必要があるため、すべての初期化関数を生成し、コンパイルし、リンクしてください。プログラムが明示的に環境ハンドルを作成しない場合、初期化関数は必要ありません。

OTT で生成したヘッダー・ファイルを使用するプログラムでは、同時に生成された初期化関数も使用する必要があります。OTT でヘッダー・ファイルを生成し、プログラムで環境ハンドルの明示的に作成した場合は、実装ファイルもコンパイルして、実行可能ファイルにリンクする必要があります。

初期化関数の作業

C 初期化関数は、OTT で処理する型のバージョン情報を提供します。C 初期化関数は、OTT で処理する各オブジェクト・データ型の名前とバージョン識別子を型バージョン表に追加します。

型バージョン表は、Oracle の型マネージャが、特定のプログラムで使用する型のバージョンを特定するときに使用します。異なる時期に OTT で生成した異なる複数の初期化関数によって、型バージョン表に同じ型が複数回追加される可能性があります。型が複数回追加される場合、Oracle では、毎回同じバージョンの型が確実に登録されるようにします。

初期化関数に対して関数プロトタイプを宣言し、その関数をコールするのは、OCI プログラムの役割です。

注意： Oracle のカレント・リリースでは、型ごとにバージョンは 1 つのみです。型バージョン表の初期化は、Oracle の今後のリリースとの互換性を保つためにのみ必要です。

OTT リファレンス

OTT の動作は、OTT コマンドラインまたは CONFIG ファイルに指定するパラメータによって制御します。また、一部のパラメータは、intype ファイルにも指定できます。

この項では、次の項目について詳しく説明します。

- [OTT コマンドラインの構文](#)
- [OTT パラメータ](#)
- [OTT パラメータの指定可能な場所](#)
- [Intype ファイルの構造体](#)
- [ネストされたインクルード・ファイルの生成](#)
- [SCHEMA_NAMES の使用方法](#)
- [デフォルトの名前のマッピング](#)
- [OTT でのファイル名比較の制限事項](#)

この章では、次の規則を使用して OTT の構文を説明します。

- イタリックの文字列は、ユーザーが指定する変数またはパラメータです。
- 大文字の文字列は、そのとおりに入力する文字列です。ただし、大 / 小文字は区別されないため、小文字で入力しても有効です。
- OTT キーワードは、例やヘッダーでは小文字の固定幅フォントでリストされていますが、本文では、目立つように大文字で印刷されています。
- 大カッコ [...] で囲んだ項目は、オプション項目です。
- 1 つの項目（あるいはカッコで囲まれた複数の項目）の直後の省略記号 (...) は、その項目を何度も繰り返し指定できることを示します。
- これ以外の句読点記号は、示されているとおりに入力します。「.」や「@」などが含まれます。

OTT コマンドラインの構文

OTT コマンドライン・インタフェースは、OTT を明示的に起動してデータベース型を C 構造体に変換するときに使用します。オブジェクトを使用する OCI アプリケーションを開発する場合は、必ずこのインタフェースが必要です。

OTT コマンドライン文は、キーワード OTT と、その後続く OTT パラメータのリストによって構成されます。

OTT コマンドライン文に指定できるパラメータは、次のとおりです。

[userid=username/password[@db_name]]

[intype=in_filename]

outtype=out_filename

code=C|ANSI_C|KR_C

[hfile=filename]

[errtype=filename]

[config=filename]

[initfile=filename]

[initfunc=filename]

[case=SAME|LOWER|UPPER|OPPOSITE]

```
[schema_name=ALWAYS|IF_NEEDED|FROM_INTYPE]
```

```
[transitive=TRUE|FALSE]
```

```
[URL=url]
```

注意： 一般に、OTT コマンドの後に続くパラメータの順序は重要ではありません。ただし、OUTTYPE および CODE の各パラメータは常に必須です。

HFILE パラメータは、ほとんどの場合に使用されます。省略すると、HFILE を intype ファイルの型ごとに個別に指定する必要があります。OTT では、intype ファイルにリストされていない型を変換する必要があると判断すると、エラーをレポートします。したがって、HFILE パラメータを省略できるのは、INTYPE ファイルが以前に OTT OUTTYPE ファイルとして生成された場合にかぎります。

intype ファイルを省略すると、スキーマ全体が変換されます。詳細は、この次の項でパラメータの説明を参照してください。

OTT コマンドライン文の例を次に示します。

```
OTT userid=marc/cayman intype=in.typ outtype=out.typ code=c hfile=demo.h  
errtype=demo.tls case=lower
```

OTT コマンドラインの各パラメータについては、この後の各項で説明します。

OTT パラメータ

OTT コマンドラインにパラメータを入力するときの書式は、次のとおりです。

```
parameter=value
```

parameter はリテラル・パラメータ文字列であり、value は有効なパラメータ設定値です。リテラル・パラメータ文字列は大 / 小文字を区別しません。

コマンドラインのパラメータは、空白またはタブのいずれかを使用して区切ります。

パラメータは構成ファイル内にも指定できます。ただし、この場合は、行内の空白は許可されないため、各パラメータは独立した行に指定する必要があります。さらに、パラメータ CASE、HFILE、INITFUNC および INITFILE を intype ファイルに指定できます。

USERID

USERID パラメータでは、Oracle ユーザー名、パスワードおよびオプションのデータベース名（Oracle Net Services のデータベース指定文字列）を指定します。データベース名を省略すると、デフォルトのデータベースが使用されます。このパラメータの構文は、次のとおりです。

```
userid=username/password[@db_name]
```

これが第1パラメータである場合は、「USERID=」を省略して、次のように指定できます。

```
OTT username/password...
```

USERID パラメータはオプションです。これが省略されると、OTT は、ユーザー OPS\$username で自動的にデフォルトのデータベースへの接続を試行します。username は、オペレーティング・システムのユーザー名です。

INTYPE

INTYPE パラメータでは、オブジェクト型指定のリストの読み込み元ファイルの名前を指定します。読み込んだリストにある型が、OTT によって変換されます。

このパラメータの構文は、次のとおりです。

```
intype=filename
```

USERID が第1パラメータ、INTYPE が第2パラメータで、USERID= を省略した場合は、INTYPE= も省略できます。INTYPE が指定されていない場合は、ユーザーのスキーマの型すべてが変換されます。

```
OTT username/password filename...
```

intype ファイルは、型宣言の Make ファイルと考えることができます。C 構造体宣言の必要な型を intype ファイルにリストします。

関連項目： intype ファイルの書式については、14-33 ページの「[Intype ファイルの構造体](#)」で説明します。

コマンドラインまたは intype ファイルのファイル名に拡張子がない場合は、「TYP」や「.typ」などのプラットフォーム固有の拡張子が追加されます。

OUTTYPE

OTT で処理されるすべてのオブジェクト・データ型の型情報を書き込むファイルの名前です。outtype ファイルには、intype ファイルで明示的に指定したすべての型が含まれます。それに加えて、変換の対象である他の型の宣言で使用しているために変換された型が含まれる場合もあります（TRANSITIVE=TRUE の場合）。outtype ファイルは、以後 OTT を起動するときに intype ファイルとして使用できます。


```
outtype=filename
```

INTYPE パラメータと OUTTYPE パラメータが同一のファイルを参照している場合、intype ファイルの古い情報は、新しい INTYPE の情報に置き換えられます。これは、型の変更から開始し、型宣言の生成、ソースコードの編集、プリコンパイル、コンパイル、デバッグに至るサイクルで、同一の intype ファイルを繰り返し使用するとき便利です。

OUTTYPE は必ず指定してください。

コマンドラインまたは intype ファイルのファイル名に拡張子がない場合は、「TYP」や「.typ」などのプラットフォーム固有の拡張子が追加されます。

CODE

CODE=C、CODE=KR_C または CODE=ANSI_C として指定される OTT 出力の目的ホスト言語です。「CODE=C」は、「CODE=ANSI_C」と等価です。

```
CODE=C|KR_C|ANSI_C
```

このパラメータは、デフォルト値がないので必ず指定する必要があります。

INITFILE

INITFILE パラメータでは、OTT で生成した初期化ファイルを書き込むファイルの名前を指定します。このパラメータを省略すると、初期化関数は生成されません。

Pro*C/C++ プログラムの場合、必要な初期化は SQLLIB ランタイム・ライブラリによって実行されるため、INITFILE は不要です。OCI プログラムのユーザーは、INITFILE ファイルをコンパイルおよびリンクし、環境ハンドルの作成時に初期化関数をコールする必要があります。

コマンドラインまたは intype ファイルで指定した INITFILE ファイル名に拡張子がない場合は、「C」や「.c」などのプラットフォーム固有の拡張子が追加されます。

```
initfile=filename
```

INITFUNC

INITFUNC パラメータは、OCI プログラムのみで使用します。OTT で生成する初期化関数の名前を指定します。このパラメータを省略すると、INITFILE の名前から初期化関数の名前が付けられます。

```
initfunc=filename
```

HFILE

インクルード（.h）ファイルの名前を指定します。これは、`intype` ファイルで、型を記述して型のインクルード・ファイルを指定していない場合に、その型を宣言するために OTT によって生成されるインクルード・ファイルです。`intype` ファイルで各型のインクルード・ファイルを個々に指定していない場合、このパラメータは必須です。このパラメータは、`intype` ファイルに記述されていない型が他の型で必要なために生成する場合、これらの他の型が 2 つ以上の異なるファイルで宣言されている場合および `TRANSITIVE=TRUE` の場合も必須です。

コマンドラインまたは `intype` ファイルで指定した `HFILE` ファイル名に拡張子がない場合は、「H」や「.h」などのプラットフォーム固有の拡張子が追加されます。

```
hfile=filename
```

CONFIG

`CONFIG` パラメータでは、共通で使用するパラメータ指定をリストした OTT 構成ファイルの名前を指定します。また、パラメータ指定は、プラットフォームによって異なる位置にあるシステム構成ファイルから読み込まれます。残りのすべてのパラメータ指定は、コマンドラインまたは `intype` ファイルで指定する必要があります。

```
config=filename
```

注意： `ACONFIG` パラメータは、`CONFIG` ファイルでは使用できません。

ERRTYPE

このパラメータを指定すると、`intype` ファイルのリストが、すべての情報メッセージおよびエラー・メッセージとともに `ERRTYPE` ファイルに書き込まれます。情報メッセージおよびエラー・メッセージは、`ERRTYPE` を指定したかどうかに関係なく、標準出力に送信されます。

実質的に、`ERRTYPE` ファイルはエラー・メッセージが追加された `intype` ファイルのコピーです。ほとんどの場合、エラー・メッセージにはエラーの原因となったテキストへのポインタが示されます。

コマンドラインまたは `INTYPE` ファイルで指定した `ERRTYPE` ファイル名に拡張子がない場合は、「TLS」や「.tls」などのプラットフォーム固有の拡張子が追加されます。

```
errtype=filename
```

CASE

このパラメータは、OTT で生成する一部の C 識別子の小 / 大文字の区別に影響を与えます。`CASE` の可能な値は、`SAME`、`LOWER`、`UPPER` および `OPPOSITE` です。`CASE = SAME` の場合は、データベース型と属性名を C 識別子に変換するときに、小 / 大文字は変更されません。`CASE=LOWER` の場合、大文字はすべて小文字に変換されます。`CASE=UPPER` の場合、小文

字はすべて大文字に変換されます。CASE=OPPOSITE の場合、大文字はすべて小文字に変換され、小文字はすべて大文字に変換されます。

```
CASE=[SAME|LOWER|UPPER|OPPOSITE]
```

このオプションは、intype ファイルに記述されていない識別子（明示的にリストされていない属性または型）のみに影響を与えます。大 / 小文字の変換は、正当な識別子が生成された後で行われます。

INTYPE オプションで指定した型の C 構造体識別子の大 / 小文字は、intype ファイルの大 / 小文字と同じです。たとえば、intype ファイルに次の行が含まれるとします。

```
TYPE Worker
```

OTT では次のように生成されます。

```
struct Worker {...};
```

一方で、intype ファイルに次のように記述したとします。

```
TYPE wOrKeR
```

OTT では次のように生成されます。

```
struct wOrKeR {...};
```

これは intype ファイルの大 / 小文字区別どおりです。

intype ファイルに記述されていない、大 / 小文字の区別のない SQL 識別子は、CASE=SAME の場合は大文字で、CASE=OPPOSITE の場合は小文字で指定されます。宣言されるとき引用符が付けられていない SQL 識別子の場合、大 / 小文字の区別はありません。

SCHEMA_NAMES

デフォルト・スキーマに基づいた型のデータベース名を、outtype ファイル内のスキーマ名で修飾する場合は、このオプションで制御できます。OTT で生成した outtype ファイルには、型名も含めて、OTT で処理された型の情報が含まれています。

関連項目： 詳細は、14-37 ページの「[SCHEMA_NAMES の使用方法](#)」を参照してください。

TRANSITIVE

TRUE（デフォルト）または FALSE の値を取得します。intype ファイルで明示的にリストされていない型の依存性に変換対象かどうかを示します。

TRANSITIVE=TRUE が指定されている場合は、他の型で必要で intype ファイルで記述されていない型が生成されます。

TRANSITIVE=FALSE が指定されている場合は、`intype` ファイルに指定されていない型は生成されません。生成した他の型の属性型として使用されている場合でも生成されません。

URL

OTT では、Java インタフェースである JDBC (Java Database Connectivity) を使用してデータベースに接続します。URL パラメータのデフォルト値は次のとおりです。

```
URL=jdbc:oracle:oci8:@
```

OCI8 ドライバは、Oracle のインストールでクライアント側で使用します。Thin ドライバ (Oracle をインストールしない場合にクライアント側で使用する Java ドライバ) は、次のように指定します。

```
URL=jdbc:oracle:thin:@host:port:sid
```

host はデータベースが実行されているホストの名前、*port* はポート番号、*sid* は Oracle SID です。

OTT パラメータの指定可能な場所

OTT パラメータは、コマンドラインまたはコマンドラインで名前が付けられた CONFIG ファイル、あるいはその両方で指定できます。パラメータの一部は、`intype` ファイルでも指定できます。

OTT は、次のように起動します。

```
OTT username/password parameters
```

コマンドラインのパラメータの 1 つが次のパラメータであるとしてします。

```
config=filename
```

構成ファイル `filename` からその他のパラメータが読み込まれます。

さらに、パラメータは、プラットフォームによって異なる位置にあるデフォルトの構成ファイルから読み込まれます。このファイルの存在は必要ですが、空でもかまいません。構成ファイルのパラメータは、スペースなしで、1 行につき 1 つ表示してください。

引数を指定せずに OTT を実行すると、オンライン・パラメータ・リファレンスが表示されます。

変換される OTT の型は、INTYPE パラメータで指定したファイルで命名されます。パラメータ CASE、INITFILE、INITFUNC および HFILE は、`intype` ファイルにも指定できます。OTT で生成した `outtype` ファイルには CASE パラメータを含めることができ、初期化ファイルが生成されている場合は、INITFILE と INITFUNC の各パラメータを含めることができます。`outtype` ファイルでは、型ごとにそれぞれ HFILE を指定します。

OTT コマンドの大 / 小文字区別は、プラットフォームによって異なります。

Intype ファイルの構造体

intype および outtype ファイルは、OTT で変換する型をリストし、型名や属性名を有効な C 識別子に変換する方法を判断するのに必要な情報すべてを提供します。これらのファイルには、1 つ以上の型指定を記述します。また、次のオプションを指定する場合があります。

- CASE
- HFILE
- INITFILE
- INITFUNC

CASE、INITFILE または INITFUNC の各オプションを指定する場合は、すべての型指定よりも前に指定する必要があります。これらのオプションをコマンドラインと intype ファイルの両方に指定した場合は、コマンドラインの値が使用されます。

関連項目： 簡単なユーザー定義の intype ファイルの例、および intype ファイルから OTT で生成される完全な outtype ファイルの例は、14-20 ページの「[Outtype ファイル](#)」を参照してください。

Intype ファイルの型指定

INTYPE での型指定によって、変換対象のオブジェクト・データ型の名前が指定されます。また、outtype での型指定によって、変換済みのオブジェクト・データ型の名前が指定されます。

```
TYPE employee
    TRANSLATE SALARY$ AS salary
        DEPTNO AS department
TYPE ADDRESS
TYPE PURCHASE_ORDER AS p_o
```

型指定の構造体は次のとおりです。[] 内の値は、オプションの入力を示します。

```
TYPE type_name [AS type_identifier]
[VERSION [=] version_string]
[HFILE [=] hfile_name]
[TRANSLATE{member_name [AS identifier]}...]
```

type_name の構文は次のとおりです。

```
[schema_name.]type_name
```

schema_name は、指定のオブジェクト・データ型を所有するスキーマの名前です。type_name は、その型の名前です。デフォルト・スキーマは、OTT を実行するユーザーのデフォルト・スキーマです。デフォルト・データベースは、ローカル・データベースです。

型指定の構成要素を次に説明します。

- `type_name` — オブジェクト・データ型の名前です。
- `type_identifier` — 型を表すのに使用する C 識別子です。省略すると、デフォルトの名前マッピング・アルゴリズムが使用されます。
- `version_string` — OTT の前の起動によるコード生成時に使用した型のバージョン文字列です。バージョン文字列は、OTT によって生成され、`outtype` ファイルに書き込まれます。このファイルは、後で OTT を実行するときに `intype` ファイルとして使用されます。バージョンの文字列は OTT の操作に影響を与えませんが、最終的には実行中のプログラムで使用されるオブジェクト・データ型のバージョンを選択するのに使用されます。
- `type_identifier` — 型を表すのに使用する C 識別子です。省略すると、デフォルトの型マッピング・アルゴリズムが使用されます。

関連項目： 14-39 ページ「[デフォルトの名前のマッピング](#)」

- `hfile_name` — 該当する構造体の宣言またはクラスが現れるヘッダー・ファイルの名前です。`hfile_name` を省略すると、宣言の生成時には、コマンドラインの `HFILE` パラメータで指定したファイルが使用されます。
- `member_name` — 次の `identifier` に変換される属性（データ番号）の名前です。
- `identifier` — ユーザーのプログラムで属性を表現する C 識別子です。識別子は、任意の数の属性に対してこの方法で指定できます。指定していない属性については、デフォルトの名前マッピング・アルゴリズムが使用されます。

オブジェクト・データ型は、次のいずれかの場合に変換する必要があります。

- `intype` ファイルに指定されています。
- 変換が必要な別の型を宣言する場合、および `TRANSITIVE=TRUE` の場合は必須です。

明示的に記述していない型があり、その型が、正確に 1 つのファイルのみに宣言した型で必要だとします。この場合、明示的に記述していない型の変換結果は、それを必要とする明示的に宣言した型と同じファイルに書き込まれます。

明示的に記述していない型が、複数の異なるファイルの型の宣言に必要な場合、この必須の型の変換結果は、グローバルな `HFILE` ファイルに書き込まれます。

ネストされたインクルード・ファイルの生成

OTT で生成した各 HFILE ファイルに、他の必要なファイルを `#includes` で組み込み、ファイル名から構成された記号を `#defines` で定義します。この記号は、HFILE がすでに組み込まれているかどうかを判断するために使用します。たとえば、データベースに次の型があるとしします。

```
create type px1 AS OBJECT (col1 number, col2 integer);
create type px2 AS OBJECT (col1 px1);
create type px3 AS OBJECT (col1 px1);
```

intype ファイルに含まれる情報は次のとおりです。

```
CASE=lower
type px1
  hfile tott95a.h
type px3
  hfile tott95b.h
```

次のようにして OTT を起動します。

```
ott scott/tiger tott95i.typ outtype=tott95o.typ code=c
```

この場合、次の 2 つのヘッダー・ファイルが生成されます。

ファイル tott95b.h は次のとおりです。

```
#ifndef TOTTT95B_ORACLE
#define TOTTT95B_ORACLE
#endif
#include <oci.h>
#endif
#ifndef TOTTT95A_ORACLE
#include "tott95a.h"
#endif
typedef OCISRef px3_ref;
struct px3
{
    struct px1 col1;
};
typedef struct px3 px3;
struct px3_ind
{
    OCIInd _atomic;
    struct px1_ind col1
};
typedef struct px3_ind px3_ind;
#endif
```

ファイル `tott95a.h` は次のとおりです。

```
#ifndef TOTTT95A_ORACLE
#define TOTTT95A_ORACLE
#endif
#include <oci.h>
typedef OCISRef px1_ref;
struct px1
{
    OCINumber col1;
    OCINumber col2;
}
typedef struct px1 px1;
struct px1_ind
{
    OCIInd _atomic;
    OCIInd col1;
    OCIInd col2;
}
typedef struct px1_ind px1_ind;
#endif
```

このファイルでは、`TOTTT95B_ORACLE` という記号を最初に定義しています。そのため、プログラマは、次の構造体を使用して `tott95b.h` を条件付きで組み込むことができます。その場合、`tott95b.h` がインクルード・ファイルに依存しているかどうかを考慮する必要はありません。

```
#ifndef TOTTT95B_ORACLE
#include "tott95b.h"
#endif
```

この方法によって、プログラマは、適当なファイル（たとえば `foo.h`）から `tott95b.h` を組み込むことができます。その際、`tott95b.h` が、`foo.h` に組み込まれるその他のファイルにも組み込まれるかどうかは、わかっていなくてもかまいません。

記号 `TOTTT95B_ORACLE` の定義の後に、ファイル `oci.h` が `#included` によって組み込まれています。`oci.h` は、OTT で生成したすべての HFILE に組み込まれます。`oci.h` には、Pro*C/C++ または OCI のプログラマにとって便利な型と関数の宣言が格納されています。OTT の `#include` で山カッコが使用されるのは、この場合のみです。

次に、ファイル `tott95a.h` を組み込みます。このファイルを組み込む理由は、`tott95b.h` に必要な「`struct px1`」の宣言が入っているからです。ユーザーの `intype` ファイルで、1 つ以上のファイルに型宣言を書き込むように要求すると、OTT では、他のファイルのうちどれを各 HFILE に組み込むかを判断し、必要な `#includes` を生成します。

この OTT の `#include` で、引用符が使用されていることに注意してください。 `tott95b.h` を組み込むプログラムをコンパイルするとき、 `tott95a.h` の検索は、ソース・プログラムが検出された場所から始まり、それ以降は実装定義の検索規則に従います。この方法で `tott95a.h` を検索できない場合、 `intype` ファイルで `tott95a.h` の位置を指定するには、完全なファイル名（/ で始まる UNIX の絶対パス名など）を使用する必要があります。

SCHEMA_NAMES の使用方法

このパラメータは、OTT が接続されたデフォルト・スキーマに基づいて付けた型の名前を、 `outtype` ファイル内のスキーマ名で修飾するかどうかを決定します。

デフォルト・スキーマ以外のスキーマに基づいて付けた型の名前は、常に `outtype` ファイル内のスキーマ名で修飾されます。

スキーマ名で修飾するかしないかで、プログラム実行中に型がどのスキーマで検索されるかが決定します。

次の 3 通りの設定があります。

- `schema_names=ALWAYS` （デフォルト）
`outtype` ファイル内のすべての型名をスキーマ名で修飾します。
- `schema_names=IF_NEEDED`
デフォルト・スキーマに所属している `OUTTYPE` ファイル内の型名はスキーマ名で修飾しません。デフォルト・スキーマ以外のスキーマに属する型名は、スキーマ名で修飾します。
- `schema_names=FROM_INTYPE`
`intype` ファイルに記述されている型は、 `intype` ファイル内のスキーマ名で修飾されている場合のみ、 `OUTTYPE` ファイル内のスキーマ名で修飾します。デフォルト・スキーマに所属している型のうち、 `intype` ファイルで記述されていない型を、型の依存性のために生成する必要がある場合があります。このような型は、その型に依存している、OTT で最初に検出された型がスキーマ名で記述されている場合にのみ、スキーマ名で記述されます。ただし、OTT が接続されたデフォルト・スキーマにない型は、常に明示的に指定したスキーマ名で記述されます。

OTT で生成する `outtype` ファイルは、 `Pro*C/C++` への入力パラメータです。 `Pro*C/C++` の観点からいえば、これは `Pro*C/C++` の `intype` ファイルです。このファイルは、データベース型名を `C` 構造体名と対応付けます。この情報は、構造体内で正しいデータベース型が確実に選択されるようにするために、実行時に使用されます。 `outtype` ファイル（ `Pro*C/C++` の `intype` ファイル）内のスキーマ名で型が指定されている場合、その型は、プログラム実行中に名前付きスキーマ内で検索されます。型がスキーマ名なしで指定される場合、プログラムが接続されているデフォルト・スキーマ内で検索されます。デフォルト・スキーマは、OTT が使用するデフォルト・スキーマと異なる場合があります。

例 : Schema_Names の使用方法

SCHEMA_NAMES に FROM_INTYPE を設定し、intype ファイルで読み込みます。

```
TYPE Person
TYPE david.Dept
TYPE sam.Company
```

OTT で生成した構造体を使用して、Pro*C/C++ アプリケーションで sam.Company、david.Dept および Person の各型を使用します。スキーマ名のない Person を使用して、アプリケーションを接続するスキーマの Person 型を表します。

OTT とアプリケーションの両方がスキーマ david に接続する場合、アプリケーションは、OTT が使用した型と同じ型 (david.Person) を使用します。OTT がスキーマ david に接続し、アプリケーションがスキーマ jana に接続している場合、アプリケーションでは、jana.Person 型を使用します。この動作が有効なのは、スキーマ david とスキーマ jana で同じ CREATE TYPE Person 文が実行された場合のみです。

一方、アプリケーションでは、どのスキーマに接続するかに関係なく david.Dept 型を使用します。この場合、intype ファイルに型名とともにスキーマ名を記述する必要があります。

明示的に指定していない型が、OTT によって変換される場合があります。たとえば、次の SQL 宣言があるとします。

```
CREATE TYPE Address AS OBJECT
( street   VARCHAR2(40),
  city     VARCHAR(30),
  state    CHAR(2),
  zip_code CHAR(10) );

CREATE TYPE Person AS OBJECT
( name     CHAR(20),
  age      NUMBER,
  addr     ADDRESS );
```

OTT がスキーマ david に接続され、SCHEMA_NAMES=FROM_INTYPE が指定されて、ユーザーの intype ファイルに次のいずれかが組み込まれているとします。

```
TYPE Person または TYPE david.Person
```

ただし、david.Address 型は記述されていません。この型は、david.Person 型でネストしたオブジェクトの型として使用されます。intype ファイルに TYPE david.Person を記述してある場合は、TYPE david.Person および TYPE david.Address が outtype ファイルに記述されます。intype ファイルに Type Person を記述してある場合は、TYPE Person および TYPE Address が outtype ファイルに記述されます。

OTT で変換された複数の型に `david.Address` 型が埋め込まれていても、`intype` ファイルに明示的に記述されていない場合、スキーマ名を使用するかどうかは、埋め込まれている `david.Address` 型が OTT で最初に検出された時点で決まります。なんらかの理由で、`david.Address` にはスキーマ名を付け、一方の `Person` 型には付けない場合、ユーザーは明示的に次の内容を要求する必要があります。

```
TYPE      david.Address
```

これは、`intype` ファイルで指定します。

各型を単一のスキーマ内に宣言する通常の場合は、すべての型名を `intype` ファイル内のスキーマ名で修飾するのが最も安全です。

デフォルトの名前のマッピング

OTT でオブジェクト型または属性の C 識別子名を作成する場合、OTT は、その名前をデータベース・キャラクタ・セットから有効な C 識別子に変換します。最初に、名前はデータベース・キャラクタ・セットから OTT で使用するキャラクタ・セットに変換されます。次に、その変換された名前の変換内容が `intype` ファイルに指定されている場合は、`intype` ファイルで指定された変換内容が使用されます。それ以外の場合、OTT では CASE オプションを適用して、その名前を文字ごとにコンパイラのキャラクタ・セットに変換します。次にこのプロセスの詳細を示します。

OTT がデータベース・エンティティの名前を読み込む場合、その名前は、データベース・キャラクタ・セットから OTT で使用するキャラクタ・セットに自動的に変換されます。OTT がデータベース・エンティティの名前を正常に読み込めるように、名前のすべての文字は OTT のキャラクタ・セットに含まれている必要があります。ただし、文字のコードが 2 つのキャラクタ・セットで異なっている場合があります。

必要なすべての文字が、OTT で使用するキャラクタ・セットに確実に含まれるようにする最も簡単な方法は、データベース・キャラクタ・セットの内容と同一にすることです。ただし、OTT のキャラクタ・セットは、コンパイラのキャラクタ・セットのスーパーセットである必要があります。つまり、コンパイラのキャラクタ・セットが 7 ビット ASCII の場合は、OTT のキャラクタ・セットにはサブセットとして 7 ビット ASCII を含む必要があります。コンパイラのキャラクタ・セットが 7 ビット EBCDIC の場合は、OTT のキャラクタ・セットにはサブセットとして 7 ビット EBCDIC を含む必要があります。ユーザーは、OTT で使用するキャラクタ・セットを、環境変数 `NLS_LANG` を設定して指定するか、他のプラットフォーム固有のメカニズムによって指定します。

OTT がデータベース・エンティティの名前を読み込むと、その名前は、OTT で使用するキャラクタ・セットからコンパイラのキャラクタ・セットに変換されます。その名前の変換内容が `INTYPE` ファイルで指定されている場合、OTT はその変換内容を使用します。

それ以外の場合、OTT では、次のように名前の変換を行います。

1. 最初に、OTT のキャラクタ・セットがマルチバイト・キャラクタ・セットの場合で、名前の中のマルチバイト文字に等価のシングルバイト文字がある場合、そのマルチバイト文字をシングルバイト文字に変換します。
2. 次に、その名前を、OTT のキャラクタ・セットからコンパイラのキャラクタ・セットに変換します。コンパイラのキャラクタ・セットは、US7ASCII などのシングルバイト・キャラクタ・セットです。
3. 最後に、有効な CASE オプションに従って、文字の大 / 小文字を設定します。C 識別子内で無効な文字やコンパイラのキャラクタ・セットに変換されない文字は、アンダースコアに置き換えられます。1 文字でもアンダースコアに置き換えられた場合、OTT は警告メッセージを出します。名前の中のすべての文字がアンダースコアに置き換えられた場合、OTT はエラー・メッセージを出します。

文字単位の名前の変換では、コンパイラのキャラクタ・セットにあるアンダースコア、数字またはシングルバイト文字は変更されません。したがって、有効な C 識別子は変更されません。

名前の変換ではたとえば、ウムラウトの付いた「ö」、アクセント符号の付いた「ä」などのシングルバイト文字をそれぞれ「o」や「a」に変換したり、またマルチバイト文字をシングルバイトと等価に変換する場合があります。名前に等価のシングルバイトがないマルチバイト文字がある場合、通常、その名前の変換はエラーとなります。この場合、ユーザーは、intype ファイルで名前の変換を指定する必要があります。

C 言語の同一の名前に複数のデータベース識別子がマッピングされている場合に生じるネーミングの競合は、OTT では検出されません。また、データベース識別子が C キーワードにマッピングされるときにネーミング問題も検出されません。

OTT でのファイル名比較の制限事項

現在、OTT は、コマンドラインまたは intype ファイルに指定したファイル名を比較することで、2 つのファイルが同一かどうかを判断しています。OTT で 2 つのファイル名が同一のファイルを参照しているかどうかを判別する必要がある場合は、1 つの潜在的な問題が生じます。たとえば、OTT で生成されたファイル foo.h で、foo1.h に記述されている型宣言と、/private/elias/foo1.h に記述されている別の型宣言が必要な場合、OTT は、2 つのファイルが同一の場合は 1 つの #include を、異なる場合は 2 つの #include を生成します。しかし、実際には 2 つのファイルは異なるという結論が出され、次のように 2 つの #includes が生成されます。

```
#ifndef FOOL_ORACLE
#include "foo1.h"
#endif
#ifndef FOOL_ORACLE
#include "/private/elias/foo1.h"
#endif
```

`foo1.h` と `/private/elias/foo1.h` が別々のファイルである場合は、最初のファイルのみが組み込まれます。`foo1.h` と `/private/elias/foo1.h` が同一ファイルである場合は、`#include` が重複して記述されます。

そのため、コマンドラインまたは `intype` ファイルでファイルを複数回記述するときは、各記述で正確に同じファイル名を使用する必要があります。

第 III 部

OCI リファレンス

第III部は、OCI 関数リファレンスの章で構成されています。

- [第 15 章「OCI リレーショナル関数」](#)
- [第 16 章「その他の OCI リレーショナル関数」](#)
- [第 17 章「OCI のナビゲーション関数と型関数」](#)
- [第 18 章「OCI のデータ型マッピング関数および操作関数」](#)
- [第 19 章「OCI カートリッジ関数」](#)
- [第 20 章「OCI の任意型関数および任意データ関数」](#)

関連項目：

- グローバル・サポート環境に適用される OCI 関数の詳細は、『Oracle9i Database グローバリゼーション・サポート・ガイド』を参照してください。
- カートリッジ・サービスに適用される OCI 関数の詳細は、『Oracle9i Data Cartridge Developer's Guide』を参照してください。
- アドバンスド・キューイングに関連する OCI コールの詳細は、『Oracle9i アプリケーション開発者ガイド - アドバンスド・キューイング』を参照してください。

OCI リレーショナル関数

この章では、最初に C 言語対応の Oracle OCI リレーショナル関数について説明します。各ファンクション・コールについて詳細に説明するとともに、使用しているアプリケーションで OCI 関数をコールする方法についても解説します。

関連項目： コード例は、Oracle のインストールに含まれているデモ・プログラムを参照してください。追加情報については、[付録 B「OCI デモ・プログラム」](#)を参照してください。

この章は、次の項目で構成されています。

- [リレーショナル関数の概要](#)
- [接続関数、認証関数および初期化関数](#)
- [ハンドル関数および記述子関数](#)
- [バインド関数、定義関数および記述関数](#)

リレーショナル関数の概要

この章では、OCI リレーショナル・ファンクション・コールについて説明します。この章と次の章で説明する関数は、基本 OCI における関数です。

関連項目： リターン・コードおよびエラー処理の詳細は、2-31 ページの「[エラー処理](#)」を参照してください。

関数の構文

各関数について、次の情報をリストします。

用途

この関数によって実行されるアクションを簡単に説明します。

構文

関数の宣言。

パラメータ

この関数の各パラメータの説明。これにはパラメータのモードが含まれます。パラメータのモードには、次の 3 つの値があります。

モード	説明
IN	OCI にデータを渡すパラメータ
OUT	このコールで OCI からデータを受け取るパラメータ
IN/OUT	このコールでデータを渡し、このコールまたは後続のコールからの戻りでデータを受け取るパラメータ

コメント

この関数に関する詳細情報。関数の使用上の制約やアプリケーション内でこの関数を使用するときに役に立つ情報が記載されています。

例

説明の対象となっているファンクション・コールの使用方法を例示するコード（全体または一部）。すべての関数に例が記載されているわけではありません。

関連関数

関連するファンクション・コールのリスト。

OCI 関数のコール

OCI のこれまでのバージョンとは異なり、リリース 8.x 以上の OCI では、ヌル文字で終了する文字列の文字列長パラメータに -1 を渡すことはできません。文字列長をパラメータとして渡すときは、ヌル終端文字バイトを組み込まないでください。OCI では、文字列はヌル文字で終了するとはみなされません。

OCI パラメータのバッファ長は、バイト単位です。ただし、次の場合は除きます。

- いくつかの LOB コール内の `amount` パラメータが文字単位である場合。
- 関数パラメータに UTF-16 エンコーディングのテキストが使用されており、その長さが文字のポイント単位である場合。

LOB 関数用のサーバー・ラウンドトリップ

個々の OCI LOB 関数に必要なサーバー・ラウンドトリップ回数については、[付録 C「OCI 関数のサーバー・ラウンドトリップ」](#)の表を参照してください。

接続関数、認証関数および初期化関数

この項では、OCI 接続関数、認証関数および初期化関数について説明します。

表 15-1 接続関数、認証関数および初期化関数

関数	用途
OCIConnectionPoolCreate() (15-5 ページ)	接続プールを初期化します。
OCIConnectionPoolDestroy() (15-8 ページ)	接続プールを破棄します。
OCIEnvCreate() (15-9 ページ)	OCI 環境を作成および初期化します。
OCIEnvNlsCreate() (15-14 ページ)	OCI 関数が実行される環境を作成および初期化します。この環境の作成時に、キャラクタ・セット ID および各国語キャラクタ・セット ID を設定できます。
OCIEnvInit() (15-12 ページ)	環境ハンドルを初期化します。
OCIInitialize() (15-18 ページ)	OCI プロセス環境を初期化します。
OCILogon() (15-22 ページ)	単純なシングルセッション・ログイン。
OCILogon2() (15-24 ページ)	この関数は、様々なモードでログイン・セッションを作成するために使用します。
OCIServerAttach() (15-27 ページ)	サーバーに連結します。サーバー・コンテキスト・ハンドルを初期化します。
OCIServerDetach() (15-29 ページ)	サーバーから連結解除します。サーバー・コンテキスト・ハンドルを未初期化します。
OCISessionBegin() (15-30 ページ)	ユーザーを認証します。
OCISessionEnd() (15-34 ページ)	ユーザー・セッションを終了します。
OCISessionGet() (15-35 ページ)	セッション・プールからセッションを取得します。
OCISessionPoolCreate() (15-39 ページ)	セッション・プールを初期化します。
OCISessionPoolDestroy() (15-43 ページ)	セッション・プールを破棄します。
OCISessionRelease() (15-44 ページ)	セッションを解放します。
OCITerminate() (15-46 ページ)	共有メモリー・サブシステムから連結解除します。

OCIConnectionPoolCreate()

用途

接続プールを初期化します。

構文

```
sword OCIConnectionPoolCreate ( OCIEnv          *envhp,  
                                OCIError        *errhp,  
                                OCIPool         *poolhp,  
                                OraText         **poolName,  
                                sb4             *poolNameLen,  
                                CONST OraText   *dblink,  
                                sb4             dblinkLen,  
                                ub4             connMin,  
                                ub4             connMax,  
                                ub4             connIncr,  
                                CONST OraText   *poolUsername,  
                                sb4             poolUserLen,  
                                CONST OraText   *poolPassword,  
                                sb4             poolPassLen,  
                                ub4             mode );
```

パラメータ

envhp (IN)

接続プールを作成する環境へのポインタです。

errhp (IN/OUT)

OCIErrorGet() に渡すことができるエラー・ハンドルです。

poolhp (IN)

割当て済みのプール・ハンドルです。

poolName (OUT)

接続先の接続プール名です。

poolNameLen (OUT)

poolName が指し示す文字列の長さです。

dblink (IN)

接続先のデータベース（サーバー）を指定します。

dblinkLen (IN)

dblink が指し示す文字列の長さです。

connMin (IN)

接続プールの最小接続数を指定します。有効な値は 0（ゼロ）以上です。

指定された数の接続が `OCIConnectionPoolCreate()` によってサーバーにオープンされます。以降は、必要なときのみ接続をオープンします。通常、接続数は、アプリケーションで実行を予定している同時文の数に設定する必要があります。

connMax (IN)

データベースに対してオープンできる最大接続数を指定します。最大数に達すると、追加の接続はオープンしません。有効な値は 1 以上です。

connIncr (IN)

現行の接続数が `connMax` 未満の場合、アプリケーションでは、データベースに対してオープンする接続に次の増分を設定できます。有効な値は 0（ゼロ）以上です。

poolUsername (IN)

接続プーリングでは、暗黙的な 1 次セッションが必要です。この属性はそのセッションに必要なユーザー名を提供します。

poolUserLen (IN)

`poolUsername` の長さです。

poolPassword (IN)

ユーザー名 `poolUsername` のパスワードです。

poolPassLen (IN)

`poolPassword` の長さです。

mode (IN)

次のモードがサポートされています。

- `OCI_DEFAULT`
- `OCI_CPOOL_REINITIALIZE`

通常、`OCIConnectionPoolCreate()` は、`mode` を `OCI_DEFAULT` に設定してコールします。

プール属性を動的に変更する（たとえば、`connMin`、`connMax` および `connIncr` の各パラメータを変更する）場合は、`mode` を `OCI_CPOOL_REINITIALIZE` に設定して、`OCIConnectionPoolCreate()` をコールします。このコールの実行時、他のパラメータは無視されます。

コメント

OUT パラメータ `poolName` と `poolNameLen` には、データベース名とその長さの引数のかわりに、後続の `OCIServerAttach()` と `OCILogon2()` コールで使用する値が格納されます。

関連項目： A-21 ページ「[接続プール・ハンドル属性](#)」

関連関数

[OCIConnectionPoolDestroy\(\)](#)、[OCILogon2\(\)](#)、[OCIServerAttach\(\)](#)

OCIConnectionPoolDestroy()

用途

接続プールを破棄します。

構文

```
sword OCIConnectionPoolDestroy ( OCIPool      *poolhp,  
                                  OCIError     *errhp,  
                                  ub4          mode );
```

パラメータ

poolhp (IN)

作成したプールのプール・ハンドルです。

errhp (IN/OUT)

OCIErrorGet() に渡すことができるエラー・ハンドルです。

mode (IN)

現在、この関数では OCI_DEFAULT モードのみがサポートされています。

関連関数

[OCIConnectionPoolCreate\(\)](#)

OCIEnvCreate()

用途

OCI 関数が実行される環境を作成および初期化します。

構文

```

sword OCIEnvCreate ( OCIEnv      **envhpp,
                    ub4          mode,
                    CONST dvoid  *ctxp,
                    CONST dvoid  *(*malocfp)
                        (dvoid *ctxp,
                         size_t size),
                    CONST dvoid  *(*ralocfp)
                        (dvoid *ctxp,
                         dvoid *memptr,
                         size_t newsz),
                    CONST void   (*mfreefp)
                        (dvoid *ctxp,
                         dvoid *memptr))
                    size_t      xtramsz,
                    dvoid       **usrmemp );

```

パラメータ

envhpp (OUT)

環境ハンドルへのポインタです。環境ハンドルのエンコーディング設定は *mode* で指定します。この設定は、*envhpp* から導出された文ハンドルに継承されます。

mode (IN)

モードの初期化を指定します。次のモードが有効です。

- OCI_DEFAULT – デフォルト値。非 UTF-16 エンコーディングです。
- OCI_THREADED – スレッド環境を使用します。ユーザーに公開されていない内部データ構造体がマルチ・スレッドによって同時にアクセスされないように保護します。
- OCI_OBJECT – オブジェクト機能を使用します。
- OCI_UTF16 – 環境ハンドルと環境ハンドルから継承されたハンドルは、UTF-16 エンコーディングとみなされます。
- OCI_SHARED – 共有データ構造を利用します。
- OCI_EVENTS – バブリッシュ / サブスクライブ通知を利用します。
- OCI_NO_UCB – 動的コールバック・ルーチン *OCIEnvCallback* のコールを抑止します。デフォルトの動作では、環境の作成時に *OCIEnvCallback* のコールが許可されます。

関連項目： 9-35 ページ「動的なコールバック登録」

- **OCI_ENV_NO_MUTEX** — このモードでは **mutex** 化されません。環境ハンドル、または環境ハンドルから導出されたハンドルで行われたすべての **OCI** コールは、シリアル化する必要があります。
- **OCI_NEW_LENGTH_SEMANTICS** — キャラクタ・セットに関係なく、すべてのハンドルに対して一貫してバイト長セマンティクスを使用します。

ctxp (IN)

メモリー・コールバック・ルーチンのユーザー定義コンテキストを指定します。

malocfp (IN)

ユーザー定義のメモリー割当て関数を指定します。モードが **OCI_THREADED** の場合、このメモリー割当てルーチンは、スレッド・セーフにしてください。

ctxp (IN)

ユーザー定義のメモリー割当て関数の、コンテキスト・ポインタを指定します。

size (IN)

ユーザー定義のメモリー割当て関数によって割り当てられるメモリーのサイズを指定します。

ralocfp (IN)

ユーザー定義のメモリー再割当て関数を指定します。モードが **OCI_THREADED** の場合、このメモリー割当てルーチンは、スレッド・セーフにしてください。

ctxp (IN)

ユーザー定義のメモリー再割当て関数の、コンテキスト・ポインタを指定します。

memp (IN)

メモリー・ブロックのポインタです。

newsize (IN)

新しく割り当てられるメモリーのサイズを指定します。

mfreefp (IN)

ユーザー定義のメモリー解放関数を指定します。モードが **OCI_THREADED** の場合、このメモリー解放ルーチンは、スレッド・セーフにしてください。

ctxp (IN)

ユーザー定義のメモリー解放関数の、コンテキスト・ポインタを指定します。

memptr (IN)

解放されるメモリーのポインタです。

xtramemsz (IN)

環境の継続時間中に割り当てられるユーザー・メモリーの量を指定します。

usrmempp (OUT)

コールによりユーザーに割り当てられた、サイズ *xtramemsz* のユーザー・メモリーのポインタを戻します。

コメント

このコールにより、ユーザーによって指定されたモードを使用して、すべての OCI コールの環境が作成されます。

注意： このコールは、他の OCI コールより先に実行する必要があります。また、OCIInitialize() および OCIEnvInit() コールのかわりに使用してください。OCIInitialize() および OCIEnvInit() コールは、下位互換性を保つためにサポートされます。

このコールは、他の OCI コールの前に呼び出されます。したがって、このコールによってトップレベルでの環境ハンドルの Unicode サポートが設定されます。Unicode の設定は、*mode* の引数によって行われます。*mode* を OCI_UTF16 に設定します。

このコールでは、残りの OCI 関数で使用する環境ハンドルが戻されます。OCI には、それぞれ独自の環境モードを持つ複数の環境が存在する可能性があります。この関数では、どのモードで初期化を要求されても、プロセス・レベルの初期化を実行します。たとえば、環境を OCI_THREADED で初期化した場合は、OCI で使用されるすべてのライブラリもそのスレッド・モードで初期化されます。

OCI ライブラリを使用して DLL または共有ライブラリを記述している場合は、必ずこのコールを使用し、OCIInitialize() および OCIEnvInit() コールは使用しないでください。

関連項目： *xtramemsz* パラメータおよびユーザー・メモリー割当ての詳細は、2-14 ページの「ユーザー・メモリーの割当て」を参照してください。

関連関数

OCIHandleAlloc(), OCIHandleFree(), OCIEnvInit(), OCIEnvNlsCreate(), OCITerminate()

OCIEnvInit()

用途

OCI 環境ハンドルを割り当て、初期化します。

構文

```
sword OCIEnvInit ( OCIEnv      **envhpp,  
                   ub4         mode,  
                   size_t      xtramemsz,  
                   dvoid       **usrmempp );
```

パラメータ

envhpp (OUT)

環境へのハンドルのポインタです。

mode (IN)

環境モードの初期化を指定します。次のモードが有効です。

- OCI_DEFAULT
- OCI_NO_MUTEX
- OCI_ENV_NO_UCB

OCI_DEFAULT モードの場合、OCI ライブラリでハンドルが常に **mutex** 化されます。

OCI_NO_MUTEX モードの場合、この環境では **mutex** 化されません。

OCI_NO_MUTEX モードでは、環境ハンドルまたは環境ハンドルから導出されたハンドルで行われたすべての OCI コールは、シリアル化する必要があります。シリアル化するには、独自の **mutex** 化を行うか、または環境ハンドル上で操作中のスレッドを 1 つのみにします。

OCI_ENV_NO_UCB モードは、環境の初期化時に動的コールバック・ルーチン *OCIEnvCallback* のコールを抑止するために使用します。デフォルトでは、このコールは抑止されません。

関連項目： 9-35 ページ [「動的なコールバック登録」](#)

xtramemsz (IN)

環境の継続時間中に割り当てられるユーザー・メモリーの量を指定します。

usrmempp (OUT)

環境の継続時間中コールによってユーザー用に割り当てられた、*xtramemsz* サイズのユーザー・メモリーのポインタを戻します。

コメント

注意： `OCIInitialize()` および `OCIEnvInit()` コールは使用せずに、[OCIEnvCreate\(\)](#) を使用してください。`OCIInitialize()` および `OCIEnvInit()` コールは、下位互換性を保つためにサポートされます。

このコールは、OCI 環境ハンドルの割当ておよび初期化を行います。すでに初期化済みのハンドルには何も行いません。`OCI_ERROR` または `OCI_SUCCESS_WITH_INFO` が戻った場合は、この環境ハンドルを使用して Oracle 固有のエラーおよび診断を取得できます。

これはローカルに処理され、サーバー・ラウンドトリップはありません。

環境ハンドルは、`OCIHandleFree()` を使用して解放できます。

関連項目： `xtramemsz` パラメータおよびユーザー・メモリー割当ての詳細は、2-14 ページの「[ユーザー・メモリーの割当て](#)」を参照してください。

関連関数

[OCIHandleAlloc\(\)](#)、[OCIHandleFree\(\)](#)、[OCIEnvCreate\(\)](#)、[OCITerminate\(\)](#)

OCIEnvNlsCreate()

用途

OCI 関数が実行される環境ハンドルを作成および初期化します。これは、OCIEnvCreate() 関数を拡張した関数です。

構文

```

sword OCIEnvNlsCreate ( OCIEnv      **envhpp,
                        ub4          mode,
                        dvoid        *ctxp,
                        dvoid        *(*malocfp)
                                (dvoid *ctxp,
                                 size_t size),
                        dvoid        *(*ralocfp)
                                (dvoid *ctxp,
                                 dvoid *memptr,
                                 size_t newsize),
                        void          (*mfreefp)
                                (dvoid *ctxp,
                                 dvoid *memptr)
                        size_t       xtramemsz,
                        dvoid        **usrmempp
                        ub2          charset,
                        ub2          ncharset );

```

パラメータ

envhpp (OUT)

環境ハンドルへのポインタです。環境ハンドルのエンコーディング設定は *mode* で指定します。この設定は、*envhpp* から導出された文ハンドルに継承されます。

mode (IN)

モードの初期化を指定します。次のモードが有効です。

- OCI_DEFAULT — デフォルト値。非 UTF-16 エンコーディングです。
- OCI_THREADED — スレッド環境を使用します。ユーザーに公開されていない内部データ構造体がマルチ・スレッドによって同時にアクセスされないように保護します。
- OCI_OBJECT — オブジェクト機能を使用します。
- OCI_SHARED — 共有データ構造を利用します。
- OCI_EVENTS — パブリッシュ / サブスクライブ通知を利用します。
- OCI_NO_UCB — 動的コールバック・ルーチン *OCIEnvCallback* のコールを抑止します。デフォルトの動作では、環境の作成時に *OCIEnvCallback* のコールが許可されます。

関連項目： 9-35 ページ [「動的なコールバック登録」](#)

- **OCI_ENV_NO_MUTEX** — このモードでは **mutex** 化されません。環境ハンドル、または環境ハンドルから導出されたハンドルで行われたすべての OCI コールは、シリアル化する必要があります。

ctxp (IN)

メモリー・コールバック・ルーチンのユーザー定義コンテキストを指定します。

malocfp (IN)

ユーザー定義のメモリー割当て関数を指定します。*mode* が **OCI_THREADED** の場合、このメモリー割当てルーチンは、スレッド・セーフにしてください。

ctxp (IN)

ユーザー定義のメモリー割当て関数の、コンテキスト・ポインタを指定します。

size (IN)

ユーザー定義のメモリー割当て関数によって割り当てられるメモリーのサイズを指定します。

ralocfp (IN)

ユーザー定義のメモリー再割当て関数を指定します。モードが **OCI_THREADED** の場合、このメモリー割当てルーチンは、スレッド・セーフにしてください。

ctxp (IN)

ユーザー定義のメモリー再割当て関数の、コンテキスト・ポインタを指定します。

memp (IN)

メモリー・ブロックのポインタです。

newsize (IN)

新しく割り当てられるメモリーのサイズを指定します。

mfrefp (IN)

ユーザー定義のメモリー解放関数を指定します。モードが **OCI_THREADED** の場合、このメモリー解放ルーチンは、スレッド・セーフにしてください。

ctxp (IN)

ユーザー定義のメモリー解放関数の、コンテキスト・ポインタを指定します。

memptr (IN)

解放されるメモリーのポインタです。

xtramemsz (IN)

環境の継続時間中に割り当てられるユーザー・メモリーの量を指定します。

usrmempp (OUT)

コールによりユーザーに割り当てられた、サイズ *xtramemsz* のユーザー・メモリーのポインタを返します。

charset (IN)

現行の環境ハンドルに対するクライアント側キャラクタ・セットです。0（ゼロ）の場合は、NLS_LANG 設定が使用されます。OCI_UTF16ID は有効な設定で、メタデータおよび CHAR データで使用されます。

ncharset (IN)

現行の環境ハンドルに対するクライアント側各国語キャラクタ・セットです。0（ゼロ）の場合は、NLS_NCHAR 設定が使用されます。OCI_UTF16ID は有効な設定で、NCHAR データで使用されます。

戻り値

OCI_SUCCESS — 環境ハンドルは正常に作成されました。

OCI_ERROR — エラーが発生しました。

コメント

このコールにより、ユーザーによって指定されたモードを使用して、すべての OCI コールの環境が作成されます。この関数では、OCI_UTF16 モードはサポートされていません。OCIEnvNlsCreate() で OCI_UTF16ID を *charset* および *ncharset* として渡すと、OCIEnvCreate() で OCI_UTF16 および OCI_NEW_LENGTH_SEMANTICS モードを設定するのと同等になります。

OCIEnvNlsCreate() を使用して環境ハンドルを作成すると、バインド・ハンドルおよび定義ハンドルの実際の長さとは戻される長さは、常にバイト単位になります。これは、次のコールに適用されます。

- `OCIBindByName()`
- `OCIBindByPos()`
- `OCIBindDynamic()`
- `OCIDefineByPos()`
- `OCIDefineDynamic()`

環境の作成時に、この関数を使用して *charset* ID および *ncharset* ID を設定できます。これは、OCIEnvCreate() 関数を拡張した関数です。したがって、この関数では OCI_UTF16 モードはサポートされていませんが、OCI_UTF16ID を *charset* および *ncharset* として設定することによってこの機能を実現できます。

この関数は、クライアント側データベースおよび各国語キャラクタ・セットとして *charset* および *ncharset* を 0（ゼロ）以外の値に設定し、NLS_LANG および NLS_NCHAR で指定した値を置換します。*charset* および *ncharset* が 0（ゼロ）の場合は、OCIEnvCreate() の動作と完全に同じになります。*charset* は、暗黙的なフォーム属

性を持つメタデータおよびデータのエンコーディングを制御し、`ncharset` は、`SQLCS_NCHAR` フォーム属性を持つデータのエンコーディングを制御します。

`OCI_UTF16ID` は、`OCIEnvNlsCreate()` では設定できますが、`NLS_LANG` または `NLS_NCHAR` では設定できません。`NLS_LANG` および `NLS_NCHAR` のキャラクタ・セット ID にアクセスするには、`OCINlsEnvironmentVariableGet()` を使用します。

このコールでは、残りの OCI 関数で使用する環境ハンドルが戻されます。OCI には、それぞれ独自の環境モードを持つ複数の環境が存在する可能性があります。この関数では、どのモードで初期化を要求されても、プロセス・レベルの初期化を実行します。たとえば、環境を `OCI_THREADED` で初期化した場合は、OCI で使用されるすべてのライブラリもそのスレッド・モードで初期化されます。

OCI ライブラリを使用して DLL または共有ライブラリを記述している場合は、必ずこのコールを使用し、`OCIInitialize()` および `OCIEnvInit()` コールは使用しないでください。

関連項目： `xtramemsize` パラメータおよびユーザー・メモリー割当ての詳細は、2-14 ページの「[ユーザー・メモリーの割当て](#)」を参照してください。

関連関数

`OCIHandleAlloc()`、`OCIHandleFree()`、`OCITerminate()`、
`OCINlsEnvironmentVariableGet()`

OCIInitialize()

用途

OCI プロセス環境を初期化します。

構文

```
sword OCIInitialize ( ub4          mode,
                     CONST dvoid *ctxp,
                     CONST dvoid *(*malocfp)
                        (/* dvoid *ctxp,
                          size_t size _*/),
                     CONST dvoid *(*ralocfp)
                        (/* _ dvoid *ctxp,
                          dvoid *memptr,
                          size_t newsize _*/),
                     CONST void (*mfreefp)
                        (/* _ dvoid *ctxp,
                          dvoid *memptr _*/));
```

パラメータ

mode (IN)

モードの初期化を指定します。次のモードが有効です。

- OCI_DEFAULT — デフォルト・モード。
- OCI_THREADED — スレッド環境。このモードでは、ユーザーに公開されていない内部データ構造がマルチ・スレッドによって同時にアクセスされないように保護します。
- OCI_OBJECT — オブジェクト機能を使用します。
- OCI_SHARED — 共有データ構造を利用します。
- OCI_EVENTS — パブリッシュ / サブスクライブ通知を利用します。

ctxp (IN)

メモリー・コールバック・ルーチン用のユーザー定義コンテキストです。

malocfp (IN)

ユーザー定義のメモリー割当て関数です。*mode* が OCI_THREADED の場合、このメモリー割当てルーチンは、スレッド・セーフにしてください。

ctxp (IN/OUT)

ユーザー定義のメモリー割当て関数のためのコンテキスト・ポインタです。

size (IN)

ユーザー定義のメモリー割当て関数によって割り当てられるメモリーのサイズです。

ralocfp (IN)

ユーザー定義のメモリー再割当て関数です。mode が OCI_THREADED の場合、このメモリー割当てルーチンは、スレッド・セーフにしてください。

ctxp (IN/OUT)

ユーザー定義のメモリー再割当て関数のためのコンテキスト・ポインタです。

memptr (IN/OUT)

メモリー・ブロックのポインタです。

newsize (IN)

新しく割り当てられるメモリーのサイズです。

mfreefp (IN)

ユーザー定義のメモリー解放関数です。mode が OCI_THREADED の場合、このメモリー解放ルーチンは、スレッド・セーフにしてください。

ctxp (IN/OUT)

ユーザー定義のメモリー解放関数のためのコンテキスト・ポインタです。

memptr (IN/OUT)

解放されるメモリーのポインタです。

コメント

注意： OCIInitialize() および OCIEnvInit() コールは使用せずに、OCIEnvCreate() を使用してください。OCIInitialize() および OCIEnvInit() コールは、下位互換性を保つためにサポートされます。

このコールは、OCI プロセス環境を初期化します。OCIInitialize() は、他の OCI をコールする前にコールする必要があります。

この関数を使用すると、アプリケーションでコールバックを使用して固有のメモリー管理関数を定義できます。アプリケーションでこのようなメモリー管理関数（つまりメモリー割当て、メモリー再割当ておよびメモリー解放）がすでに定義されている場合は、この関数のコールバック・パラメータを使用して登録する必要があります。

これらのメモリー・コールバックはオプションです。アプリケーションで、この関数のメモリー・コールバックに NULL 値を渡すと、デフォルトのプロセス・メモリー割当てメカニズムが使用されます。

共有データ構造モード

SQL 文が処理されると、基礎となる特定のデータが文に関連付けられます。このデータには、問合せの定義情報および記述情報とともに、文のテキスト・データおよびバインド・データに関する情報が格納されています。文を何度実行しても、別のユーザーが実行しても、このデータは変わりません。

OCI アプリケーションが OCI_SHARED モードで初期化された場合は、複数の文ハンドル間で共通の文データが共有されるため、アプリケーションのメモリーを節約できます。このメモリーの節約は、複数の文ハンドルを作成するアプリケーションに特に有効です。このようなアプリケーションでは、同じ SQL 文が、同一または複数の接続で、複数のユーザーのセッション上で実行されるためです。

関連項目：

- 詳細は、2-22 ページの「[共有データ・モード](#)」を参照してください。
- OCI を使用したマルチスレッド・アプリケーションのプログラミングの詳細は、9-2 ページの「[スレッド・セーフティ](#)」を参照してください。
- オブジェクトを使用した OCI プログラミングの詳細は、[第 10 章「OCI オブジェクト・リレーショナル・プログラミング」](#)を参照してください。

例

次の文は、ユーザー定義メモリー関数がない場合に、スレッド・モードおよびオブジェクト・モードで OCIInitialize() をコールする方法の例です。

```
OCIInitialize((ub4) OCI_THREADED | OCI_OBJECT, (dvoid *)0,  
              (dvoid * (*)()) 0, (dvoid * (*)()) 0, (void (*)()) 0 );
```

関連関数

[OCIHandleAlloc\(\)](#)、[OCIHandleFree\(\)](#)、[OCIEnvCreate\(\)](#)、[OCIEnvInit\(\)](#)、[OCITerminate\(\)](#)

OCILogoff()

用途

この関数は、`OCILogon2()` または `OCILogon()` を使用して取得したセッションを解放するために使用します。

構文

```
sword OCILogoff ( OCISvcCtx      *svchp  
                  OCIError      *errhp );
```

パラメータ

svchp (IN)

`OCILogon()` または `OCILogon2()` のコールに使用されたサービス・コンテキスト・ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために `OCIErrorGet()` に渡すエラー・ハンドルです。

コメント

この関数は、`OCILogon2()` または `OCILogon()` を使用して取得したセッションを解放するために使用します。`OCILogon()` を使用した場合、この関数はその接続およびセッションを終了します。`OCILogon2()` を使用した場合、このコールの動作は、対応する `OCILogon2()` 関数がコールされたときの *mode* によって決まります。デフォルトでは、セッション / 接続をクローズします。接続プーリングの場合、この関数は、セッションをクローズして接続をプールに戻します。セッション・プーリングの場合、この関数は、セッションと接続のペアをプールに戻します。

関連項目： アプリケーションでのログオンおよびログオフの詳細は、2-26 ページの「[アプリケーションの初期化、接続およびセッション作成](#)」を参照してください。

関連関数

`OCILogon()`、`OCILogon2()`

OCILogon()

用途

この関数は、ログイン・セッションを作成するときに使用します。

構文

```
sword OCILogon ( OCIEnv          *envhp,  
                  OCIError       *errhp,  
                  OCISvcCtx       **svchp,  
                  CONST OraText   *username,  
                  ub4             uname_len,  
                  CONST OraText   *password,  
                  ub4             passwd_len,  
                  CONST OraText   *dbname,  
                  ub4             dbname_len );
```

パラメータ

envhp (IN)

OCI 環境ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

svchp (IN/OUT)

サービス・コンテキスト・ポインタです。

username (IN)

ユーザー名です。OCI_UTF16 環境モードでの UTF-16 エンコーディングであることが必要です。

uname_len (IN)

username の長さは、エンコーディングの有無に関係なく、バイト単位です。

password (IN)

ユーザーのパスワードです。OCI_UTF16 環境モードでの UTF-16 エンコーディングであることが必要です。

passwd_len (IN)

password の長さは、エンコーディングの有無に関係なく、バイト単位です。

dbname (IN)

接続先のデータベース名です。OCI_UTF16 環境モードでの UTF-16 エンコーディングであることが必要です。

dbname_len (IN)

dbname の長さは、エンコーディングの有無に関係なく、バイト単位です。

コメント

この関数は、アプリケーションのためのログイン・セッションを作成するときに使用します。

注意： TP モニター・アプリケーションなどの複雑なセッションを必要とするユーザーは、2-26 ページの「[アプリケーションの初期化、接続およびセッション作成](#)」を参照してください。

このコールは、渡されたサービス・コンテキスト・ハンドルの割当てを行います。このコールは、セッションに対応付けられたサーバーとユーザー・セッション・ハンドルの暗黙的な割当てでも実行します。これらのハンドルは、サービス・コンテキスト・ハンドルに対して [OCIAttrGet\(\)](#) をコールすることにより取り出せます。

関連関数

[OCILogoff\(\)](#)

OCILogon2()

用途

セッションを取得します。このセッションは、基礎となる新規の接続があるセッション、既存の接続プールから仮想接続によって開始されたセッション、または既存のセッション・プールからのセッションである場合があります。この関数がコールされるとき *mode* によって、動作が決まります。

構文

```
sword OCILogon2 ( OCIEnv          *envhp,
                  OCIError        *errhp,
                  OCISvcCtx        **svchp,
                  CONST OraText    *username,
                  ub4              uname_len,
                  CONST OraText    *password,
                  ub4              passwd_len,
                  CONST OraText    *dbname,
                  ub4              dbname_len );
                  ub4              mode );
```

パラメータ

envhp (IN)

OCI 環境ハンドルです。接続プーリングおよびセッション・プーリングの場合は、それぞれのプールが作成されたときの環境ハンドルに設定する必要があります。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

svchp (IN/OUT)

OCI サービス・コンテキスト・ポインタのアドレスです。サーバーおよびセッション・ハンドルが格納されます。

デフォルトでは、新しいセッションおよびサーバー・ハンドルが割り当てられ、接続およびセッションが開始し、サービス・コンテキストにこれらのハンドルが移入されます。

接続プーリングの場合、新しいセッション・ハンドルが割り当てられ、接続プールから仮想接続によってセッションが開始します。

セッション・プーリングの場合、サービス・コンテキストには、セッション・プールからセッション・ハンドルとサーバー・ハンドルの既存のペアが移入されます。

ユーザーは、サーバーの属性、およびサービス・コンテキスト・ポインタに関連付けられたユーザー / セッション・ハンドルは変更できないことに注意してください。変更を行うと、OCIAttrSet () コールでエラーが戻されます。

変更できるサービス・コンテキスト属性は、OCI_ATTR_STMTCACHESIZE のみです。

username (IN)

セッションを認証するために使用するユーザー名です。OCI_UTF16 環境モードでの UTF-16 エンコーディングである必要があります。

uname_len (IN)

username の長さは、エンコーディングの有無に関係なく、バイト単位です。

password (IN)

ユーザーのパスワードです。接続プーリングでこのパラメータが NULL の場合、OCILogon2() では、プロキシ・ユーザーのログインとみなされます。このような場合は、プール・ユーザーがプロキシ・ユーザーの認証に使用され、プロキシ接続が暗黙的に作成されます。OCI_UTF16 環境モードでの UTF-16 エンコーディングである必要があります。

passwd_len (IN)

password の長さは、エンコーディングの有無に関係なく、バイト単位です。

dbname (IN)

デフォルトでは、Oracle データベース・サーバーへの接続で使用する接続文字列を示します。

接続プーリングの場合は、仮想接続を取得してセッションを開始するための接続プールを示します。この値は、OCIConnectionPoolCreate() コールによって戻されます。

セッション・プーリングの場合は、セッションを取得するプールを示します。この値は、OCISessionPoolCreate() コールによって戻されます。

OCI_UTF16 環境モードでの UTF-16 エンコーディングである必要があります。

dbname_len (IN)

dbname の長さです。セッション・プーリングおよび接続プーリングの場合、この値は、それぞれ OCISessionPoolCreate() コールまたは OCIConnectionPoolCreate() コールによって戻されます。

mode (IN)

指定できる値は、次のとおりです。

- OCI_DEFAULT
- OCI_LOGON2_CPOOL
- OCI_LOGON2_SPOOL
- OCI_LOGON2_STMTCACHE
- OCI_LOGON2_PROXY

デフォルト（プーリング以外の場合）では、次のモードが有効です。

OCI_DEFAULT – OCILogon() のコールと等価です。

OCI_LOGON2_STMTCACHE – 文キャッシュを使用可能にします。

接続プーリングの場合は、次のモードが有効です。

OCI_LOGON2_CPOOL または OCI_CPOOL – 接続プーリングを使用する場合に設定する必要があります。

OCI_LOGON2_STMTCACHE – 文キャッシュを使用可能にします。

接続プーリングでプロキシ認証を使用する場合、パスワードは NULL に設定する必要があります。ユーザーには、OCILogon2() コールで提供されたユーザー名によって認証されるセッションが、OCIConnectionPoolCreate() コールで提供されたプロキシ資格証明を通じて与えられます。

セッション・プーリングの場合は、次のモードが有効です。

OCI_LOGON2_SPOOL – セッション・プーリングを使用する場合に設定する必要があります。

OCI_LOGON2_STMTCACHE – 文キャッシュを使用可能にします。

OCI_LOGON2_PROXY – プロキシ認証を使用します。ユーザーには、OCILogon2() コールで提供されたユーザー名によって認証されるセッションが、OCISessionPoolCreate() コールで提供されたプロキシ資格証明を通じて与えられます。

コメント

なし

関連関数

[OCILogon\(\)](#)、[OCILogoff\(\)](#)、[OCISessionGet\(\)](#)、[OCISessionRelease\(\)](#)

OCIServerAttach()

用途

OCI オペレーションの対象となるデータ・ソースへのアクセス・パスを作成します。

構文

```
sword OCIServerAttach ( OCIServer      *srvhp,  
                        OCIError       *errhp,  
                        CONST text     *dblink,  
                        sb4            dblink_len,  
                        ub4            mode );
```

パラメータ

srvhp (IN/OUT)

このコールにより初期化される未初期化サーバー・ハンドルです。初期化済みのサーバー・ハンドルを渡すとエラーが発生します。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

dblink (IN)

使用するデータベース・サーバーを指定します。このパラメータは、接続文字列またはサービス・ポイントを指定する文字列を指示します。接続文字列が NULL の場合、このコールは、デフォルト・ホストに接続します。文字列自体が UTF-16 であるかないかは、アプリケーションの環境ハンドルのモードまたは設定によって決まります。dblink の長さは、dblink_len で指定します。dblink ポインタは、戻り時にコール元によって解放されます。

mode=OCI_CPOOL 時の接続先プールの名前です。この名前は、OCIConnectionPoolCreate () で作成された接続プールの poolName パラメータと同じ名前であることが必要です。OCI_UTF16 環境モードでの UTF-16 エンコーディングであることが必要です。

dblink_len (IN)

dblink が指し示す文字列の長さです。接続文字列名または別名を有効にするには、dblink_len は 0 (ゼロ) 以外にしてください。値はバイト単位です。

mode=OCI_CPOOL 時の poolName の長さは、エンコーディングの有無に関係なく、バイト単位です。

mode (IN)

様々な操作モードを指定します。次のモードが有効です。

- OCI_DEFAULT – エンコーディングについて、この値では、サーバー・ハンドルで環境ハンドルの設定が使用されることを示しています。
- OCI_CPOOL – 接続プーリングを使用します。

連結済みのサーバー・ハンドルは、任意の接続セッション・ハンドルに設定されている可能性があるため、ここでの *mode* 値は、セッション・ハンドルに対しては無効です。

コメント

このコールは、OCI アプリケーションと特定のサーバー間の対応付けを作成するときに使用します。

このコールでは、接続プーリングが有効になった時点で、OCIConnectionPoolCreate() がすでにコールされ、poolName が指定されていると想定しています。

このコールは、サーバー・コンテキスト・ハンドルを初期化しますが、ハンドルは OCIHandleAlloc() のコールを使用して事前に割り当てる必要があります。このコールによって初期化されたサーバー・コンテキスト・ハンドルは、OCIAttrSet() のコールを介してサービス・コンテキストに対応付けられます。対応付けを行うと、そのサーバーに対して OCI オペレーションを実行できます。

複数のサーバーに対してアプリケーションを実行している場合、複数のサーバー・コンテキスト・ハンドルを維持できます。OCI オペレーションは、現在サービス・コンテキストに対応付けられているサーバー・コンテキストに対して実行されます。

OCIServerAttach() が正常に完了すると、Oracle シャドウ・プロセスが開始されます。Oracle シャドウ・プロセスをクリーン・アップするには、OCISessionEnd() および OCIServerDetach() をコールする必要があります。コールしないと、シャドウ・プロセスが蓄積され、UNIX システムのプロセス数が足りなくなります。データベースが再起動したときにプロセス数が不足している場合は、データベースが起動しないことがあります。

例

次のコードは OCIServerAttach() の使用方法の例です。このコード・セグメントでは、サーバー・ハンドルを割り当て、連結コールを行い、サービス・コンテキスト・ハンドルを割り当て、最後にそれにサーバー・コンテキストを設定します。

```
OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4)
    OCI_HTYPE_SERVER, 0, (dvoid **) &tmp);
OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);
OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4)
    OCI_HTYPE_SVCCTX, 0, (dvoid **) &tmp);
/* set attribute server context in the service context */
OCIAttrSet( (dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *) srvhp,
    (ub4) 0, (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);
```

関連関数

[OCIServerDetach\(\)](#)

OCIServerDetach()

用途

OCI オペレーションの対象となるデータ・ソースへのアクセスを削除します。

構文

```
sword OCIServerDetach ( OCIServer   *srvhp,  
                        OCIError    *errhp,  
                        ub4          mode );
```

パラメータ

srvhp (IN)

未初期化状態にリセットされる初期化済みサーバー・コンテキストへのハンドルです。ハンドルの割当ては解除されません。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

mode (IN)

様々な操作モードを指定します。有効なモードはデフォルト・モード用の OCI_DEFAULT のみです。

コメント

このコールは、OCIServerAttach () のコールによって確立された OCI オペレーションの対象となるデータ・ソースへのアクセスを削除します。

関連関数

[OCIServerAttach\(\)](#)

OCISessionBegin()

用途

ユーザー・セッションを作成し、指定のサーバーに対してユーザー・セッションを開始します。

構文

```
sword OCISessionBegin ( OCISvcCtx      *svchp,
                        OCIError       *errhp,
                        OCISession     *usrhp,
                        ub4             credt,
                        ub4             mode );
```

パラメータ

svchp (IN)

サービス・コンテキストへのハンドルです。svchp には、有効なサーバー・ハンドルを設定する必要があります。

errhp (IN)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

usrhp (IN/OUT)

このコールによって初期化されるユーザー・セッション・コンテキストへのハンドルです。

credt (IN)

ユーザー・セッションの作成に使用する資格証明のタイプを指定します。credt の有効な値は次のとおりです。

- OCI_CRED_RDBMS — 資格証明としてデータベースのユーザー名とパスワードを使用して認証します。このコールの前に、ユーザー・セッション・コンテキストに OCI_ATTR_USERNAME と OCI_ATTR_PASSWORD 属性を設定しておく必要があります。
- OCI_CRED_EXT — 外部資格証明を使用して認証します。ユーザー名とパスワードは提供されません。

mode (IN)

様々な操作モードを指定します。次のモードが有効です。

- OCI_DEFAULT — このモードでは、戻されるユーザー・セッション・コンテキストは、svchp に指定された、同じサーバー・コンテキストとともにのみ設定できます。エンコーディングについては、サーバー・ハンドルでは、環境ハンドルの設定が使用されます。

- **OCI_MIGRATE** — このモードでは、新しいユーザー・セッション・コンテキストは、別のサーバー・ハンドルとともにサービス・ハンドルに設定されます。このモードは、ユーザー・セッション・コンテキストを確立します。移行可能なセッションを作成するには、サービス・ハンドルを移行不可能なセッションであらかじめ設定しておく必要があります。この移行不可能なセッションは移行可能なセッションの作成者セッションになります。つまり、移行可能なセッションの親は、移行不可能なセッションにしてください。
- **OCI_SYSDBA** — このモードでは、ユーザーは、SYSDBA アクセス用に認証されます。
- **OCI_SYSOPER** — このモードでは、ユーザーは、SYSOPER アクセス用に認証されます。
- **OCI_PRELIM_AUTH** — このモードは、特定の管理タスクを認証するために **OCI_SYSDBA** または **OCI_SYSOPER** とともにのみ使用できます。

コメント

OCISessionBegin() コールは、サービス・コンテキスト・ハンドルに設定されたサーバー・セットに対して、ユーザーを認証するときに使用します。

注意： セッションを開始するときは戻されたエラーをすべてチェックしてください。たとえば、アカウントのパスワードの期限が切れた場合は、ORA-28001 エラーが戻されます。

リリース 8.1 以上では、サーバー・ハンドルに要求を行う前に、そのサーバー・ハンドルに対して **OCISessionBegin()** をコールする必要があります。 **OCISessionBegin()** は、サーバー・ハンドルによってサービス・コンテキスト内に指定されたユーザーによる Oracle サーバーへの接続に対する認証のみをサポートします。つまり、 **OCIserverAttach()** をコールしてサーバー・ハンドルを初期化した後、指定のサーバーに対してユーザーを認証するために **OCISessionBegin()** をコールする必要があります。

Unicode の使用時に、*mode* または環境ハンドルが適切に設定されている場合、セッション・ハンドル *usrhp* に設定されているユーザー名とパスワードは、すでに Unicode になっています。この関数をコールして、ユーザー名とパスワードでセッションを開始するには、 **OCIAttrSet()** をコールして、この 2 つの Unicode 文字列を対応するバイト長でセッション・ハンドルに設定しておくことが必要です。これは、 **OCIAttrSet()** では、 **dvoid** ポインタのみを取得するためです。その結果、 **OCISessionBegin()** によって文字列バッファが解釈されます。

指定のサーバー・ハンドルに対して、最初に **OCISessionBegin()** をコールした場合は、ユーザー・セッションは、移行可能モード (**OCI_MIGRATE**) では作成されません。

サーバー・ハンドルに対して `OCISessionBegin()` をコールした後、アプリケーションでは `OCISessionBegin()` を再度コールし、別のユーザー・セッション・ハンドルを、別の（または同じ）資格証明と別の（または同じ）操作モードを使用して初期化できます。アプリケーションでユーザーを `OCI_MIGRATE` モードで認証する場合、サービス・ハンドルは、移行不可能なユーザー・ハンドルにすでに関連付けられている必要があります。このユーザー・ハンドルのユーザー ID は、移行可能なユーザー・セッションの所有者 ID になります。移行可能なすべてのセッションは、移行不可能な親セッションを持つ必要があります。

`OCI_MIGRATE` を指定しない場合、ユーザー・セッション・コンテキストは、`svchp` に設定されたサーバー・ハンドルと同じサーバー・ハンドルでしか使用できません。`OCI_MIGRATE` モードを指定した場合、ユーザー認証は、異なるサーバー・ハンドルで設定されます。ただし、ユーザー・セッション・コンテキストは、同じデータベース・インスタンスを解決するサーバー・ハンドルでのみ使用できます。セキュリティ・チェックは、セッションの切替え中に行われます。セッションは、ユーザー ID が作成者のユーザー ID または固有のユーザー ID と同一であるプロセスに現在接続している移行不可能なセッションがある場合にかぎり、そのプロセスへ移行できます。

`OCI_SYSDBA`、`OCI_SYSOPER` および `OCI_PRELIM_AUTH` は、主ユーザー・セッション・コンテキストでのみ使用できます。

`OCISessionBegin()` のコール用に資格証明を与えるために、2 通りの方法のうち 1 つがサポートされています。最初の方法は、`OCISessionBegin()` に渡されるユーザー・セッション・ハンドル内にデータベース認証用の有効なユーザー名とパスワードのペアを用意することです。この方法では、`OCIAttrSet()` を使用して、ユーザー・セッション・ハンドルに対して `OCI_ATTR_USERNAME` および `OCI_ATTR_PASSWORD` 属性を設定します。これにより、`OCI_CRED_RDBMS` を指定して `OCISessionBegin()` がコールされます。

注意： ユーザー名とパスワード属性は、`OCISessionEnd()` を使用してユーザー・セッション・ハンドルを終了しても変更されず、以降の `OCISessionBegin()` のコールで再使用できます。再使用しない場合は、次の `OCISessionBegin()` コールの前に新しい値を設定しなおす必要があります。

もう 1 つの資格証明の方法は、外部資格証明です。この方法では、`OCISessionBegin()` をコールする前に、ユーザー・セッション・ハンドルについて属性を設定する必要はありません。資格証明型は `OCI_CRED_EXT` です。これは、Oracle7 の `connect/` 構文と等価です。すでに `OCI_ATTR_USERNAME` および `OCI_ATTR_PASSWORD` に値が設定してある場合、`OCI_CRED_EXT` を使用すると、これらの値は無視されます。

資格証明を設定するもう 1 つの方法では、`OCI_MIGSESSION` 属性とともに、すでに認証されているユーザーのセッション ID を使用します。この ID は、`OCIAttrGet()` コールを使用して認証されたユーザーのセッション・ハンドルから抽出できます。

例

次のコードは `OCISessionBegin()` の使用方法の例です。この例では、ユーザー・セッション・ハンドルを割り当て、ユーザー名とパスワードを設定し、`OCISessionBegin()` をコールして、最後にサービス・コンテキスト内にユーザー・セッションを設定します。

```
/* allocate a user session handle */
OCIHandleAlloc((dvoid *)envhp, (dvoid **)&usrhp, (ub4)
    OCI_HTYPE_SESSION, (size_t) 0, (dvoid **) 0);
OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION, (dvoid *)"jessica",
    (ub4)strlen("jessica"), OCI_ATTR_USERNAME, errhp);
OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION, (dvoid *)"doogie",
    (ub4)strlen("doogie"), OCI_ATTR_PASSWORD, errhp);
checkerr(errhp, OCISessionBegin (svchp, errhp, usrhp, OCI_CRED_RDBMS,
    OCI_DEFAULT));
OCIAttrSet((dvoid *)svchp, (ub4)OCI_HTYPE_SVCCTX, (dvoid *)usrhp,
    (ub4)0, OCI_ATTR_SESSION, errhp);
```

関連関数

`OCISessionEnd()`

OCISessionEnd()

用途

OCISessionBegin() によって作成されたユーザー・セッション・コンテキストを終了します。

構文

```
sword OCISessionEnd ( OCISvcCtx      *svchp,  
                      OCIError      *errhp,  
                      OCISession    *usrhp,  
                      ub4            mode );
```

パラメータ

svchp (IN/OUT)

サービス・コンテキスト・ハンドルです。*svchp* には、有効なサーバー・ハンドルおよびユーザー・セッション・ハンドルを対応付ける必要があります。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet() に渡すエラー・ハンドルです。

usrhp (IN)

このユーザーの認証を解除します。このパラメータが NULL として渡された場合は、サービス・コンテキスト・ハンドル内のユーザーの認証が解除されます。

mode (IN)

有効なモードは OCI_DEFAULT のみです。

コメント

サービス・コンテキストに対応付けられているユーザー・セキュリティ・コンテキストがこのコールによって無効化されます。ユーザー・セッション・コンテキスト用の記憶域は解放されません。サービス・コンテキストによって指定されたトランザクションが暗黙的にコミットされます。トランザクション・ハンドルは、明示的に割り当てられ、使用されていない場合は、解放されます。このユーザーのためにサーバー上に割り当てられていたリソースが解放されます。ユーザー・セッション・ハンドルは、OCISessionBegin() の次のコールで再使用できます。

関連関数

[OCISessionBegin\(\)](#)

OCISessionGet()

用途

セッションを取得します。このセッションは、基礎となる新規の接続があるセッション、既存の接続プールから仮想接続によって開始されたセッション、または既存のセッション・プールからのセッションである場合があります。この関数がコールされるときの *mode* によって、動作が決まります。

構文

```
sword OCISessionGet ( OCIEnv          *envhp,
                     OCIError        *errhp,
                     OCISvcCtx       **svchp,
                     OCIAuthInfo     *authInfo,
                     OraText         *dbName,
                     ub4              dbName_len,
                     CONST OraText   *tagInfo,
                     ub4              tagInfo_len,
                     OraText         **retTagInfo,
                     ub4              *retTagInfo_len,
                     boolean          *found,
                     ub4              mode );
```

パラメータ

envhp (IN/OUT)

OCI 環境ハンドルです。接続プーリングおよびセッション・プーリングの場合は、それぞれのプールが作成されたときの環境ハンドルに設定する必要があります。

errhp (IN/OUT)

OCI エラー・ハンドルです。

svchp (OUT)

OCI サービス・コンテキスト・ポインタのアドレスです。サーバーおよびセッション・ハンドルが格納されます。

デフォルトでは、新しいセッションおよびサーバー・ハンドルが割り当てられ、接続およびセッションが開始し、サービス・コンテキストにこれらのハンドルが移入されます。

接続プーリングの場合、新しいセッション・ハンドルが割り当てられ、接続プールから仮想接続によってセッションが開始します。

セッション・プーリングの場合、サービス・コンテキストには、セッション・プールからセッション・ハンドルとサーバー・ハンドルの既存のペアが移入されます。

サーバーの属性、およびサービス・コンテキスト・ポインタに関連付けられたユーザー・ハンドルとセッション・ハンドルは変更しないでください。変更を行うと、OCIAttrSet() コールでエラーが戻されます。

変更できるサービス・コンテキスト属性は、OCI_ATTR_STMTCACHESIZE のみです。

authInfo (IN)

セッションの取得時に使用する認証情報ハンドルです。

接続プーリングの場合、デフォルトでは、このハンドルはセッション・ハンドルのすべての属性を使用できます。

セッション・プーリングの場合、認証情報ハンドルは、セッション・プールのモードが OCI_SPC_HOMOGENEOUS 以外の場合のみ使用されます。この場合、このハンドルで利用できる属性は次のとおりです。

OCI_ATTR_USERNAME

OCI_ATTR_PASSWORD

OCI_ATTR_INITIAL_CLIENT_ROLES

詳細は、「ユーザー・セッション・ハンドル属性」を参照してください。

関連項目： [A-17 ページ「ユーザー・セッション・ハンドル属性」](#)

dbName (IN)

デフォルトでは、Oracle データベース・サーバーへの接続で使用する接続文字列を示します。

接続プーリングの場合は、仮想接続を取得してセッションを開始するための接続プールを示します。この値は、OCIConnectionPoolCreate() コールによって戻されます。

セッション・プーリングの場合は、セッションを取得するプールを示します。この値は、OCISessionPoolCreate() コールによって戻されます。

dbname_len (IN)

dbName の長さです。セッション・プーリングおよび接続プーリングの場合、この値は、それぞれ OCISessionPoolCreate() または OCIConnectionPoolCreate() のコールによって戻されます。

tagInfo (IN)

このパラメータは、セッション・プーリングでのみ使用します。

これは、ユーザーが使用するセッションのタイプを示します。デフォルト・セッションを使用する場合、ユーザーはこのパラメータを NULL に設定する必要があります。このパラメータの使用の詳細は、「コメント」を参照してください。

tagInfo_len (IN)

バイトで示した *tagInfo* の長さです。セッション・プーリングでのみ使用します。

retTagInfo (OUT)

このパラメータは、セッション・プーリングでのみ使用します。これは、ユーザーに戻されるセッションのタイプを示します。このパラメータの使用の詳細は、「コメント」を参照してください。

retTagInfo_len (OUT)

バイトで示した *retTagInfo* の長さです。セッション・プーリングでのみ使用します。

found (OUT)

このパラメータは、セッション・プーリングでのみ使用します。ユーザーが要求したセッションのタイプが戻された場合（つまり、*tagInfo* と *retTagInfo* の値が同じ場合）、*found* は TRUE に設定され、それ以外の場合、*found* は FALSE に設定されます。

mode (IN)

次のモードが有効です。

- OCI_DEFAULT
- OCI_SESSGET_CPOOL
- OCI_SESSGET_SPOOL
- OCI_SESSGET_CREDPROXY
- OCI_SESSGET_CREDEXT
- OCI_SESSGET_SPOOL_MATCHANY
- OCI_SESSGET_STMTCACHE

デフォルト（プーリング以外の場合）では、次のモードが有効です。

OCI_SESSGET_STMTCACHE — セッションでの文キャッシュを使用可能にします。

OCI_SESSGET_CREDEXT — 外部資格証明を使用して認証されたセッションを戻します。

接続プーリングの場合は、次のモードが有効です。

OCI_SESSGET_CPOOL — 接続プーリングを使用するには、このモードに設定する必要があります。

OCI_SESSGET_STMTCACHE — セッションでの文キャッシュを使用可能にします。

OCI_SESSGET_CREDPROXY — プロキシ・セッションを戻します。ユーザーには、OCI_SessionGet() コールで提供されたユーザー名によって認証されるセッションが、OCI_ConnectionPoolCreate() コールで提供されたプロキシ資格証明を通じて与えられます。

OCI_SESSGET_CREDEXT — 外部資格証明を使用して認証されたセッションを戻します。

セッション・プーリングの場合は、次のモードが有効です。

OCI_SESSGET_SPOOL — セッション・プーリングを使用するには、このモードに設定する必要があります。

OCI_SESSGET_CREDPROXY — ユーザーには、`OCISessionGet()` コールで提供されたユーザー名によって認証されるセッションが、`OCISessionPoolCreate()` コールで提供されたプロキシ資格証明を通じて与えられます。

OCI_SESSGET_SPOOL_MATCHANY — タグ付けの動作を参照します。このモードに設定すると、要求されたタグと異なるタグを持つセッションを戻すことができます。次の「コメント」を参照してください。

コメント

ユーザーは、タグを使用して、プール内のセッションをカスタマイズできます。クライアントは、デフォルト・セッションまたはタグが付いていないセッションをプールから取得して、特定の属性（グローバル化セッション設定など）をセッションに設定し、`OCISessionRelease()` コールでそのセッションをプールに戻すと、セッションに適切なラベル（タグ）を付けることができます。

ユーザーまたは他のユーザーは、`OCISessionGet()` コールで同じタグを指定して、同じ属性を持つセッションを要求できます。

タグ 'A' を持つセッションをユーザーが要求し、一致するセッションが使用不可である場合は、適切に認証されたタグなしのセッション（NULL タグを持つセッション）が戻されます（そのセッションが使用可能な場合）。タグなしのセッションが使用不可で、**OCI_SESSGET_SPOOL_MATCHANY** が指定されている場合は、適切に認証されて異なるタグを持つセッションが戻されます。**OCI_SESSGET_SPOOL_MATCHANY** が設定されていない場合は、異なるタグを持つセッションが戻されることはありません。

関連関数

`OCISessionRelease()`、`OCISessionPoolCreate()`、`OCISessionPoolDestroy()`

OCISessionPoolCreate()

用途

セッション・プールを初期化します。データベースに対して、`sessMin` の数のセッションおよび接続を開始します。この関数をコールする前に、`OCIHandleAlloc()` をコールしてセッション・プール・ハンドルのメモリーを割り当ててください。

構文

```
sword OCISessionPoolCreate ( OCIEnv          *envhp,  
                             OCIError        *errhp,  
                             OCISPool        *spoolhp,  
                             OraText         **poolName,  
                             ub4             *poolNameLen,  
                             CONST OraText   *connStr,  
                             ub4             connStrLen,  
                             ub4             sessMin,  
                             ub4             sessMax,  
                             ub4             sessIncr,  
                             OraText         *userid,  
                             ub4             useridLen,  
                             OraText         *password,  
                             ub4             passwordLen,  
                             ub4             mode );
```

パラメータ

envhp (IN)

セッション・プールで作成する必要がある環境ハンドルへのポインタです。

errhp (IN/OUT)

`OCIErrorGet()` に渡すことができるエラー・ハンドルです。

spoolhp (IN/OUT)

初期化するセッション・プール・ハンドルへのポインタです。

poolName (OUT)

戻されるセッション・プールの名前です。これは、環境内にあるすべてのセッション・プール間で一意の名前です。この値を `OCISessionGet()` コールに渡す必要があります。

poolNameLen (OUT)

バイトで示した `poolName` の長さです。

connStr (IN)

接続先のデータベースの TNS 別名です。

connStrLen (IN)

バイトで示した connStr の長さです。

sessMin (IN)

セッション・プールの最小セッション数を指定します。

この数のセッションが OCISessionPoolCreate() で開始されます。以降は、必要なときのみセッションをオープンします。

この値は、mode が OCI_SPC_HOMOGENEOUS に設定されている場合に使用されます。それ以外の場合、この値は無視されます。

sessMax (IN)

セッション・プールでオープンできる最大セッション数を指定します。最大数に達すると、追加のセッションはオープンしません。有効な値は 1 以上です。

sessIncr (IN)

現行のセッション数が sessMax 未満の場合、アプリケーションでは、開始するセッションに次の増分を設定できます。有効な値は 0（ゼロ）以上です。

sessMin + sessIncr が sessMax を超えることはできません。

userid (IN)

セッションを開始するために使用するユーザー ID を指定します。

関連項目： このパラメータの詳細は、15-41 ページの「[認証に関する注意](#)」を参照してください。

useridLen (IN)

バイトで示したユーザー ID の長さです。

password (IN)

ユーザー ID に対応するパスワードです。

passwordLen (IN)

バイトで示したパスワードの長さです。

mode (IN)

次のモードがサポートされています。

- OCI_DEFAULT — 新しいセッション・プールを作成します。
- OCI_SPC_REINITIALIZE — セッション・プールの作成後、プール属性を動的に変更する (sessMin、sessMax および sessIncr パラメータを変更する) 場合は、mode を OCI_SPC_REINITIALIZE に設定して OCISessionPoolCreate() をコールします。mode が OCI_SPC_REINITIALIZE に設定されている場合、connStr、userid および password は無視されます。
- OCI_SPC_STMTCACHE — セッション・プール用に OCI の文キャッシュが作成されます。OCI の文キャッシュを使用可能にしてプールが作成されていない場合は、サーバー側の文キャッシュが自動的に使用されます。通常、クライアント側の文キャッシュを使用した方がパフォーマンスが向上することに注意してください。

関連項目： 9-28 ページ「[文キャッシュ](#)」

- OCI_SPC_HOMOGENEOUS — プール内のすべてのセッションは、OCISessionPoolCreate() に渡されるユーザー名とパスワードを使用して認証されます。この場合、OCISessionGet() に渡される認証ハンドル (authInfo パラメータ) は無視されます。sessMin および sessIncr の値は、この場合のみ使用されます。このモードでは、プロキシ・セッションは作成できません。

コメント

認証に関する注意

セッション・プールでは、データベースへの接続として、ダイレクト接続とプロキシ接続の 2 通りの接続があります。プロキシ接続を行う場合、ユーザーには *Connect through Proxy* 権限が必要です。

関連項目： プロキシ接続の詳細は、次を参照してください。

- 『Oracle9i SQL リファレンス』
- 『Oracle9i データベース概要』

セッション・プールの作成時に、userid および password を指定する場合と指定しない場合があります。これらの値が NULL の場合は、このプールでプロキシ接続を行うことができません。mode を OCI_SPC_HOMOGENEOUS に設定した場合は、プロキシ接続を行うことができません。

userid と password のペアは、OCISessionGet() コールの認証ハンドルを使用して指定することもできます。mode を OCI_SESSGET_CREDPROXY に設定してこの関数をコールした場合、ユーザーには、OCISessionGet() コールで提供された userid によって認証されるセッションが、OCISessionPoolCreate() コールで提供されたプロキシ資格証明を通じて与えられます。この場合、OCISessionGet() コールのパスワードは無視されます。

mode を OCI_SESSGET_CREDPROXY に設定しないで OCISessionGet() をコールした場合、ユーザーは、OCISessionGet() コールで提供された資格証明によって認証されるダイレクト・セッションを取得します。このコールで資格証明が提供されなかった場合、ユーザーは、OCISessionPoolCreate() コールの資格証明によって認証されるセッションを取得します。

関連関数

`OCISessionRelease()`、`OCISessionGet()`、`OCISessionPoolDestroy()`

OCISessionPoolDestroy()

用途

セッション・プールを破棄します。

構文

```
sword OCISessionPoolDestroy ( OCISPool      *spoolhp,  
                              OCIError      *errhp,  
                              ub4           mode );
```

パラメータ

spoolhp (IN/OUT)

破棄するセッション・プールのセッション・プール・ハンドルです。

errhp (IN/OUT)

OCIErrorGet() に渡すことができるエラー・ハンドルです。

mode (IN)

現在、OCISessionPoolDestroy() がサポートするモードは OCI_DEFAULT および OCI_SPD_FORCE です。

mode を OCI_SPD_FORCE に設定してこの関数をコールし、アクティブなセッションがプール内に存在する場合、そのセッションはクローズし、プールは破棄されます。ただし、このモードを設定せず、ビジーなセッションがプール内に存在する場合は、エラーが戻されます。

関連関数

[OCISessionPoolCreate\(\)](#)、[OCISessionRelease\(\)](#)、[OCISessionGet\(\)](#)

OCISessionRelease()

用途

この関数は、OCISessionGet() を使用して取得したセッションを解放するために使用します。このコールの動作は、対応する OCISessionGet() 関数をコールしたときの *mode* によって決まります。デフォルトでは、セッション / 接続をクローズします。接続プーリングの場合、この関数は、セッションをクローズして接続をプールに戻します。セッション・プーリングの場合、この関数は、セッションと接続のペアをプールに戻します。

構文

```
sword OCISessionRelease ( OCISvcCtx      *svchp,  
                           OCIError      *errhp,  
                           Oratext       *tag,  
                           ub4           tag_len,  
                           ub4           mode );
```

パラメータ

svchp (IN)

対応する OCISessionGet() のコール時に移入されたサービス・コンテキストです。

デフォルトでは、このハンドルに関連付けられたセッションおよび接続はクローズされます。

接続プーリングの場合は、セッションがクローズされ、接続がプールに対して解放されます。

セッション・プーリングの場合は、このサービス・コンテキストに関連付けられたセッションと接続がプールに対して解放されます。

errhp (IN/OUT)

OCI エラー・ハンドルです。

tag (IN)

このパラメータは、セッション・プーリングでのみ使用します。

OCI_SESSRLS_RETAG モードが指定されていない場合、このパラメータは無視されます。その場合、セッションには、このタグを使用してラベルが付けられ、プールに戻されます。このパラメータが NULL の場合、セッションにタグは付けられません。

tag_len (IN)

このパラメータは、セッション・プーリングでのみ使用します。

タグの長さです。OCI_SESSRLS_RETAG モードが設定されていない場合、このパラメータは無視されます。

mode (IN)

次のモードがサポートされています。

- OCI_DEFAULT
- OCI_SESSRLS_DROPSESS
- OCI_SESSRLS_RETAG

接続プーリングの場合、デフォルトでは、OCI_DEFAULT のみ使用できます。

OCI_SESSRLS_DROPSESS および OCI_SESSRLS_RETAG は、セッション・プーリングでのみ使用できます。

OCI_SESSRLS_DROPSESS に設定すると、セッションはセッション・プールから削除されます。

OCI_SESSRLS_RETAG に設定した場合のみ、セッションのタグを変更できます。このモード以外に設定すると、tag パラメータおよび tag_len パラメータは無視されます。

コメント

このコールでは、正しいタグを渡すように注意してください。デフォルト・セッションが要求され、ユーザーが (ALTER SESSION コマンドを使用して) そのセッションの特定のプロパティを設定する場合、ユーザーは、正しいタグを使用してこのセッションにラベルを付ける必要があります。

これとは逆に、ユーザーがタグ付けされたセッションを要求して取得し、そのセッションのプロパティを変更した場合、ユーザーは、別のタグ (ある場合) を渡す必要があります。

セッション・プール・レイヤーで正しく処理が実行されるために、アプリケーション開発者は、正しいタグを OCISessionGet () コールおよび OCISessionRelease () コールに渡すように注意してください。

関連関数

[OCISessionGet \(\)](#)、[OCISessionPoolCreate \(\)](#)、[OCISessionPoolDestroy \(\)](#)、[OCILogon2 \(\)](#)

OCITerminate()

用途

共有メモリー・サブシステムからプロセスを連結解除して共有メモリーを解放します。

構文

```
sword OCITerminate ( ub4      mode );
```

パラメータ

mode (IN)

コール固有モードです。有効な値は次のとおりです。

- OCI_DEFAULT — デフォルトのコールを実行します。

コメント

OCITerminate() は、各プロセス内で1回のみコールしてください。OCIInitialize() コールと対でコールします。このコールは共有メモリー・サブシステムからプロセスを連結解除してプロセスを終了します。また、プロセス・クリーンアップ操作も行います。同一の共有メモリーに接続している2つ以上のプロセスが、OCITerminate() を同時にコールしている場合は、最も高速のプロセスによって、共有メモリー・サブシステムが完全に解放されて、速度の遅いプロセスは異常終了することになります。

関連関数

[OCIInitialize\(\)](#)

ハンドル関数および記述子関数

この項では、OCI ハンドル関数および記述子関数について説明します。

表 15-2 ハンドル関数および記述子関数

関数	用途
OCIAttrGet() (15-48 ページ)	ハンドルの属性を取得します。
OCIAttrSet() (15-50 ページ)	ハンドルまたは記述子の属性を設定します。
OCIDescriptorAlloc() (15-52 ページ)	記述子または LOB ロケータの割当てと初期化を行います。
OCIDescriptorFree() (15-54 ページ)	割当て済みの記述子を解放します。
OCIHandleAlloc() (15-56 ページ)	ハンドルの割当てと初期化を行います。
OCIHandleFree() (15-59 ページ)	割当て済みのハンドルを解放します。
OCIParamGet() (15-61 ページ)	パラメータ記述子を取得します。
OCIParamSet() (15-63 ページ)	COR ハンドルにパラメータ記述子を設定します。

OCIAttrGet()

用途

このコールは、ハンドルの特定の属性を取得するときに使用します。

構文

```
sword OCIAttrGet ( CONST dvoid      *trgthndlp,
                   ub4              trgthndltyp,
                   dvoid            *attributep,
                   ub4              *sizep,
                   ub4              attrtype,
                   OCIError         *errhp );
```

パラメータ

trgthndlp (IN)

ハンドル・タイプのポインタです。実際のハンドルは、文ハンドルやセッション・ハンドルなどの場合があります。このコールによってエンコーディングを取得すると、ユーザーは環境ハンドルまたは文ハンドルと照合してチェックできます。

trgthndltyp (IN)

ハンドル・タイプです。次の型が有効です。

- OCI_DTYPE_PARAM、パラメータ記述子用
- OCI_HTYPE_STMT、文ハンドル用
- 表 2-1「OCI ハンドル・タイプ」に示されるすべてのハンドル・タイプ

attributep (OUT)

属性値の格納場所へのポインタです。OCI_UTF16 環境モードでは、文字列属性の値は UTF-16 文字列として戻されます。

sizep (OUT)

属性値のサイズは常にバイト単位です。これは、*attributep* が **dvoid** ポインタであるためです。非文字列属性のサイズは、OCI ライブラリですでに認識されているため、ほとんどの属性の値は、NULL として渡すことができます。**text*** パラメータでは、文字列の長さを取得するために **ub4** のポインタを渡す必要があります。

attrtype (IN)

取り出される属性のタイプです。属性タイプは、このマニュアルの次の章にリストされています。

関連項目： ハンドル・タイプとその読取り可能な属性については、[付録 A「ハンドルおよび記述子の属性」](#) のリストを参照してください。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

コメント

このコールは、ハンドルの特定の属性を取得するときに使用します。OCI_DTYPE_PARAM は暗黙的におよび明示的に記述する場合に使用します。このパラメータ記述子は、ダイレクト・パス・ロードでも使用します。暗黙的な記述の場合、パラメータ記述子には選択リストごとに列記述があります。明示的な記述の場合、パラメータ記述子には記述する各スキーマ・オブジェクトについての記述情報が含まれています。トップレベル・パラメータ記述子にそれ自身が記述子となる属性がある場合、OCIAttrGet () への後続コールで属性の型として OCI_ATTR_PARAM を使用します。

関連項目： コード・フラグメントの例については、6-23 ページの「[OCIDescribeAny\(\) の使用例](#)」および 4-12 ページの「[選択リスト項目の記述](#)」を参照してください。

このコールを使用して環境ハンドルまたは文ハンドルの Unicode 情報を取得します。

OCIAttrGet () と密接に関連している関数は、OCIDescribeAny () です。これは、表、ビュー、シノニム、プロシージャ、ファンクション、パッケージ、順序、型などの既存のスキーマ・オブジェクトを記述する汎用的な記述コールです。このコールの結果、記述ハンドルには、OCIAttrGet () コールを介して取得できるオブジェクト固有の属性が移入されます。

次に、この記述ハンドルに対する OCIParamGet () によって、指定位置のパラメータ記述子が戻されます。パラメータ位置は 1 から開始します。パラメータ記述子に対して OCIAttrGet () をコールすると、ストアド・プロシージャやストアド・ファンクションのパラメータの特定の属性、場合によっては、表の列記述子が戻されます。

OCIDescribeAny () によってスキーマ・オブジェクト記述全体がクライアント側にキャッシュされているため、これらの後続コールは、サーバーへのラウンドトリップを別に行う必要がありません。記述ハンドルに対する OCIAttrGet () コールによって、位置の総数も戻すことができます。

特に UTF-16 モードでループを実行する場合は、属性に対応している同じポインタ変数を再利用し、OCIAttrGet () のコール後にその内容をローカル変数にコピーしてください。複数のポインタを同じ属性に使用すると、メモリー・リークが発生する可能性があります。

関連関数

[OCIAttrSet \(\)](#)

OCIAttrSet()

用途

このコールは、ハンドルまたは記述子の特定の属性を設定するときに使用されます。

構文

```
sword OCIAttrSet ( dvoid          *trgthndlp,
                  ub4            trgthndltyp,
                  dvoid          *attributep,
                  ub4            size,
                  ub4            attrtype,
                  OCIError       *errhp );
```

パラメータ

trgthndlp (IN/OUT)

その属性が変更されるハンドル・タイプのポインタです。

trgthndltyp (IN/OUT)

ハンドル・タイプです。

attributep (IN)

属性値のポインタです。この属性値がターゲット・ハンドルにコピーされます。属性値がポインタの場合は、そのポインタのみがコピーされ、ポインタの内容はコピーされません。文字列の属性は、OCI_UTF16 環境では UTF-16 であることが必要です。

size (IN)

属性値のサイズです。サイズは OCI ライブラリですでにわかっているため、ほとんどの属性に対して 0（ゼロ）で渡すことができます。**text*** 属性では、エンコーディングに関係なく、バイト単位で文字列の長さを設定するために **ub4** で渡す必要があります。

attrtype (IN)

設定される属性のタイプです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

コメント

ハンドル・タイプとその書込み可能な属性については、[付録 A「ハンドルおよび記述子の属性」](#) のリストを参照してください。

例

次のコード例は、アプリケーションの開始部分で `OCIAttrSet()` を何回か使用している例です。

```
int main()
{
    OCIEnv *envhp;
    OCIServer *srvhp;
    OCIError *errhp;
    OCISvcCtx *svchp;
    OCISmt *stmthp;
    OCISession *usrhp;

    OCIInitialize((ub4) OCI_THREADED | OCI_OBJECT, (dvoid *)0,
        (dvoid * (*)()) 0, (dvoid * (*)()) 0, (void (*)()) 0 );
    OCIHandleAlloc( (dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
        0, (dvoid **) &tmp);
    OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 0, (dvoid **) &tmp );
    OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4)
        OCI_HTYPE_ERROR, 0, (dvoid **) &tmp);
    OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4)
        OCI_HTYPE_SERVER, 0, (dvoid **) &tmp);
    OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);
    OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp,
        (ub4) OCI_HTYPE_SVCCTX, , (dvoid **) &tmp);
    /* set attribute server context in the service context */
    OCIAttrSet( (dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *) srvhp,
        (ub4) 0, (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);
    /* allocate a user session handle */
    OCIHandleAlloc((dvoid *)envhp, (dvoid **)&usrhp,
        (ub4) OCI_HTYPE_SESSION, (size_t) 0, (dvoid **) 0);
    OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION, (dvoid *)"sherry",
        (ub4)strlen("sherry"), OCI_ATTR_USERNAME, errhp);
    OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION, (dvoid *)"penfield",
        (ub4)strlen("penfield"), OCI_ATTR_PASSWORD, errhp);
    checkerr(errhp, OCISessionBegin (svchp, errhp, usrhp, OCI_CRED_RDBMS,
        OCI_DEFAULT));
    OCIAttrSet((dvoid *)svchp, (ub4)OCI_HTYPE_SVCCTX, (dvoid *)usrhp,
        (ub4)0, OCI_ATTR_SESSION, errhp);
```

関連関数

[OCIAttrGet\(\)](#)

OCIDescriptorAlloc()

用途

記述子または LOB ロケータを格納する記憶域を割り当てます。

構文

```
sword OCIDescriptorAlloc ( CONST dvoid   *parenth,
                           dvoid        **descpp,
                           ub4          type,
                           size_t       xtrmem_sz,
                           dvoid        **usrmemp);
```

パラメータ

parenth (IN)

環境ハンドルです。

descpp (OUT)

目的の型の記述子または LOB ロケータを戻します。

type (IN)

割り当てられる記述子または LOB ロケータの型を指定します。

- OCI_DTYPE_SNAP – C 型のスナップショット記述子 **OCISnapshot** の生成を指定します。
- OCI_DTYPE_LOB – C 型の LOB 値型ロケータ (BLOB または CLOB の場合) **OCILobLocator** の生成を指定します。
- OCI_DTYPE_FILE – C 型の FILE 値型ロケータ **OCILobLocator** の生成を指定します。
- OCI_DTYPE_ROWID – C 型の ROWID 記述子 **OCIRowid** の生成を指定します。
- OCI_DTYPE_DATE – C 型の ANSI DATE 記述子 **OCIDateTime** の生成を指定します。
- OCI_DTYPE_TIMESTAMP – C 型の TIMESTAMP 記述子 **OCIDateTime** の生成を指定します。
- OCI_DTYPE_TIMESTAMP_TZ – C 型の TIMESTAMP WITH TIME ZONE 記述子 **OCIDateTime** の生成を指定します。
- OCI_DTYPE_TIMESTAMP_LTZ – C 型の TIMESTAMP WITH LOCAL TIME ZONE 記述子 **OCIDateTime** の生成を指定します。
- OCI_DTYPE_INTERVAL_YM – C 型の INTERVAL YEAR TO MONTH 記述子 **OCIInterval** の生成を指定します。
- OCI_DTYPE_INTERVAL_DS – C 型の INTERVAL DAY TO SECOND 記述子 **OCIInterval** の生成を指定します。

- OCI_DTYPE_COMPLEXOBJECTCOMP – C 型の複合オブジェクト検索記述子 **OCIComplexObjectComp** の生成を指定します。
- OCI_DTYPE_AQENQ_OPTIONS – C 型のアドバンスト・キューイング・エンキュー・オプション記述子 **OCIAQEnqOptions** の生成を指定します。
- OCI_DTYPE_AQDEQ_OPTIONS – C 型のアドバンスト・キューイング・デキュー・オプション記述子 **OCIAQDeqOptions** の生成を指定します。
- OCI_DTYPE_AQMSG_PROPERTIES – C 型のアドバンスト・キューイング・メッセージ・プロパティ記述子 **OCIAQMsgProperties** の生成を指定します。
- OCI_DTYPE_AQAGENT – C 型のアドバンスト・キューイング・エージェント記述子 **OCIAQAgent** の生成を指定します。

xtrmem_sz (IN)

記述子の存続期間中アプリケーションで使用するために割り当てられるユーザー・メモリーの量を指定します。

usrmempp (OUT)

記述子の存続期間中コールによってユーザー用に割り当てられた、サイズ *xtrmem_sz* のユーザー・メモリーのポインタを戻します。

コメント

type で指定された型に対応する、割当ておよび初期化済みの記述子のポインタを戻します。成功した場合は、非 NULL 記述子または LOB ロケータが戻ります。エラー発生時に診断は利用できません。

このコールは、成功した場合は OCI_SUCCESS を、メモリー不足エラーが発生した場合は OCI_INVALID_HANDLE を戻します。

関連項目： *xtrmem_sz* パラメータおよびユーザー・メモリー割当ての詳細は、2-14 ページの「[ユーザー・メモリーの割当て](#)」を参照してください。

関連関数

[OCIDescriptorFree\(\)](#)

OCIDescriptorFree()

用途

割当て済み記述子の割当てを解除します。

構文

```
sword OCIDescriptorFree ( dvoid      *descp,  
                          ub4        type );
```

パラメータ

descp (IN)

割り当てられた記述子です。

type (IN)

解放する記憶域の型を指定します。指定する型を次に示します。

- OCI_DTYPE_SNAP — スナップショット記述子
- OCI_DTYPE_LOB — LOB 値型記述子
- OCI_DTYPE_FILE — FILE 値型記述子
- OCI_DTYPE_ROWID — ROWID 記述子
- OCI_DTYPE_DATE — ANSI DATE 記述子
- OCI_DTYPE_PARAM — パラメータ記述子
- OCI_DTYPE_TIMESTAMP — TIMESTAMP 記述子
- OCI_DTYPE_TIMESTAMP_TZ — TIMESTAMP WITH TIME ZONE 記述子
- OCI_DTYPE_TIMESTAMP_LTZ — TIMESTAMP WITH LOCAL TIME ZONE 記述子
- OCI_DTYPE_INTERVAL_YM — INTERVAL YEAR TO MONTH 記述子
- CI_DTYPE_INTERVAL_DS — INTERVAL DAY TO SECOND 記述子
- OCI_DTYPE_COMPLEXOBJECTCOMP — 複合オブジェクト検索記述子
- OCI_DTYPE_AQENQ_OPTIONS — AQ エンキュー・オプション記述子
- OCI_DTYPE_AQDEQ_OPTIONS — AQ デキュー・オプション記述子
- OCI_DTYPE_AQMSG_PROPERTIES — AQ メッセージ・プロパティ記述子
- OCI_DTYPE_AQAGENT — AQ エージェント記述子

コメント

このコールにより、記述子に対応付けられた記憶域が解放されます。OCI_SUCCESS または OCI_INVALID_HANDLE を戻します。記述子はすべて明示的に割当てを解除できますが、環境ハンドルの割当てを解除すると、OCI は記述子の割当てを自動的に解除します。

関連関数

`OCIDescriptorAlloc()`

OCIHandleAlloc()

用途

このコールは、割当ておよび初期化済みハンドルのポインタを戻します。

構文

```
sword OCIHandleAlloc ( CONST dvoid    *parenth,
                      dvoid          **hndlpp,
                      ub4            type,
                      size_t         xtrmem_sz,
                      dvoid          **usrmempp );
```

パラメータ

parenth (IN)

環境ハンドルです。

hndlpp (OUT)

ハンドルを戻します。

type (IN)

割り当てられるハンドルの型を指定します。使用できる型を次に示します。

- OCI_HTYPE_AUTHINFO – C 型の認証情報ハンドル **OCIAuthInfo** の生成を指定します。
- OCI_HTYPE_COMPLEXOBJECT – C 型の複合オブジェクト検索ハンドル **OCIComplexObject** の生成を指定します。
- OCI_HTYPE_SECURITY – C 型のセキュリティ・ハンドル **OCISecurity** の生成を指定します。
- OCI_HTYPE_CPOOL – C 型の接続プーリング・ハンドル **OCICPool** の生成を指定します。
- OCI_HTYPE_DIRPATH_CTX – C 型のダイレクト・パス・コンテキスト・ハンドル **OCIDirPathCtx** の生成を指定します。
- OCI_HTYPE_DIRPATH_COLUMN_ARRAY – C 型のダイレクト・パス列配列ハンドル **OCIDirPathColArray** の生成を指定します。
- OCI_HTYPE_DIRPATH_STREAM – C 型のダイレクト・パス・ストリーム・ハンドル **OCIDirPathStream** の生成を指定します。
- OCI_HTYPE_ENV – C 型の環境ハンドル **OCIEnv** の生成を指定します。
- OCI_HTYPE_ERROR – C 型のエラー・レポート・ハンドル **OCIError** の生成を指定します。

- `OCI_HTYPE_SVCCTX` – C 型のサービス・コンテキスト・ハンドル **`OCISvcCtx`** の生成を指定します。
- `OCI_HTYPE_STMT` – C 型の文（アプリケーション要求）ハンドル **`OCIStmt`** の生成を指定します。
- `OCI_HTYPE_DESCRIBE` – C 型の選択リスト記述ハンドル **`OCIDescribe`** の生成を指定します。
- `OCI_HTYPE_SERVER` – C 型のサーバー・コンテキスト・ハンドル **`OCISever`** の生成を指定します。
- `OCI_HTYPE_SESSION` – C 型のユーザー・セッション・ハンドル **`OCISession`** の生成を指定します。
- `OCI_HTYPE_TRANS` – C 型のトランザクション・コンテキスト・ハンドル **`OCITrans`** の生成を指定します。
- `OCI_HTYPE_SPOOL` – C 型のセッション・プール・ハンドル **`OCISPool`** の生成を指定します。
- `OCI_HTYPE_SUBSCR` – C 型のサブスクリプション・ハンドル **`OCISubscription`** の生成を指定します。
- `OCI_HTYPE_PROCESS` – C 型のプロセス・ハンドル **`OCIProcess`** の生成を指定します。

`xtramem_sz` (IN)

割り当てられるユーザー・メモリーの量を指定します。

`usrmempp` (OUT)

コールによりユーザーに割り当てられた、サイズ `xtramem_sz` のユーザー・メモリーのポインタを戻します。

コメント

`type` で指定された型に対応する、割当ておよび初期化済みのハンドルのポインタを戻します。成功時には非 `NULL` ハンドルが戻ります。すべてのハンドルが、親ハンドルとして渡される環境ハンドルと対応付けて割り当てられます。

エラー発生時に診断は利用できません。このコールは、成功した場合は `OCI_SUCCESS` を、エラーが発生した場合は `OCI_INVALID_HANDLE` を戻します。

ハンドルは、OCI コールに渡す前に `OCIHandleAlloc()` を使用して割り当てる必要があります。

環境ハンドルの割当ておよび初期化を行うには、`OCIEnvInit()` をコールします。

関連項目： `xtramem_sz` パラメータを使用したユーザー・メモリー割当ての詳細は、2-14 ページの「ユーザー・メモリーの割当て」を参照してください。

例

次のコード例は、OCIHandleAlloc() を使用して、アプリケーションの開始時に様々なハンドルを割り当てる例です。

```
OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4)
    OCI_HTYPE_ERROR, 0, (dvoid **) &tmp);
OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4)
    OCI_HTYPE_SERVER, 0, (dvoid **) &tmp);
OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4)
    OCI_HTYPE_SVCCTX, 0, (dvoid **) &tmp);
```

関連関数

[OCIHandleFree\(\)](#)、[OCIEnvInit\(\)](#)

OCIHandleFree()

用途

このコールは、ハンドルの割当てを明示的に解除します。

構文

```
sword OCIHandleFree ( dvoid      *hndlp,  
                      ub4        type );
```

パラメータ

hndlp (IN)

OCIHandleAlloc() により割り当てられるハンドルです。

type (IN)

解放する記憶域の型を指定します。指定する型を次に示します。

- OCI_HTYPE_CPOOL — 接続プール・ハンドル
- OCI_HTYPE_ENV — 環境ハンドル
- OCI_HTYPE_ERROR — エラー・レポート・ハンドル
- OCI_HTYPE_SVCCTX — サービス・コンテキスト・ハンドル
- OCI_HTYPE_STMT — 文（アプリケーション要求）ハンドル
- OCI_HTYPE_DESCRIBE — 選択リスト記述ハンドル
- OCI_HTYPE_SERVER — サーバー・ハンドル
- OCI_HTYPE_SESSION — ユーザー・セッション・ハンドル
- OCI_HTYPE_TRANS — トランザクション・ハンドル
- OCI_HTYPE_COMPLEXOBJECT — 複合オブジェクト検索ハンドル
- OCI_HTYPE_SECURITY — セキュリティ・ハンドル
- OCI_HTYPE_SUBSCR — サブスクリプション・ハンドル
- OCI_HTYPE_DIRPATH_CTX — ダイレクト・パス・コンテキスト・ハンドル
- OCI_HTYPE_DIRPATH_COLUMN_ARRAY — ダイレクト・パス列配列ハンドル
- OCI_HTYPE_DIRPATH_STREAM — ダイレクト・パス・ストリーム・ハンドル
- OCI_HTYPE_PROCESS — プロセス・ハンドル

コメント

このコールは、ハンドルに対応付けられている記憶域で、`type` パラメータに指定された型に該当するものを解放します。

このコールは、`OCI_SUCCESS` または `OCI_INVALID_HANDLE` を戻します。

ハンドルはすべて明示的に割当て解除できます。OCI では、親ハンドルの割当てを解除すると、子ハンドルの割当てが自動的に解除されます。

関連関数

`OCIHandleAlloc()`、`OCIEnvInit()`

OCIParamGet()

用途

記述ハンドルまたは文ハンドル内の指定位置にあるパラメータの記述子を戻します。

構文

```
sword OCIParamGet ( CONST dvoid      *hndlp,
                    ub4              htype,
                    OCIError        *errhp,
                    dvoid            **parmdpp,
                    ub4              pos );
```

パラメータ

hndlp(IN)

文ハンドルまたは記述ハンドルです。OCIParamGet () 関数は、このハンドル用のパラメータ記述子を戻します。

htype (IN)

hndlp パラメータに渡されるハンドルの型です。次の型が有効です。

- OCI_DTYPE_PARAM、パラメータ記述子用
- OCI_HTYPE_COMPLEXOBJECT、複合オブジェクト検索ハンドル用
- OCI_HTYPE_STMT、文ハンドル用

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

parmdpp (OUT)

ハンドル・タイプ OCI_DTYPE_PARAM の、*pos* パラメータで指定された位置におけるパラメータの記述子です。

pos (IN)

文ハンドルまたは記述ハンドルにおける位置番号です。この位置にあるパラメータ記述子が戻ります。

注意： 指定位置にパラメータ記述子がない場合は、OCI_ERROR が戻されます。

コメント

このコールは、記述ハンドルまたは文ハンドル内の指定位置にあるパラメータの記述子に戻します。パラメータ記述子は、常に OCI ライブラリによって内部的に割り当てられます。また、OCIDescriptorFree() を使用して解放できます。たとえば、文を実行するたびに同じ列のメタデータをフェッチする場合は、OCIParamGet() の各コール間でパラメータ記述子を明示的に解放しないかぎり、プログラムではメモリーがリークします。

関連項目： パラメータ記述子の属性の詳細は、[付録 A「ハンドルおよび記述子の属性」](#) を参照してください。

関連関数

[OCIAttrGet\(\)](#)、[OCIAttrSet\(\)](#)、[OCIParamSet\(\)](#)、[OCIDescriptorFree\(\)](#)

OCIParamSet()

用途

COR ハンドルへの複合オブジェクト検索（COR）記述子の設定に使用します。

構文

```
sword OCIParamSet ( dvoid          *hndlp,
                    ub4            htype,
                    OCIError       *errhp,
                    CONST dvoid    *dscp,
                    ub4            dtyp,
                    ub4            pos );
```

パラメータ

hndlp (IN/OUT)

ハンドル・ポインタです。

htype (IN)

ハンドル・タイプです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

dscp (IN)

複合オブジェクト検索記述子ポインタです。

dtyp (IN)

記述子型です。COR 記述子の型は、OCI_DTYPE_COMPLEXOBJECTCOMP です。

pos (IN)

位置番号です。

コメント

COR ハンドルには [OCIHandleAlloc\(\)](#) を、記述子には [OCIDescriptorAlloc\(\)](#) を使用して、それぞれ前もって割り当てておく必要があります。記述子の属性は、[OCIAttrSet\(\)](#) を使用して設定します。

関連項目： 複合オブジェクト検索の詳細は、10-20 ページの「[複合オブジェクト検索](#)」を参照してください。

関連関数

[OCIParamGet\(\)](#)

バインド関数、定義関数および記述関数

この項では、バインド関数、定義関数および記述関数について説明します。

表 15-3 バインド関数、定義関数および記述関数

関数	用途
OCIBindArrayOfStruct() (15-65 ページ)	静的配列バインド用のスキップ・パラメータを設定します。
OCIBindByName() (15-66 ページ)	名前によりバインドします。
OCIBindByPos() (15-71 ページ)	位置によりバインドします。
OCIBindDynamic() (15-75 ページ)	OCI_DATA_AT_EXEC モードでバインドした後、追加属性を設定します。
OCIBindObject() (15-79 ページ)	名前付きデータ型のバインド用の追加属性を設定します。
OCIDefineArrayOfStruct() (15-81 ページ)	静的配列定義用の追加属性を設定します。
OCIDefineByPos() (15-82 ページ)	出力変数の関連性を定義します。
OCIDefineDynamic() (15-86 ページ)	OCI_DYNAMIC_FETCH モードでの定義用の追加属性を設定します。
OCIDefineObject() (15-89 ページ)	名前付きデータ型の定義用の追加属性を設定します。
OCIDescribeAny() (15-91 ページ)	既存のスキーマ・オブジェクトを記述します。
OCIStmtGetBindInfo() (15-94 ページ)	バインドおよびインジケータ変数名とハンドルを取得します。

OCIBindArrayOfStruct()

用途

このコールは、静的配列バインド用のスキップ・パラメータを設定します。

構文

```
sword OCIBindArrayOfStruct ( OCIBind      *bindp,
                             OCIError     *errhp,
                             ub4          pvskip,
                             ub4          indskip,
                             ub4          alskip,
                             ub4          rcskip );
```

パラメータ

bindp (IN/OUT)

バインド構造体へのハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

pvskip (IN)

次のデータ値用のスキップ・パラメータです。

indskip (IN)

次のインジケータ値または構造体のためのスキップ・パラメータです。

alskip (IN)

次の実際の長さ値のためのスキップ・パラメータです。

rcskip (IN)

次の列レベルのリターン・コード値のためのスキップ・パラメータです。

コメント

このコールは、静的配列バインドに必要なスキップ・パラメータを設定します。これは、OCIBindByName () または OCIBindByPos () の後続コールです。初期バインド・コールによって戻されたバインド・ハンドルが OCIBindArrayOfStruct () コールのパラメータとして使用されます。

関連項目： スキップ・パラメータについては、5-25 ページの「[構造体配列のバインドと定義](#)」を参照してください。

関連関数

[OCIBindByName \(\)](#)、[OCIBindByPos \(\)](#)

OCIBindByName()

用途

プログラム変数と SQL 文または PL/SQL ブロック内のプレースホルダを関連付けます。

構文

```
sword OCIBindByName ( OCISstmt      *stmtp,
                      OCIBind      **bindpp,
                      OCIError      *errhp,
                      CONST text    *placeholder,
                      sb4            placeh_len,
                      dvoid          *valuep,
                      sb4            value_sz,
                      ub2            dty,
                      dvoid          *indp,
                      ub2            *alenp,
                      ub2            *rcodep,
                      ub4            maxarr_len,
                      ub4            *curelep,
                      ub4            mode );
```

パラメータ

stmtp (IN/OUT)

処理中の SQL または PL/SQL 文への文ハンドルです。

bindpp (IN/OUT)

このコールによって暗黙的に割り当てられるバインド・ハンドルのポインタを保存するポインタです。この特定の入力値に対するバインド情報はすべて、このバインド・ハンドルによってメンテナンスされます。このコールのデフォルトのエンコーディングは、*mode* パラメータに異なる値がないかぎり、*stmtp* での UTF-16 の設定に依存します。ハンドルは、文ハンドルの割当てが解除されたときに暗黙的に解放されます。入力時には、このポインタの値は NULL または有効なバインド・ハンドルにしてください。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

placeholder (IN)

名前で指定されるプレースホルダです。このプレースホルダは、文ハンドルに関連付けられた文の変数にマップされます。*placeholder* のエンコーディングは、環境のエンコーディングとの間で一貫性を常に保つ必要があります。つまり、文が UTF-16 で準備されているときは、プレースホルダも UTF-16 で準備します。文字列型パラメータのように、(*text* *) としてキャストし、NULL で終了する必要があります。

placeh_len (IN)

エンコーディングに関係なく、*placeholder* にバイト数で指定された名前の長さです。

valuep (IN/OUT)

データ値へのポインタまたは *dtty* パラメータで指定された型のデータ値の配列です。OCIAttrSet() 関数をコールして、OCI_ATTR_CHARSET_ID を OCI_UTF16ID (OCI_UCS2ID は使用できません) として設定した場合、このデータは UTF-16 (以前の名は UCS-2) 文字列になることがあります。OCI_UTF16ID は、OCI_UCS2ID にかわる新しい指定です。

さらに、OCIStmtPrepare() で説明したように、環境モードが OCI_UTF16 に設定されている場合は、ユーザーが OCIAttrSet() をコールして、バインド・ハンドル用に設定された文字を手動でリセットしないかぎり、*valuep* 文字列型のデフォルト・エンコーディングは、UTF-16 になります。

関連項目： A-35 ページの「[OCI_ATTR_CHARSET_ID](#)」を参照してください。

データ値配列は、PL/SQL 表にマップする、または SQL 複数行操作のデータを提供するために指定できます。バインド値配列が提供された場合、これは OCI 用語では配列バインドと呼ばれます。

SQLT_NTY または SQLT_REF バインドの場合、*valuep* パラメータは無視されます。OUT バッファへのポインタは、OCIBindObject() によって初期化される *pgvpp* パラメータ内に設定します。

value_sz(IN)

dvoid ポインタ *valuep* が指し示すデータ値のサイズ (バイト単位) です。バインド・バッファ *valuep* は、文字列型のこともあります、長さはバイト数で測定されます。これは、渡されるポインタが (*dvoid* *) 型であるためです。配列バインドの場合、これは、*alenp* パラメータに指定される実サイズで可能な要素の最大サイズになります。

クライアント・アプリケーションではサイズが不明な記述子、ロケータまたは REF の場合は、プログラマが渡す構造体のサイズ、つまり sizeof(OCIlobLocator *) が使用されます。

dtty (IN)

バインドされる値のデータ型です。名前付きデータ型 (SQLT_NTY) と REF (SQLT_REF) は、アプリケーションがオブジェクト・モードで初期化されている場合のみ有効です。名前付きデータ型または REF の場合は、バインド・ハンドルで追加コールを行い、データ型固有の属性を設定する必要があります。

indp (IN/OUT)

インジケータ変数または配列へのポインタです。SQLT_NTY 以外のデータ型の場合は、sb2 または sb2 の配列へのポインタです。

SQLT_NTY に対しては、このポインタは無視され、インジケータ構造体またはインジケータ構造体配列の実際のポインタが後続のコール OCIBindObject() で初期化されます。このパラメータは、動的バインドでは無視されます。

関連項目： 2-36 ページ [「インジケータ変数」](#)

alenp (IN/OUT)

配列要素の実際の長さの配列へのポインタです。alenp 内の各要素は、実行前後のバインド値配列内の該当要素のデータ長です。text 型として渡される文字列の長さはバイト単位にする必要があります。このパラメータは、動的バインドでは無視されます。

rancode (OUT)

列レベルのリターン・コードの配列へのポインタです。このパラメータは、動的バインドでは無視されます。

maxarr_len (IN)

PL/SQL バインドで可能なタイプ dty の要素の最大数です。このパラメータは、非 PL/SQL バインドでは必要ありません。maxarr_len が 0 (ゼロ) 以外であれば、OCIBindDynamic() または OCIBindArrayOfStruct() を呼び出して、追加バインド属性を設定できます。

curelep (IN/OUT)

実際の要素の数へのポインタです。このパラメータは、PL/SQL バインドでのみ必要です。

mode (IN)

コーディングの一貫性を保持するため、理論上このパラメータには、OCIStmtPrepare() で使用できる 3 つの値すべてを設定できます。バインド変数のエンコーディングは、常にこの変数を含む文のエンコーディングと同じであることが必要なため、ユーザーが文のエンコーディングと異なるエンコーディングを指定すると、エラーが発生します。そのため、モード設定は、OCI_DEFAULT にすることをお勧めします。この設定によって、バインド変数に文と同じエンコーディングが設定されます。

次のモードが有効です。

- OCI_DEFAULT — デフォルト・モード。文ハンドル stmtp は、親の環境ハンドルに指定されている内容を使用します。
- OCI_DATA_AT_EXEC — このモードが選択される場合、value_sz パラメータは、実行時に提供可能なデータの最大サイズを定義します。アプリケーションは、OCI ライブ ラリのランタイム IN データ・バッファをいつでも何回でも提供できる状態にしておく必要があります。ランタイム・データは、次の 2 つの方法のどちらかを使用して提供されます。

- OCIBindDynamic() への後続コールによって登録されるユーザー定義関数を使用するコールバック。
- OCI で提供されるコールを使用するポーリング・メカニズム。コールバックが定義されていない場合は、このモードになります。

関連項目： OCI_DATA_AT_EXEC モードの使用方法的詳細は、5-44 ページの「ランタイム・データ割当てとピース単位操作」を参照してください。

モードが OCI_DATA_AT_EXEC に設定されているときは、メイン・コールで *valuep*、*indp*、*alenp* および *rcodep* に値を指定しないでください。*indp* および *alenp* に対しては、0（ゼロ）を渡してください。値は、OCIBindDynamic() を使用し、登録されているコールバック関数を介して指定してください。

割り当てたバッファは、不要になった際にクライアント側で解放する必要があります。

コメント

このコールは、基本バインド操作を実行するときに使用します。バインドは、プログラム変数のアドレスと SQL 文または PL/SQL ブロック内のプレースホルダの関連付けを作成します。このバインド・コールは、バインドされるデータの型も指定し、実行時にデータを提供するメソッドも指示できます。

エンコーディングの決定は、文ハンドルの設定をデフォルトとして使用してバインド・ハンドルで行うか、*mode* パラメータを明示的に指定してその設定を上書きして行うことができます。

注意： OCIEnvNlsCreate() を使用して環境ハンドルを作成すると、バインド・ハンドルおよび定義ハンドルの実際の長さとは異なる長さは、常にバイト単位になります。

この関数は、*bindpp* パラメータによって指示されたバインド・ハンドルの暗黙的な割り当ても行います。****bindpp** に非 NULL ポインタが渡されると、このポインタは、OCIHandleAlloc() または OCIBindByName() のコールで以前に割り当てられた有効なハンドルを指し示します。

OCI アプリケーション内のデータは、プレースホルダに静的または動的にバインドできます。実行の直前にすべての IN バインド・データと OUT バインド・バッファが正しく定義された場合、バインドは静的になります。アプリケーションの要求によって、IN バインド・データと OUT バインド・バッファが実行時にクライアント・ライブラリに提供された場合、バインドは動的になります。動的バインドを指定するには、このコールの *mode* パラメータを OCI_DATA_AT_EXEC に設定します。

関連項目： 動的バインドの詳細は、5-44 ページの「ランタイム・データ割当てとピース単位操作」を参照してください。

`OCIBindByName()` と `OCIBindByPos()` は、パラメータとしてバインド・ハンドルを取りますが、これはバインド・コールによって暗黙的に割り当てられます。アプリケーションでバインドしているすべてのプレースホルダに別個のバインド・ハンドルが割り当てられます。

特定のデータ型をバインドする際または入力データを特定の方法で処理する際に必要となる特定の属性を指定する場合は、次の追加バインド・コールが必要です。

- 構造体配列が使用されている場合、`OCIBindArrayOfStruct()` をコールして必要なスキップ・パラメータを設定してください。
- 実行時にデータが動的に提供されている場合、アプリケーションでユーザー定義のコールバック関数を使用するときには `OCIBindDynamic()` をコールしてコールバックを登録します。
- `alenp` に 64KB を超える長さが必要な場合は、`OCIBindDynamic()` を使用します。
- 名前付きデータ型がバインドされている場合、`OCIBindObject()` をコールしてその他の必要情報を指定する必要があります。
- `RETURNING` 句を含む文を使用する場合は、このコールの後に `OCIBindDynamic()` をコールする必要があります。

関連関数

`OCIBindDynamic()`、`OCIBindObject()`、`OCIBindArrayOfStruct()`

OCIBindByPos()

用途

プログラム変数と SQL 文または PL/SQL ブロック内のプレースホルダを関連付けます。

構文

```
sword OCIBindByPos ( OCISmt      *stmt,
                    OCIBind      **bindpp,
                    OCIError      *errhp,
                    ub4           position,
                    dvoid         *valuep,
                    sb4           value_sz,
                    ub2           dty,
                    dvoid         *indp,
                    ub2           *alenp,
                    ub2           *rcodep,
                    ub4           maxarr_len,
                    ub4           *curelep,
                    ub4           mode );
```

パラメータ

stmt (IN/OUT)

処理中の SQL または PL/SQL 文への文ハンドルです。

bindpp (IN/OUT)

このコールによって暗黙的に割り当てられるバインド・ハンドルのアドレスです。この特定の入力値に対するバインド情報はすべて、このバインド・ハンドルによって メンテナンスされます。ハンドルは、文ハンドルの割当てが解除されたときに暗黙的に解放されます。入力時には、このポインタの値は NULL または有効なバインド・ハンドルにしてください。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

position (IN)

OCIBindByPos () がコールされている場合、プレースホルダの属性が位置別に指定されます。

valuep (IN/OUT)

データ値のアドレスまたは dty パラメータで指定されたタイプのデータ値配列です。データ値配列は、PL/SQL 表にマップする、または SQL 複数行操作作用のデータを提供するために指定できます。バインド値配列が提供された場合、これは OCI 用語では配列バインドと呼ばれます。

SQLT_NTY または SQLT_REF バインドの場合、*valuep* パラメータは無視されます。OUT バッファへのポインタは、OCIBindObject() によって初期化される *pgvpp* パラメータ内に設定されます。

OCI_ATTR_CHARSET_ID 属性が OCI_UTF16ID（下位互換性のために保持されている、使用できない OCI_UCS2ID にかわる指定）に設定されている場合は、対応するバインド・コールとの間で受渡しを行うデータは、すべて UTF-16 エンコーディングとみなされます。

関連項目： A-35 ページ [OCI_ATTR_CHARSET_ID](#)

value_sz (IN)

データ値のサイズ。配列バインドの場合、これは、*alenp* パラメータに指定される実サイズで可能な要素の最大サイズになります。

クライアント・アプリケーションではサイズが不明な記述子、ロケータまたは REF の場合は、プログラマが渡す構造体のサイズ、たとえば `sizeof (OCILobLocator *)` などが使用されます。

dtty (IN)

バインドされる値のデータ型です。名前付きデータ型 (SQLT_NTY) と REF (SQLT_REF) は、アプリケーションがオブジェクト・モードで初期化されている場合のみ有効です。名前付きデータ型または REF の場合は、バインド・ハンドルで追加コールを行い、データ型固有の属性を設定する必要があります。

indp (IN/OUT)

インジケータ変数または配列へのポインタです。すべてのデータ型について、これは *sb2* または *sb2* の配列へのポインタです。ただ 1 つの例外は SQLT_NTY です。SQLT_NTY に対してはこのポインタは無視され、インジケータ構造体の実際のポインタまたはインジケータ構造体配列が OCIBindObject() により初期化されます。動的バインドでは無視されます。

関連項目： 2-36 ページの「[インジケータ変数](#)」を参照してください。

alenp (IN/OUT)

配列要素の実際の長さの配列へのポインタです。*alenp* 内の各要素は、実行の前後ともに、バインド値配列内の対応する要素のデータ長です（コードポイントの場合は、*valuep* のデータが Unicode でないかぎり、バイト単位です）。このパラメータは、動的バインドでは無視されます。

注意： *alenp* が *value_sz* より小さい場合、データはスキップされます。

rccodep (OUT)

列レベルのリターン・コードの配列へのポインタです。このパラメータは、動的バインドでは無視されます。

maxarr_len (IN)

PL/SQL バインドで可能なタイプ *dtty* の要素の最大数です。このパラメータは、非 PL/SQL バインドでは必要ありません。*maxarr_len* が 0 (ゼロ) 以外であれば、`OCIBindDynamic()` または `OCIBindArrayOfStruct()` を呼び出して、追加バインド属性を設定できます。

curelep (IN/OUT)

実際の要素の数へのポインタです。このパラメータは、PL/SQL バインドでのみ必要です。

mode (IN)

このパラメータに有効なモードは次のとおりです。

OCI_DEFAULT — これはデフォルト・モードです。

OCI_DATA_AT_EXEC — このモードが選択される場合、*value_sz* パラメータは、実行時に提供可能なデータの最大サイズを定義します。アプリケーションは、OCI ライブラリのランタイム IN データ・バッファをいつでも何回でも提供できる状態にしておく必要があります。ランタイム・データは、次の 2 つの方法のどちらかを使用して提供されます。

- `OCIBindDynamic()` への後続コールによって登録されるユーザー定義関数を使用するコールバック。
- OCI で提供されるコールを使用するポーリング・メカニズム。コールバックが定義されていない場合は、このモードになります。

関連項目： `OCI_DATA_AT_EXEC` モードの使用の詳細は、5-44 ページの「[ランタイム・データ割当てとピース単位操作](#)」を参照してください。

モードが `OCI_DATA_AT_EXEC` に設定されているときは、メイン・コールで *valuep*、*indp*、*alenp* および *rcodep* に値を指定しないでください。*indp* および *alenp* に対しては、0 (ゼロ) を渡してください。値は、`OCIBindDynamic()` を使用し、登録されているコールバック関数を介して指定してください。

割り当てたバッファは、不要になった際にクライアント側で解放する必要があります。

コメント

このコールは、基本バインド操作を実行するときに使用します。バインドは、プログラム変数のアドレスと SQL 文または PL/SQL ブロック内のプレースホルダの関連付けを作成します。このバインド・コールは、バインドされるデータの型も指定し、実行時にデータを提供するメソッドも指示できます。

注意： `OCIEnvNlsCreate()` を使用して環境ハンドルを作成すると、バインド・ハンドルおよび定義ハンドルの実際の長さと戻される長さは、常にバイト単位になります。

この関数は、`bindpp` パラメータによって指示されたバインド・ハンドルの暗黙的な割当ても行います。`**bindpp` に非 `NULL` ポインタが渡されると、このポインタは、`OCIHandleAlloc()` または `OCIBindByPos()` のコールで以前に割り当てられた有効なハンドルを指し示します。

OCI アプリケーション内のデータは、プレースホルダに静的または動的にバインドできます。実行の直前にすべての `IN` バインド・データと `OUT` バインド・バッファが正しく定義された場合、バインドは静的になります。アプリケーションの要求によって、`IN` バインド・データと `OUT` バインド・バッファが実行時にクライアント・ライブラリに提供された場合、バインドは動的になります。動的バインドを指定するには、このコールの `mode` パラメータを `OCI_DATA_AT_EXEC` に設定します。

関連項目： 動的バインドの詳細は、5-44 ページの「ランタイム・データ割当てとピース単位操作」を参照してください。

`OCIBindByName()` と `OCIBindByPos()` は、パラメータとしてバインド・ハンドルを取りますが、これはバインド・コールによって暗黙的に割り当てられます。アプリケーションでバインドしているすべてのプレースホルダに別個のバインド・ハンドルが割り当てられます。

特定のデータ型をバインドする際または入力データを特定の方法で処理する際に必要となる特定の属性を指定する場合は、次の追加バインド・コールが必要です。

- 構造体配列が使用されている場合、`OCIBindArrayOfStruct()` をコールして必要なスキップ・パラメータを設定してください。
- 実行時にデータが動的に提供されている場合、アプリケーションでユーザー定義のコールバック関数を使用するときには `OCIBindDynamic()` をコールしてコールバックを登録します。
- `alenp` に 64KB を超える長さが必要な場合は、`OCIBindDynamic()` を使用します。
- 名前付きデータ型がバインドされている場合、`OCIBindObject()` をコールしてその他の必要情報を指定する必要があります。
- `RETURNING` 句を含む文を使用する場合は、このコールの後に `OCIBindDynamic()` をコールする必要があります。

関連関数

`OCIBindDynamic()`、`OCIBindObject()`、`OCIBindArrayOfStruct()`

OCIBindDynamic()

用途

このコールは、動的データ割当て用のユーザー・コールバックを登録するときに使用します。

構文

```
sword OCIBindDynamic ( OCIBind      *bindp,
                      OCLError      *errhp,
                      dvoid          *ictxp,
                      OCICallbackInBind
                        dvoid          (*icbfp) (/*_
                        dvoid          *ictxp,
                        OCIBind      *bindp,
                        ub4           iter,
                        ub4           index,
                        dvoid         **bufpp,
                        ub4           *alenp,
                        ub1           *piecep,
                        dvoid         **indpp */),
                      dvoid          *octxp,
                      OCICallbackOutBind
                        dvoid          (*ocbfp) (/*_
                        dvoid          *octxp,
                        OCIBind      *bindp,
                        ub4           iter,
                        ub4           index,
                        dvoid         **bufpp,
                        ub4           *alenpp,
                        ub1           *piecep,
                        dvoid         **indpp,
                        ub2           **rcodepp */) );
```

パラメータ

bindp (IN/OUT)

OCIBindByName () または OCIBindByPos () のコールにより戻されるバインド・ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCLErrorGet () に渡すエラー・ハンドルです。

ictxp (IN)

コールバック関数 *icbfp* に必要なコンテキスト・ポインタです。

icbfp (IN)

実行時に IN バインド値またはピースのポインタを戻すコールバック関数です。コールバックは、次のパラメータを取ります。

ictxp (IN/OUT)

このコールバック関数用のコンテキスト・ポインタです。

bindp (IN)

このバインド変数を一意に指定するために渡されたバインド・ハンドルです。

iter (IN)

0（ゼロ）ベースの実行繰返し値。

index (IN)

カレント配列の索引（PL/SQL の配列バインド用）。SQL では行の索引です。値は 0（ゼロ）以上、バインド・コールの *curelep* パラメータ値以下です。

bufpp (OUT)

バッファまたは記憶域のポインタです。記述子の場合、**bufpp* にはその記述子へのポインタが含まれています。たとえば、次のような定義を行うとします。

```
OCILOBLocator    *lobp;
```

この場合、**bufpp* に **lobp* ではなく、*lobp* を設定します。

REF の場合はそのアドレスを渡します。つまり、**bufpp* に対して *&my_ref* を渡します。

OCI_ATTR_CHARSET_ID 属性が OCI_UTF16ID（下位互換性のために保持されている、使用できない OCI_UCS2ID にかわる指定）に設定されている場合は、対応するバインド・コールとの間で受渡しを行うデータは、すべて UTF-16 エンコーディングとみなされます。

関連項目： A-35 ページ [OCI_ATTR_CHARSET_ID](#)

alenp (OUT)

バインド値またはバインド・ピースを読み込んだ後それを保存するための OCI 用記憶域へのポインタです。記述子の場合、記述子にポインタのサイズを渡します。

例：`sizeof(OCILOBLocator *)`

piecep (OUT)

バインド値のいずれかのピース。この値は、OCI_ONE_PIECE、OCI_FIRST_PIECE、OCI_NEXT_PIECE または OCI_LAST_PIECE のいずれかです。ピース単位操作をサポートしないデータ型の場合、OCI_ONE_PIECE を渡す必要があります。OCI_ONE_PIECE を渡さないとエラーになります。

indp (OUT)

インジケータ値を含みます。これは、`sb2` 値のポインタ、あるいは名前付きデータ型をバインドするインジケータ構造体のポインタです。

octxp (IN)

コールバック関数 `ocbfp` に必要なコンテキスト・ポインタです。

ocbfp (IN)

実行時に OUT バインド値またはピースのポインタを戻すコールバック関数です。コールバックは、次のパラメータを取ります。

octxp (IN/OUT)

このコールバック関数用のコンテキスト・ポインタです。

bindp (IN)

このバインド変数を一意に指定するために渡されたバインド・ハンドルです。

iter (IN)

0 (ゼロ) ベースの実行繰返し値。

index (IN)

カレント配列の PL/SQL 用索引（配列バインド用）です。SQL の場合、カレント反復の行番号です。値は 0 (ゼロ) 以上、バインド・コールの `curelep` パラメータ値以下です。

bufpp (OUT)

バインド値またはバインド・ピースを書き込むバッファのポインタ。

`OCI_ATTR_CHARSET_ID` 属性が `OCI_UTF16ID`（下位互換性のために保持されている、使用できない `OCI_UCS2ID` にかわる指定）に設定されている場合は、対応するバインド・コールとの間で受渡しを行うデータは、すべて UTF-16 エンコーディングとみなされます。詳細は、A-35 ページの「[OCI_ATTR_CHARSET_ID](#)」を参照してください。

alenpp (IN/OUT)

バインド値またはバインド・ピースを読み込んだ後それを保存するための OCI 用記憶域へのポインタです。コードポイントである場合、このポインタはバイト単位です。ただし、Unicode エンコーディングの場合（`OCI_ATTR_CHARSET_ID` 属性が `OCI_UTF16ID` に設定されている場合）を除きます。

piecep (IN/OUT)

コールバック（アプリケーション）から Oracle へ、次のようにピース値を戻します。

- **IN** – 値は OCI_ONE_PIECE または OCI_NEXT_PIECE です。
- **OUT** – IN の値に応じて異なります。

IN 値が OCI_ONE_PIECE の場合、OUT 値は OCI_ONE_PIECE または OCI_FIRST_PIECE です。

IN 値が OCI_NEXT_PIECE の場合、OUT 値は OCI_NEXT_PIECE または OCI_LAST_PIECE です。

indpp (OUT)

インジケータ値（sb2 値または名前付きデータ型のインジケータ構造体のポインタ）を含むポインタを戻します。

rcodepp (OUT)

リターン・コードを含むポインタを戻します。

コメント

このコールは、OCIBindByName() または OCIBindByPos() の以前のコールで OCI_DATA_AT_EXEC モードが指定された場合に、データの用意または受取り用のユーザー定義コールバック関数を登録するときに使用されます。

コールバック関数ポインタは、コールが成功した場合には OCI_CONTINUE を戻します。OCI_CONTINUE 以外のリターン・コードは、クライアントが処理の即時終了を求めていることを示します。

関連項目： OCI_DATA_AT_EXEC モードの詳細は、5-44 ページの「[ランタイム・データ割当てとピース単位操作](#)」を参照してください。

記憶領域のアドレスを渡す際は、コールバックからアプリケーションが復帰した後も必ずその記憶領域が存在するようにしてください。したがって、このような記憶域はスタック上に割り当てないでください。

注意： OCIEnvNlsCreate() を使用して環境ハンドルを作成すると、バインド・ハンドルおよび定義ハンドルの実際の長さで戻される長さは、常にバイト単位になります。

関連関数

[OCIBindByName\(\)](#)、[OCIBindByPos\(\)](#)

OCIBindObject()

用途

この関数は、名前付きデータ型（オブジェクト）のバインドに必要な追加属性を設定します。

構文

```
sword OCIBindObject ( OCIBind          *bindp,
                     OCIError         *errhp,
                     CONST OCIType    *type,
                     dvoid            **pgvpp,
                     ub4              *pvszsp,
                     dvoid            **indpp,
                     ub4              *indszp, );
```

パラメータ

bindp (IN/OUT)

OCIBindByName() または OCIBindByPos() のコールにより戻されるバインド・ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet() に渡すエラー・ハンドルです。

type (IN)

バインドされているプログラム変数の型を示す TDO を指し示します。OCITypeByName() をコールして取り出します。SQL の REF ではオプションですが、PL/SQL の REF では必須です。

pgvpp (IN/OUT)

プログラム変数バッファのアドレスです。配列では、pgvpp はアドレス配列を示します。バインド変数が OUT 変数でもある場合は、オブジェクト・キャッシュ内に OUT 名前付きデータ型値または REF が割り当てられ、REF が戻されます。

pgvpp は、OCI_DATA_AT_EXEC モードが設定されている場合、無視されます。名前付きデータ型バッファは、実行時に要求されます。静的配列バインドでは、OCIBindArrayOfStruct() コールを使用してスキップ係数を指定できます。スキップ係数は、値に対する次のポインタのアドレスおよびインジケータ構造体、各自のサイズを計算するために使用されます。

pvszsp (OUT) [optional]

プログラム変数のサイズを指し示します。名前付きデータ型のサイズは、入力時には必要ありません。配列の場合、pvszsp は、ub4 の配列です。戻り時には、OUT バインド変数に対して、受け取った名前付きデータ型と REF のサイズを指し示します。

OCI_DATA_AT_EXEC モードが設定されている場合、*pvszsp* は無視されます。この場合、バッファのサイズは実行時に要求されます。

indpp (IN/OUT) [optional]

パラレル・インジケータ構造体を含むプログラム変数バッファのアドレスです。配列では、ポインタ配列を示します。バインド変数が OUT バインド変数でもある場合は、オブジェクト・キャッシュにメモリーが割り当てられ、OUT インジケータ値が格納されます。すべての OUT 値が受け取られた実行終了時には、*indpp* は、新たに割り当てられたこれらのインジケータ構造体のポインタを指し示します。SQLT_NTY バインドの場合のみ必要です。*indpp* は、OCI_DATA_AT_EXEC モードが設定されている場合無視されます。この場合、インジケータは実行時に要求されます。

indszp (IN/OUT)

IN インジケータ構造体プログラム変数のサイズを指し示します。配列の場合は、**sb2** の配列です。戻り時には、OUT バインド変数に対して、受け取った OUT インジケータ構造体のサイズを指し示します。OCI_DATA_AT_EXEC モードが設定されている場合、*indszp* は無視されます。この場合、インジケータのサイズは実行時に要求されます。

コメント

この関数は、名前付きデータ型または REF をバインドする追加属性を設定します。OCI 環境が非オブジェクト・モードで初期化されているときにこの関数をコールした場合は、エラーが戻ります。

この関数は、定義される名前付きデータ型用にデータ型 **OCIType** の型記述子オブジェクト (TDO) をパラメータとして取ります。TDO は、OCITypeByName () をコールして取り出すことができます。

OCIBindByName () または OCIBindByPos () に OCI_DATA_AT_EXEC モードが指定された場合、IN バッファのポインタは、OCIBindDynamic () コールに登録されたコールバック icbfp を使用するか、OCIStmtSetPieceInfo () コールによって取得されます。

バッファは OUT データに対して動的に割り当てられます。これらのバッファへのポインタは、次のいずれかの方法で戻されます。

- OCIBindDynamic () によって登録された ocbfp () をコールします。
- OCIStmtExecute () が OCI_NEED_DATA を戻したときにコールされた OCIStmtSetPieceInfo () によって渡されたバッファ内のバッファに対するポインタを設定します。

クライアント・ライブラリが割り当てたバッファのメモリーは、不要になったときに OCIObjectFree () コールを使用して解放する必要があります。

関連関数

[OCIBindByName \(\)](#)、[OCIBindByPos \(\)](#)

OCIDefineArrayOfStruct()

用途

このコールは、構造体配列（複数行、複数列）のフェッチで使用する静的配列定義に必要な追加属性を指定します。

構文

```
sword OCIDefineArrayOfStruct ( OCIDefine      *defnp,
                                OCIError      *errhp,
                                ub4            pvskip,
                                ub4            indskip,
                                ub4            rlskip,
                                ub4            rcskip );
```

パラメータ

defnp (IN/OUT)

OCIDefineByPos() のコールにより戻された定義構造体のハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet() に渡すエラー・ハンドルです。

pvskip (IN)

次のデータ値用のスキップ・パラメータです。

indskip (IN)

次のインジケータ位置のためのスキップ・パラメータです。

rlskip (IN)

次の戻り長さ値のためのスキップ・パラメータです。

rcskip (IN)

次のリターン・コードのためのスキップ・パラメータです。

コメント

このコールは、OCIDefineByPos() の後続コールです。アプリケーションでオブジェクトを含む構造体配列をバインドしている場合、最初に OCIDefineObject() をコールして、次に OCIDefineArrayOfStruct() をコールする必要があります。

関連項目： 5-26 ページ [「スキップ・パラメータ」](#)

関連関数

[OCIDefineByPos\(\)](#)、[OCIDefineObject\(\)](#)

OCIDefineByPos()

用途

選択リスト内の項目を型と出力データ・バッファに関連付けます。

構文

```
sword OCIDefineByPos ( OCISstmt      *stmtp,
                      OCIDefine     **defnpp,
                      OCIError      *errhp,
                      ub4            position,
                      dvoid          *valuep,
                      sb4            value_sz,
                      ub2            dtty,
                      dvoid          *indp,
                      ub2            *rlenp,
                      ub2            *rcodep,
                      ub4            mode );
```

パラメータ

stmtp (IN/OUT)

要求された SQL 問合せ操作へのハンドルです。

defnpp (IN/OUT)

定義ハンドルのポインタへのポインタです。このパラメータを NULL で渡すと、定義ハンドルが暗黙的に割り当てられます。再定義の場合は、非 NULL ハンドルをこのパラメータに渡すことができます。このハンドルは、この列用の定義情報を格納するために使用されます。

注意： ユーザーはこのポインタを追跡する必要があります。同じ列位置に対する 2 回目の OCIDefineByPos () のコールでは、同じポインタが戻されるとはかぎりません。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

position (IN)

選択リストでのこの値の位置です。位置は 1 から始まり、左から右へ番号が振られます。たとえば、次の SELECT 文があるとしたします。

```
SELECT empno, ssn, mgrno FROM employees;
```

empno は位置 1 に、ssn は位置 2 に、mgrno は位置 3 になります。

valuep (IN/OUT)

`dtty` パラメータに指定されているタイプのバッファまたはバッファの配列のポインタです。単独のフェッチ・コールで複数の行をフェッチする場合は、複数のバッファを指定できます。

value_sz (IN)

各 `valuep` バッファのバイト・サイズです。データが `VARCHAR2` 形式で内部的に格納されている場合、必要な文字数がバイト単位のバッファ・サイズと異なるときは、`OCIAttrSet()` を使用して追加指定できます。

マルチバイト変換環境では、指定したバイト数が、処理する文字数に対して不十分な場合、切捨てエラーが発生します。

`OCI_ATTR_CHARSET_ID` 属性が `OCI_UTF16ID`（下位互換性のために保持されている、使用できない `OCI_UCS2ID` にかわる指定）に設定されている場合は、対応する定義コールとの間で受渡しを行うデータは、すべて UTF-16 エンコーディングとみなされます。

関連項目： A-38 ページ [OCI_ATTR_CHARSET_ID](#)

dtty (IN)

データ型です。名前付きデータ型 (`SQLT_NTY`) と `REF` (`SQLT_REF`) は、環境がオブジェクト・モードで初期化されている場合のみ有効です。

`SQLT_CHAR` と `SQLT_LNG` を `CLOB` 列に、`SQLT_BIN` と `SQLT_LBI` を `BLOB` 列に指定できます。

関連項目： データ型コードおよび値については、第3章「データ型」のリストを参照してください。

indp (IN)

インジケータ変数または配列へのポインタです。スカラー・データ型の場合は、`sb2` または `sb2` の配列へのポインタです。`SQLT_NTY` 定義の場合は無視されます。`SQLT_NTY` 定義では、名前付きデータ型のインジケータ構造体または名前付きデータ型のインジケータ構造体配列のポインタは、後続の `OCIDefineObject()` コールによって関連付けられます。

関連項目： 2-36 ページ「インジケータ変数」

rlnep (IN/OUT)

フェッチされたデータの長さの配列のポインタです。`rlnep` 内の各要素は、フェッチ後の行にある対応する要素のデータ長です（コードポイントの場合は、`valuep` のデータが Unicode でないかぎり、バイト単位です）。

rcodep (OUT)

列レベルのリターン・コードの配列のポインタです。

mode (IN)

次のモードが有効です。

- OCI_DEFAULT – これはデフォルトのモードです。
- OCI_DYNAMIC_FETCH – フェッチするときにデータを動的に割り当てる必要があるアプリケーションでは、このモードを使用する必要があります。ユーザーは、OCIDefineDynamic() を追加でコールして、動的に割り当てられたバッファを受け取るために呼び出されるコールバック関数を設定できます。このモードでは、valuep および value_sz パラメータは無視されます。

コメント

このコールは、Oracle から取り出されたデータを受け取る出力バッファを定義します。この定義は、SELECT 文が OCI アプリケーションにデータを戻すときに必要なローカル・ステップです。

注意： OCIEnvNlsCreate() を使用して環境ハンドルを作成すると、バインド・ハンドルおよび定義ハンドルの実際の長さとして戻される長さは、常にバイト単位になります。

このコールは、選択リスト項目用の定義ハンドルの暗黙的な割り当ても行います。非 NULL ポインタが *defnpp に渡されると、OCI では OCIHandleAlloc() または OCIDefineByPos() のコールで以前に割り当てられた有効なハンドルを指し示します。別のアドレスに対してハンドルを再定義しているアプリケーションの場合はこれがあてはまるため、複数のフェッチに同じ定義ハンドルを再利用できます。

列をフェッチするための属性の定義は、1 つ以上のコールで実行されます。最初のコールは、フェッチを指定するために必要な最小限の属性を定義する OCIDefineByPos() です。

ある種のデータ型またはフェッチ・モードでは、OCIDefineByPos() のコールの後に、次の追加定義コールが必要です。

- 複数列を配列フェッチするためのスキップ・パラメータを設定するには、OCIDefineArrayOfStruct() のコールが必要です。
- 名前付きデータ型（つまり、オブジェクトやコレクション）または REF のフェッチに適切な属性を設定するには、OCIDefineObject() のコールが必要です。この場合、OCIDefineByPos() 内のデータ・バッファ・ポインタは無視されます。
- 名前付きデータ型の列を持つ複数行をフェッチするには、OCIDefineByPos() の後に、OCIDefineArrayOfStruct() と OCIDefineObject() の両方をコールする必要があります。

LOB 定義では、バッファ・ポインタは、OCIDescriptorAlloc() コールによって割り当てられる OCILobLocator 型の LOB ロケータのポインタにしてください。LOB 列には、LOB の値ではなく、常に LOB ロケータが戻ります。LOB 値は、フェッチしたロケータに対

して OCILOB コールを使用するとフェッチできます。これと同じ方式がすべての記述子データ型で使用されます。

NCHAR（固定長および可変長）では、バッファ・ポインタは、必要な NCHAR 文字を保持するのに十分なバイト配列を指し示している必要があります。

NESTED TABLE の列は、他のすべての名前付きデータ型と同じように定義およびフェッチされます。

記述子またはロケータの配列を定義するときに、記述子またはロケータのポインタ配列を渡す必要があります。

キャラクタ列の配列を定義するときに、文字バッファ配列を渡す必要があります。

このコールの *mode* パラメータに OCI_DYNAMIC_FETCH が設定された場合は、クライアント・アプリケーションから実行時にデータを動的にフェッチできます。ランタイム・データは、次の 2 つの方法で用意できます。

- OCIDefineDynamic() の後続コールによって登録する必要があるユーザー定義関数を使用するコールバック。クライアント・ライブラリがフェッチしたデータを戻すためにバッファが必要になると、このコールバックが呼び出され、用意されたランタイム・バッファがデータの一部分または全部を戻します。
- OCI で提供されるコールを使用するポーリング・メカニズム。コールバックが定義されていない場合は、このモードになります。この場合、フェッチ・コールにより OCI_NEED_DATA エラー・コードが戻され、データはピース単位のポーリング・メソッドで用意されます。

関連項目：

- OCI_DYNAMIC_FETCH モードの使用の詳細は、5-44 ページの「ランタイム・データ割当てとピース単位操作」を参照してください。
- 定義の詳細は、5-18 ページの「定義」を参照してください。

関連関数

[OCIDefineArrayOfStruct\(\)](#)、[OCIDefineDynamic\(\)](#)、[OCIDefineObject\(\)](#)

OCIDefineDynamic()

用途

このコールは、OCIDefineByPos() で OCI_DYNAMIC_FETCH モードが選択されたときに必要な追加属性を設定するために使用されます。

構文

```
sword OCIDefineDynamic ( OCIDefine    *defnp,
                        OCIError      *errhp,
                        dvoid          *octxp,
                        OCICallbackDefine (ocbfp) (/*_
                        dvoid          *octxp,
                        OCIDefine      *defnp,
                        ub4             iter,
                        dvoid          **bufpp,
                        ub4             **alenpp,
                        ub1             *piecep,
                        dvoid          **indpp,
                        ub2             **rcodep _*/) );
```

パラメータ

defnp (IN/OUT)

OCIDefineByPos() のコールにより戻される定義構造体へのハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet() に渡すエラー・ハンドルです。

octxp (IN)

コールバック関数のコンテキストを指し示します。

ocbfp (IN)

コールバック関数を指し示します。この関数は実行時にコールされ、検索されるフェッチ済みデータまたはそのピースを格納するバッファへのポインタを取得します。このコールバックは、インジケータ、リターン・コードおよびデータ・ピースとインジケータの長さも指定します。

注意： コールバック・パラメータを使用する場合は、パラメータ・モードに関する IN と OUT の意味を念頭に置いて作業する必要があります。通常、OCI 関数では、IN パラメータは Oracle に渡すデータを表し、OUT パラメータは Oracle から戻されるデータを表します。コールバックの場合、これが逆になります。IN は Oracle からコールバックに渡されるデータ、OUT はコールバックから Oracle に渡されるデータです。

次にコールバック・パラメータをリストします。

octxp (IN/OUT)

すべてのコールバック関数へ引数として渡されるコンテキスト・ポインタです。

defnp (IN)

定義ハンドルです。

iter (IN)

このカレント・フェッチのいずれかの列 (0 (ゼロ) ベース) です。

bufpp (OUT)

列値を格納するバッファへのポインタを戻します。つまり、**bufpp* は列値に対する適切な記憶域を指し示します。

alenpp (IN/OUT)

これは、**bufpp* に提供している記憶域のサイズを設定するために、アプリケーションで使用します。データがバッファにフェッチされると、*alenpp* はデータの実サイズをバイト単位で示します。

piecep (IN/OUT)

コールバック (アプリケーション) から Oracle へ、次のようにピース値を戻します。

- **IN** — 値は OCI_ONE_PIECE または OCI_NEXT_PIECE です。
- **OUT** — IN の値に応じて異なります。

IN 値が OCI_ONE_PIECE の場合、OUT 値は OCI_ONE_PIECE または OCI_FIRST_PIECE です。

IN 値が OCI_NEXT_PIECE の場合、OUT 値は OCI_NEXT_PIECE または OCI_LAST_PIECE です。

indpp (IN)

インジケータ変数ポインタです。

rcodep (IN)

リターン・コード変数ポインタです。

コメント

このコールは、OCIDefineByPos() のコールで OCI_DYNAMIC_FETCH モードが選択されている場合、必要な追加属性を設定するために使用されます。OCI_DYNAMIC_FETCH モードが選択されているときに OCIDefineDynamic() のコールがスキップされた場合、アプリケーションでは、OCI コール ([OCIStmtGetPieceInfo\(\)](#) および [OCIStmtSetPieceInfo\(\)](#)) を使用してデータ・ピース単位がフェッチされます。OCI_DYNAMIC_FETCH モードの詳細は、5-44 ページの「[ランタイム・データ割当てとピース単位操作](#)」を参照してください。

注意： `OCIEnvNlsCreate()` を使用して環境ハンドルを作成すると、バインド・ハンドルおよび定義ハンドルの実際の長さと戻される長さは、常にバイト単位になります。

関連関数

[OCIDefineByPos\(\)](#)

関連項目：『Oracle9i アプリケーション開発者ガイド - 基礎編』のセキュリティ・ポリシーの確立に関する章を参照してください。

OCIDefineObject()

用途

名前付きデータ型または REF の定義のために必要な追加属性を設定します。

構文

```
sword OCIDefineObject ( OCIDefine      *defnp,
                        OCIError       *errhp,
                        CONST OCIType  *type,
                        dvoid          **pgvpp,
                        ub4            *pvszsp,
                        dvoid          **indpp,
                        ub4            *indszp );
```

パラメータ

defnp (IN/OUT)

以前 OCIDefineByPos() のコールで割り当てられた定義ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet() に渡すエラー・ハンドルです。

type (IN) [optional]

プログラム変数のタイプを示す型記述子オブジェクト (TDO) を指し示します。型 SQLT_NTY のプログラム変数にのみ使用されます。このパラメータはオプションであり、使用しない場合は NULL として渡せます。

pgvpp (IN/OUT)

プログラム変数バッファのポインタを指し示します。配列では、*pgvpp* はポインタ配列を示します。フェッチした名前付きデータ型インスタンス用のメモリーは、オブジェクト・キャッシュ内に動的に割り当てられます。すべての値を受け取ったフェッチの終了時に、*pgvpp* は、新たに割り当てられた名前付きデータ型インスタンスのポインタを示します。名前付きデータ型インスタンスが不要になった際には、アプリケーションから OCIObjectFree() をコールして、これらの割当てを解除する必要があります。

注意： アプリケーションでバッファを暗黙的にキャッシュに割り当てるには、**pgvpp* を NULL として渡す必要があります。

pvszsp (IN/OUT)

プログラム変数のサイズを指し示します。配列では、**ub4** 配列です。

indpp (IN/OUT)

パラレル・インジケータ構造体を含むプログラム変数バッファのポインタを指し示します。配列では、ポインタ配列を示します。インジケータ構造体を格納するためのメモリーがオブジェクト・キャッシュ内に割り当てられます。すべての値を受け取ったフェッチの終了時には、*indpp* は、新たに割り当てられたインジケータ構造体のポインタを示しています。

indszp (IN/OUT)

インジケータ構造体プログラム変数のサイズを指し示します。配列では、**ub4** 配列です。

コメント

この関数は、初期定義情報を設定する `OCIDefineByPos()` の後続コールです。このコールは、名前付きデータ型定義のために必要な追加属性を設定します。OCI 環境が非オブジェクト・モードで初期化されているときにこの関数をコールした場合は、エラーが戻ります。

この関数は、定義される名前付きデータ型用にデータ型 **OCITYPE** の型記述子オブジェクト (TDO) をパラメータとして取ります。TDO は、`OCIDescribeAny()` をコールして取り出すことができます。

関連項目： OCI プロセス環境の初期化の詳細は、15-18 ページの「`OCIInitialize()`」の説明を参照してください。

関連関数

`OCIDefineByPos()`

OCIDescribeAny()

用途

既存のスキーマ・オブジェクトおよびサブスキーマ・オブジェクトを記述します。

構文

```
sword OCIDescribeAny ( OCISvcCtx      *svchp,
                      OCIError       *errhp,
                      dvoid          *objptr,
                      ub4            objptr_len,
                      ub1            objptr_typ,
                      ub1            info_level,
                      ub1            objtyp,
                      OCIDescribe   *dschp );
```

パラメータ

svchp (IN)

サービス・コンテキスト・ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

objptr (IN)

このパラメータは次のいずれかです。

1. 記述されるオブジェクト名を含む文字列。OCI_UTF16 環境でエンコーディングされた UTF16 であることが必要です。
2. TDO に対する REF のポインタ (型用)。
3. TDO のポインタ (型用)。

前述の各値は、*objptr_typ* に適切な値を渡すことで区別されます。このパラメータは非 NULL にしてください。

1 の場合、オブジェクト名を含む文字列は、*scott.emp.empno@mydb* のように、*name1[.name2 ...][@linkname]* の書式にしてください。データベース・リンクは、Oracle8i 以上のデータベースでのみ使用できます。オブジェクト名は、次の SQL ルールによって解釈されます。

- *name1* が入力され *objtyp* が OCI_PTYPE_SCHEMA の場合にのみ、名前は、指定されたスキーマを参照します。Oracle データベースは、Oracle8i 以上である必要があります。
- *name1* が入力され、*objtyp* が OCI_PTYPE_DATABASE の場合にのみ、名前は、指定されたデータベースを参照します。*database_name@db_link_name* を使用してリ

モート・データベースを記述する場合は、リモート Oracle データベースは Oracle8i 以上である必要があります。

- *name1* が入力され、*objtyp* が OCI_PTYPE_SCHEMA または OCI_PTYPE_DATABASE 以外の場合にのみ、名前は、カレント・ユーザーのカレント・スキーマ内の名前付きオブジェクト（表 / ビュー / プロシージャ / ファンクション / パッケージ / 型 / シノニム / 順序）を参照します。Oracle7 Server に接続したときは、有効な型はプロシージャおよびファンクションのみです。
- *name1.name2.name3 ...* と入力した場合、オブジェクト名は、*name1* というスキーマ内のスキーマ・オブジェクトまたはサブスキーマ・オブジェクトを参照します。たとえば、文字列 *scott.emp.deptno* では、*scott* はスキーマの名前、*emp* はスキーマ内の表の名前で、*deptno* は表内の列の名前です。

objnm_len (IN)

objptr が指し示す名前文字列の長さです。名前が渡される場合は、0（ゼロ）以外にしてください。*objptr* が TDO またはその REF へのポインタの場合は、0（ゼロ）でもかまいません。

objptr_typ (IN)

objptr に渡されるオブジェクトのタイプです。次の値が有効です。

- OCI_OTYPE_NAME、*objptr* がスキーマ・オブジェクトの名前を示す場合
- OCI_OTYPE_REF、*objptr* が TDO に対する REF へのポインタの場合
- OCI_OTYPE_PTR、*objptr* が TDO のポインタの場合

info_level (IN)

今後の拡張要素のために確保されています。OCI_DEFAULT を渡します。

objtyp (IN)

記述されるスキーマ・オブジェクトのタイプです。次の値が有効です。

- OCI_PTYPE_TABLE（表用）
- OCI_PTYPE_VIEW（ビュー用）
- OCI_PTYPE_PROC（プロシージャ用）
- OCI_PTYPE_FUNC（ファンクション用）
- OCI_PTYPE_PKG（パッケージ用）
- OCI_PTYPE_TYPE（型用）
- OCI_PTYPE_SYN（シノニム用）
- OCI_PTYPE_SEQ（順序用）
- OCI_PTYPE_SCHEMA（スキーマ用）

- OCI_PTYPE_DATABASE (データベース用)
- OCI_PTYPE_UNK (不明なスキーマ・オブジェクト用)

dschp (IN/OUT)

コール後のオブジェクトに関する記述情報とともに移入されている記述ハンドルです。
NULL 以外にしてください。

コメント

このコールは、表、ビュー、シノニム、プロシージャ、ファンクション、パッケージ、順序、型、スキーマおよびデータベースなどの既存のスキーマ・オブジェクトを記述する汎用的な記述コールです。このコールでは、表内の列などのサブスキーマ・オブジェクトも記述できます。このコールは、OCIAttrGet () コールを使用して取得できるオブジェクト固有の属性を記述ハンドルに移入します。

記述ハンドルに対する OCIParamGet () は、指定位置のパラメータ記述子を戻します。パラメータ位置は 1 から開始します。パラメータ記述子に対して OCIAttrGet () をコールすると、ストアド・プロシージャまたはファンクション・パラメータの特定の属性、あるいは表の列記述子が戻されます。OCIDescribeAny () によってスキーマ・オブジェクト記述全体がクライアント側にキャッシュされているため、これらの後続コールは、サーバーへのラウンドトリップを別に行う必要がありません。記述ハンドルに対する OCIAttrGet () は、位置の総数も戻します。

記述ハンドルに対して OCI_ATTR_DESC_PUBLIC 属性が設定されている場合、カレント・スキーマに名前付きオブジェクトが存在せず *name1* のみが指定されているときは、このオブジェクトはパブリック・シノニムとして参照されます。

関連項目： 記述操作の詳細は、第 6 章「スキーマ・メタデータの記述」を参照してください。

関連関数

OCIAttrGet ()、OCIParamGet ()

OCIStmtGetBindInfo()

用途

バインドおよびインジケータ変数名を取得します。

構文

```
sword OCIStmtGetBindInfo ( OCIStmt      *stmtp,
                           OCIError     *errhp,
                           ub4          size,
                           ub4          startloc,
                           sb4          *found,
                           text         *bvnp[],
                           ub1          bvnl[],
                           text         *invp[],
                           ub1          inpl[],
                           ub1          dupl[],
                           OCIBind      *hndl[] );
```

パラメータ

stmtp (IN)

OCIStmtPrepare() によって準備される文ハンドルです。

errhp (IN)

エラー発生時の診断情報のために OCIErrorGet() に渡すエラー・ハンドルです。

size (IN)

各配列の要素数です。

startloc (IN)

バインド情報の取得を開始するバインド変数の位置です。

found (IN)

abs (*found*) は、開始位置にかかわらず、文のバインド変数の総数を指定します。戻されたバインド変数の数が用意されたサイズよりも少ない場合は正数に、そうでない場合は負数になります。

bvnp (OUT)

バインド変数名を保持するポインタの配列です。環境設定が OCI_UTF16 モードの場合は、UTF-16 であることが必要です。

bvnl (OUT)

各 *bvnp* 要素の長さを保持する配列です。長さはバイト単位です。

invp (OUT)

インジケータ変数名を保持するポインタの配列です。環境設定が OCI_UTF16 モードの場合は、UTF-16 であることが必要です。

inpl (OUT)

各 *invp* 要素の長さを保持するポインタの配列です。バイト単位です。

dupl (OUT)

バインド位置が他と重複しているかどうかにより、その要素値が 0 または 1 になる配列です。

hndl (OUT)

バインド位置のバインドが完了している場合にバインド・ハンドルを戻す配列です。重複していると、ハンドルは戻りません。

コメント

このコールは、文が準備された後にバインド変数に関する情報を戻します。バインド名、インジケータ名および重複バインドかどうかなどの情報が含まれます。このコールでは、対応付けられたバインド・ハンドルがあれば、それも戻します。バインド変数の区別ごとの数ではなく、総数を *found* パラメータに設定します。

SELECT INTO リスト変数はバインドとはみなされていないため、この関数には含まれません。

このコールの前に、`OCIStmtPrepare()` のコールを使用して文を準備する必要があります。文ハンドルのエンコーディング設定によって、Unicode 文字列が取り出されるかどうかが決まります。

このコールは、ローカルで処理されます。

関連関数

[OCIStmtPrepare\(\)](#)

その他の OCI リレーショナル関数

この章では、前の章に続いて、OCI リレーショナル関数の詳細を説明します。各関数コールの詳細を説明するとともに、使用しているアプリケーションで OCI 関数をコールする方法についても説明します。

関連項目： コード例は、Oracle のインストールに含まれているデモ・プログラムを参照してください。追加情報については、[付録 B「OCI デモ・プログラム」](#)を参照してください。

この章は、次の項目で構成されています。

- [その他のリレーショナル関数の概要](#)
- [文関数](#)
- [LOB 関数](#)
- [アドバンスド・キューイング関数およびパブリッシュ・サブスクライブ関数](#)
- [ダイレクト・パス・ロード関数](#)
- [スレッド管理関数](#)
- [トランザクション関数](#)
- [その他の関数](#)

その他のリレーショナル関数の概要

この章では、OCI リレーショナル関数コールの詳細について説明します。この章は前の章からの続きです。オブジェクトを操作するための関数コールについては、この後の3つの章で説明します。

関連項目： リターン・コードおよびエラー処理の詳細は、2-31 ページの「[エラー処理](#)」を参照してください。

関数の構文

各関数について、次の情報が記載されています。

用途

この関数によって実行されるアクションを簡単に説明します。

構文

関数の宣言。

パラメータ

この関数の各パラメータの説明。これにはパラメータのモードが含まれます。パラメータのモードには、次の3つの値があります。

モード	説明
IN	OCI にデータを渡すパラメータ
OUT	このコールで OCI からデータを受け取るパラメータ
IN/OUT	このコールでデータを渡し、このコールまたは後続のコールからの戻りでデータを受け取るパラメータ

コメント

この関数に関する詳細情報。関数の使用上の制約やアプリケーション内でこの関数を使用するときに役に立つ情報が記載されています。

例

説明の対象となっている関数コールの使用方法を例示するコード（全体または一部）。すべての関数に例が記載されているわけではありません。

関連関数

関連する関数コールのリスト。

OCI 関数のコール

OCI のこれまでのバージョンとは異なり、Oracle8i OCI 以降では、ヌル文字で終了する文字列の文字列長パラメータに -1 を渡せません。

文字列長をパラメータとして渡すときは、ヌル終端文字バイトを組み込まないでください。OCI では、文字列はヌル文字で終了するとはみなされません。

OCI パラメータのバッファ長はバイト単位です。ただし、一部の LOB コールの `amount` パラメータは文字単位です。

LOB 関数用のサーバー・ラウンドトリップ

個々の OCI LOB 関数に必要なサーバー・ラウンドトリップ回数については、[付録 C 「OCI 関数のサーバー・ラウンドトリップ」](#) の表を参照してください。

文関数

この項では、文関数について説明します。

表 16-1 文関数

関数	用途
OCIStmtExecute() (16-5 ページ)	実行する文をサーバーに送信します。
OCIStmtFetch() (16-8 ページ)	問合せから行をフェッチします (使用できません)。
OCIStmtFetch2() (16-10 ページ)	問合せから行をフェッチします。
OCIStmtGetPieceInfo() (16-13 ページ)	ピース単位操作のためのピース情報を取得します。
OCIStmtPrepare() (16-15 ページ)	実行する SQL 文または PL/SQL 文を準備します。
OCIStmtPrepare2() (16-17 ページ)	実行する SQL 文または PL/SQL 文を準備します。
OCIStmtRelease() (16-19 ページ)	文ハンドルを解放します。
OCIStmtSetPieceInfo() (16-20 ページ)	ピース単位操作のためのピース情報を設定します。

OCIStmtExecute()

用途

このコールは、アプリケーション要求をサーバーに対応付けます。

構文

```
sword OCIStmtExecute ( OCISvcCtx          *svchp,
                       OCIStmt           *stmtp,
                       OCIError          *errhp,
                       ub4                iters,
                       ub4                rowoff,
                       CONST OCISnapshot *snap_in,
                       OCISnapshot       *snap_out,
                       ub4                mode );
```

パラメータ

svchp (IN/OUT)

サービス・コンテキスト・ハンドルです。

stmtp (IN/OUT)

文ハンドルです。サーバーで実行される文および対応付けられたデータを定義します。
svchp が Oracle7 Server を指し示しているときに、リリース 8.x 以上でのみサポートされるデータ型のバインドを持つ文ハンドルを渡しても無効になります。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

iters (IN)

SELECT 文以外の場合、この文が実行される回数は、*iters* - *rowoff* の場合と同じになります。

SELECT 文では、*iters* が 0（ゼロ）以外の場合は、文ハンドルに対する定義を行う必要があります。実行すると、*iters* が事前定義バッファにフェッチされ、プリフェッチ行カウントに従ってさらに行がプリフェッチされます。SELECT 文によって取り出される行数が不明の場合は、*iters* を 0（ゼロ）に設定します。

この関数は、SELECT 文以外に対して *iters*=0 の場合は、エラーを戻します。

注意： 配列 DML 操作の場合は、*iters* <= 32767 を設定することで、より高いパフォーマンスが得られます。

rowoff (IN)

この複数行実行に関連する配列バインドのデータが始まる開始索引です。

snap_in (IN)

このパラメータはオプションです。指定する場合は、OCI_DTYPE_SNAP 型のスナップショット記述子を指示する必要があります。この記述子の内容は、直前のコールの *snap_out* パラメータから取得する必要があります。この記述子は、SQL が SELECT でない場合は無視されます。この機能を使用すると、Oracle への複数サービス・コンテキストによって、データベースのコミット済みデータに関して同じ一貫性のあるスナップショットを参照できます。ただし、あるコンテキストでコミットされていないデータは、同じスナップショットを使用しても別のコンテキストで参照できません。

snap_out (OUT)

このパラメータはオプションです。指定する場合は、OCI_DTYPE_SNAP 型の記述子を指示する必要があります。この記述子には、現行の Oracle のシステム変更番号が暗号化されて格納されており、後続の *OCIStmtExecute()* コールの *snap_in* への入力値として使用できます。この記述子は必要以上に長く使用しないでください。「スナップショットが古すぎます」というエラーが発生します。

mode (IN)

次のモードが有効です。

- OCI_BATCH_ERRORS — このモードの詳細は、4-9 ページの「*OCIStmtExecute()* 用のバッチ・エラー・モード」を参照してください。
- OCI_COMMIT_ON_SUCCESS — このモードで文を実行した場合、実行が正常に終了すると、実行後にカレント・トランザクションがコミットされます。
- OCI_DEFAULT — このモードで *OCIStmtExecute()* をコールすると文が実行されます。また、選択リストに関する記述情報が暗黙的に戻されます。
- OCI_DESCRIBE_ONLY — このモードは、実行前に問合せを記述するユーザー用です。*OCIStmtExecute()* をこのモードでコールすると、文は実行されませんが、選択リスト記述は戻されます。パフォーマンスを最大にするために、アプリケーションではデフォルト・モードで文を実行し、実行に伴う暗黙的な記述を使用することをお勧めします。
- OCI_EXACT_FETCH — アプリケーションでフェッチされる行数があらかじめ正確にわかっているときに使用します。このモードにより、Oracle リリース 8.x 以上のモードのプリフェッチがオフになります。また、実行コールの前に定義されている必要があります。このモードを使用すると要求した行がフェッチされた後カーソルが取り消されるため、サーバー側のリソース使用量を削減できます。
- OCI_PARSE_ONLY — このモードを使用すると、ユーザーは実行前に問合せを解析できます。このモードで実行すると問合せが解析され、SQL 内に解析エラーがある場合は、そのエラーが戻されます。このモードではサーバーへの追加ラウンドトリップが発生することに注意する必要があります。パフォーマンスを向上させるには、バンドル操作の一部として文を解析するデフォルト・モードで、文を実行することをお勧めします。

- `OCI_STMT_SCROLLABLE_READONLY` — 結果セットをスクロール可能に設定する場合は必須です。結果セットは更新できません。4-16 ページの「[結果のフェッチ](#)」を参照してください。他のモードとの併用はできません。

これらのモードは相互排他的ではなく、組み合わせて使用できます。ただし、`OCI_STMT_SCROLLABLE_READONLY` を除きます。

コメント

この関数は、プリコンパイルされた SQL 文を実行するために使用します。アプリケーションは、実行コールを使用して要求をサーバーに対応付けます。

SELECT 文が実行されると、選択リストの記述が応答として暗黙的に使用可能になります。この記述は、記述、フェッチおよび型変換定義用にクライアント側にバッファ処理されます。したがって、選択リストの記述は、実行後のみに行うのが最善の方法です。

関連項目： 4-12 ページ「[選択リスト項目の記述](#)」

SELECT 文の場合は、一部の結果も暗黙的に使用可能となります。実行終了の時点で行が受け取られ、バッファ処理されます。行数が少ない問合せでは、プリフェッチすることでフェッチの最後に達したときにサーバー内のメモリーが解放され、これによってメモリーの使用量が削減されるように最適化できます。プリフェッチする行数を結果セットごとに設定する属性設定コールが定義されています。

SELECT 文では、文ハンドルは、それが実行されたサービス・コンテキストに対する参照を実行終了時に暗黙的に保持しています。サービス・コンテキストの完全性は、ユーザーに保守の義務があります。暗黙的な参照は、文ハンドルが解放されるまたはフェッチが取り消される、あるいはフェッチ条件の最後に達するまで保持されます。

注意： `OCIStmtExecute()` のコール前に SELECT 文に対して出力変数を定義すると、`iters` によって指定された行数が定義済み出力バッファに直接フェッチされ、プリフェッチ件数と同じ数の追加行がプリフェッチされます。追加行がない場合、フェッチは `OCIStmtFetch()` をコールしないで完了します。

関連関数

[OCIStmtPrepare\(\)](#)

OCIStmtFetch()

用途

問合せから行をフェッチします。新しいフェッチ・コール `OCIStmtFetch2()` の使用をお勧めします。このコールは使用できません。

構文

```
sword OCIStmtFetch ( OCIStmt      *stmtp,  
                     OCIError     *errhp,  
                     ub4          nrows,  
                     ub2          orientation,  
                     ub4          mode );
```

パラメータ

stmtp (IN)

文（アプリケーション要求）ハンドルです。

errhp (IN)

エラー発生時の診断情報のために `OCIErrorGet()` に渡すエラー・ハンドルです。

nrows (IN)

現行の位置からフェッチされる行数です。

orientation (IN)

リリース 8.1.7 以下で使用可能な値は、`OCI_FETCH_NEXT`（デフォルト値）のみです。

mode (IN)

`OCI_DEFAULT` を渡します。

コメント

プリフェッチされた行で間に合う場合、フェッチ・コールはローカル・コールになります。ただしこれは、アプリケーションに対して透過的に行われます。

LOB 列が読み込まれる場合、LOB ロケータに対して実行される後続の LOB 操作のために、それらのロケータがフェッチされます。LONG 列の場合、プリフェッチはオフになります。

この関数は、次のエラーのいずれかが発生すると EOF の `OCI_NO_DATA` および `OCI_SUCCESS_WITH_INFO` を戻します。

- ORA-24344 「正常に終了しましたが、コンパイル・エラーがあります。」
- ORA-24345 「切捨てまたは NULL フェッチ・エラーが発生しました。」

- ORA-24347「警告：グループ関数に NULL 列があります。」

`nrows` パラメータに 0（ゼロ）を設定して `OCIStmtFetch()` をコールした場合は、カーソルが取り消されます。

`OCI_ATTR_ROWS_FETCHED` を使用して、最後のフェッチ・コールでユーザーのバッファに正常にフェッチされた行数を検索します。

関連関数

[OCIStmtExecute\(\)](#)

OCIStmtFetch2()

用途

このコールは、(スクロール可能な) 結果セットから行をフェッチします。使用できない `OCIStmtFetch()` のかわりに、このフェッチ・コールを使用することをお勧めします。

構文

```
sword OCIStmtFetch2 ( OCIStmt      *stmthp,  
                      OCIError     *errhp,  
                      ub4           nrows,  
                      ub2           orientation,  
                      sb4           fetchOffset,  
                      ub4           mode );
```

パラメータ

stmthp (IN/OUT)

これは (スクロール可能な) 結果セットの文ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために `OCIErrorGet()` に渡すエラー・ハンドルです。

nrows (IN)

現行の位置からフェッチされる行数です。

orientation (IN)

受け入れ可能な値は、次のとおりです。

- `OCI_DEFAULT` — `OCI_FETCH_NEXT` と同じ結果が得られます。
- `OCI_FETCH_CURRENT` — 現在行を取得します。
- `OCI_FETCH_NEXT` — 現行位置の次の行を取得します。これは、デフォルトです (`OCI_DEFAULT` と同じ結果が得られます)。スクロール不可な文ハンドルに使用します。
- `OCI_FETCH_FIRST` — 結果セットの最初の行を取得します。
- `OCI_FETCH_LAST` — 結果セットの最後の行を取得します。
- `OCI_FETCH_PRIOR` — 結果セットの現在行の前の行を取得します。
- `OCI_FETCH_ABSOLUTE` — 絶対的な位置指定を使用して結果セットの行番号 (`fetchOffset` パラメータで指定) をフェッチします。
- `OCI_FETCH_RELATIVE` — 相対的な位置指定を使用して結果セットの行番号 (`fetchOffset` パラメータで指定) をフェッチします。

fetchOffset (IN)

現在行の位置を変更するために **orientation** パラメータと併用するオフセットです。

mode (IN)

OCI_DEFAULT を渡します。

コメント

フェッチ・コールは、OCIStmtFetch() コールに *fetchOffset* パラメータを追加した場合と同じように機能します。スクロール可能かどうかに関係なく、すべての文ハンドルに使用できます。スクロール不可な文ハンドルの場合、唯一の受け入れ可能な **orientation** 値は、OCI_FETCH_NEXT です。*fetchOffset* パラメータは無視されます。

新しいアプリケーションには、この新しいコール OCIStmtFetch2() の使用をお勧めします。

orientation が OCI_FETCH_RELATIVE に設定されている *fetchOffset* は、次のすべてのコールと等価です。

- *fetchOffset* の値が 0 (ゼロ) の OCI_FETCH_CURRENT。
- *fetchOffset* の値が 1 の OCI_FETCH_NEXT。
- *fetchOffset* の値が -1 の OCI_FETCH_PRIOR。

OCI_ATTR_ROW_COUNT には、フェッチされた最上位の行の絶対値が含まれます。

OCI_FETCH_ABSOLUTE と OCI_FETCH_RELATIVE を除くすべての **orientation** モードでは、*fetchOffset* 値は無視されます。

このコールを使用すると、OCI_FETCH_LAST を使用してから、OCI_ATTR_CURRENT_POSITION に対して OCIAttrGet() をコールすることで、結果セット内の行数を検索することもできます。ただし、このコールの応答時間はかなり長くなります。

リターン・コードは、OCIStmtFetch() の場合と同じです。ただし、スクロール可能な文ハンドルのフェッチ（または実行）のたびに、リターン・コード OCI_NO_DATA を含む OER (1403) が戻されます。また、アプリケーションが要求するすべての行がフェッチされるわけではありません。

サーバー側のリソースをこのスクロール・カーソル用に解放するには、スクロール可能な文ハンドルを明示的に取り消すか（つまり、0 (ゼロ) 行でフェッチする）、または解放する必要があります。スクロール不可な文ハンドルは、OER (1403) を受け取ると暗黙的に取り消されます。

OCI_ATTR_ROWS_FETCHED を使用して、最後のフェッチ・コールでユーザーのバッファに正常にフェッチされた行数を検索します。

関連項目： このトピックの詳細は、4-17 ページの「[スクロール・カーソル](#)」を参照してください。

関連関数

[OCIStmtExecute\(\)](#)、[OCIBindByPos\(\)](#)

OCIStmtGetPieceInfo()

用途

ピース単位操作作用のピース情報を戻します。

構文

```
sword OCIStmtGetPieceInfo( CONST OCIStmt  *stmtp,  
                           OCIError      *errhp,  
                           dvoid         **hdlpp,  
                           ub4           *typep,  
                           ub1           *in_outp,  
                           ub4           *iterp,  
                           ub4           *idxp,  
                           ub1           *piecep );
```

パラメータ

stmtp (IN)

戻された OCI_NEED_DATA が実行される際の文です。

errhp (OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

hdlpp (OUT)

バインド、バインドの定義ハンドル、またはランタイム・データが要求または提供されている定義のいずれかへのポインタを戻します。

typep (OUT)

hdlpp が指し示すハンドルのタイプです。タイプには、OCI_HTYPE_BIND（バインド・ハンドル用）または OCI_HTYPE_DEFINE（定義ハンドル用）があります。

in_outp (OUT)

IN バインド値に対してデータが必要な場合、OCI_PARAM_IN を戻します。データが OUT バインド変数または定義位置値として取得できる場合は OCI_PARAM_OUT が戻ります。

iterp (OUT)

複数行操作の行数を戻します。

idxp (OUT)

PL/SQL 配列バインド操作の配列要素の索引です。

piecep (OUT)

OCI_ONE_PIECE、OCI_FIRST_PIECE、OCI_NEXT_PIECE または OCI_LAST_PIECE のいずれかの事前定義値を戻します。

コメント

実行コールまたはフェッチ・コールから `OCI_NEED_DATA` が戻され、動的なバインドまたは定義の値、あるいはピースが取得または戻されると、`OCIStmtGetPieceInfo()` から、バインド・ハンドルまたは定義ハンドル、反復、索引番号、ピース情報などの関連情報が戻されます。

関連項目： `OCIStmtGetPieceInfo()` の使用方法の詳細は、5-44 ページの「ランタイム・データ割当てとピース単位操作」を参照してください。

関連関数

`OCIAttrGet()`、`OCIAttrSet()`、`OCIStmtExecute()`、`OCIStmtFetch()`、`OCIStmtSetPieceInfo()`

OCIStmtPrepare()

用途

このコールは、実行する SQL 文または PL/SQL 文を準備します。

構文

```
sword OCIStmtPrepare ( OCIStmt      *stmtp,  
                       OCIError     *errhp,  
                       CONST text    *stmt,  
                       ub4           stmt_len,  
                       ub4           language,  
                       ub4           mode );
```

パラメータ

stmtp (IN)

実行対象の文に関連付けられた文ハンドルです。デフォルトでは、導出元の環境ハンドルのエンコーディング設定が含まれています。文を UTF-16 エンコーディングで準備できるのは、UTF-16 環境のみです。

errhp (IN)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

stmt (IN)

実行される SQL 文または PL/SQL 文です。ヌル文字で終了する文字列にしてください。つまり、最後の文字は、エンコーディングによっては NULL バイトの数値です。環境ハンドルが OCI_UTF16 モードで作成されている場合は、文も UTF-16 であることが必要です。

パラメータを必ず (**text ***) にキャストしてください。文が UTF-16 で準備されると、バインド・バッファと定義バッファのキャラクタ・セットは、UTF-16 にデフォルト設定されます。文のテキストへのポインタは、その文が実行されたり、その文からデータがフェッチされる間は、使用可能であることが必要です。

stmt_len (IN)

文の長さです。エンコーディングによって、文字数またはバイト数の単位になります。0 (ゼロ) 以外にする必要があります。

language (IN)

V7 構文またはネイティブ構文を指定します。可能な値は次のとおりです。

- OCI_V7_SYNTAX – V7 Oracle 解析構文。
- OCI_NTV_SYNTAX – サーバーのバージョンに依存する構文。

mode (IN)

OCIEnvCreate() コールの *mode* に類似しています。ただし、このコールは必然的に継承されたモード設定を上書きできるため、優先順位が高くなります。

次の値があります。

- OCI_DEFAULT – デフォルト・モード。文ハンドル *stmt* は、親の環境ハンドルに指定されている内容を使用します。
- OCI_NO_SHARING – SQL 文の共有モードを使用禁止にします。2-22 ページの「[共有データ・モード](#)」を参照してください。

コメント

このコールは、OCI アプリケーションで実行する SQL 文または PL/SQL 文を準備するために使用します。OCIStmtPrepare() コールは、アプリケーション要求を定義します。

mode パラメータは、文の内容が UTF-16 でエンコーディングされているかどうかを判断します。文の長さは、コードポイント数またはバイト数で、エンコーディングによって異なります。

文ハンドルは、親の環境ハンドルからエンコーディング設定を継承しますが、このコールの *mode* によって、文ハンドル自体のエンコーディング設定も変更できます。

後続のバインド・コールで初期化されるこの文のデータ値は、この文ハンドルの設定をデフォルトとして使用するバインド・ハンドル内に格納されます。

このコールは、この文ハンドルと特定のサーバー間の対応付けは作成しません。

関連項目： このコールの使用方法の詳細は、4-4 ページの「[文の準備](#)」を参照してください。

関連関数

[OCIAttrGet\(\)](#)、[OCIStmtExecute\(\)](#)

OCIStmtPrepare2()

用途

このコールは、実行する SQL 文または PL/SQL 文を準備します。有効な文キャッシュがある場合は、それを使用することもできます。

構文

```
sword OCIStmtPrepare2 ( OCISvcCtx      *svchp,
                        OCIStmt        **stmthp,
                        OCIError       *errhp,
                        CONST OraText  *stmttext,
                        ub4            stmt_len,
                        CONST OraText  *key,
                        ub4            keylen,
                        ub4            language,
                        ub4            mode );
```

パラメータ

svchp (IN)

文に関連付けるサービス・コンテキストです。

errhp (IN)

診断のためのエラー・ハンドルへのポインタです。

stmthp (OUT)

戻される文ハンドルへのポインタです。

stmttext (IN)

文のテキストです。stmttext のセマンティックは、OCIStmtPrepare のセマンティックと同じです（つまり、文字列はヌル文字で終了）。

stmt_len (IN)

文のテキストの長さです。

key (IN)

文キャッシュの場合のみ指定します。キャッシュ内の戻された文に対するキーです。このパラメータは、その後の OCIStmtPrepare2 () のコールに使用できます。この場合、文のテキストや関連するパラメータの指定は不要です。このキーが指定されると、文のテキストおよび他のパラメータは無視され、このキーのみに基づいて検索が行われます。

keylen (IN)

文キャッシュの場合のみ指定します。キーの長さです。

language (IN)

V7 構文またはネイティブ構文を指定します。可能な値は次のとおりです。

- OCI_V7_SYNTAX – V7 Oracle 解析構文。
- OCI_NTV_SYNTAX – サーバーのバージョンに依存する構文。

mode (IN)

この関数では、文キャッシュを使用することも使用しないことも可能です。使用するかどうかは、接続またはセッション・プールの作成時に決まります。セッションに対してキャッシュが使用可能な場合はセッション内のすべての文がキャッシュ可能で、キャッシュが使用可能でない場合はすべての文がキャッシュされません。

次のモードが有効です。

- OCI_DEFAULT – キャッシュしない場合は、これが唯一有効な設定です。文がキャッシュ内に見つからなかった場合は、文ハンドルが新しく割り当てられ、実行用の文ハンドルが準備されます。文がキャッシュで見つかった場合は、状況に従って次のようになります。
 - (a) テキストのみが指定された場合：新しい文が割り当てられて準備され、戻されます。タグは NULL になります。OCI_SUCCESS が戻ります。
 - (b) タグのみが指定された場合：stmthp は NULL になります。OCI_ERROR が戻ります。
 - (c) テキストとキーの両方が指定された場合：新しい文が割り当てられて準備され、戻されます。タグは NULL になります。戻された文はタグが NULL である点で要求した文とは異なるため、OCI_SUCCESS_WITH_INFO が戻ります。
- OCI_STMTCACHE_SEARCH_ONLY – このモードで、文が見つからなかった（文ハンドルが NULL で戻された）場合は、さらに処置が必要です。文が見つかった場合は、OCI_SUCCESS が戻ります。見つからない場合は、OCI_ERROR が戻ります。

関連関数

[OCIStmtRelease\(\)](#)

OCIStmtRelease()

用途

OCIStmtPrepare2() のコールで取得した文ハンドルを解放します。

構文

```
sword OCIStmtRelease ( OCIStmt          *stmthp,  
                        OCIError         *errhp,  
                        CONST OraText    *key,  
                        ub4               keylen,  
                        ub4               mode );
```

パラメータ

stmthp (IN/OUT)

OCIStmtPrepare2() によって戻される文ハンドルです。

errhp (IN)

診断に使用するエラー・ハンドルです。

key (IN)

文キャッシュの場合のみ有効です。キャッシュ内の文に関連付けられているキーです。OCIStmtPrepare2() によって戻されるキー、または新しいキーを指定できます。NULL のキーが渡された場合、文はタグ付けされません。

keylen (IN)

文キャッシュの場合のみ有効です。キーの長さです。

mode (IN)

次のモードが有効です。

- OCI_DEFAULT
- OCI_STMTCACHE_DELETE — 文キャッシュの場合のみ有効です。文は、それ以上キャッシュに保持されません。

関連関数

[OCIStmtPrepare2\(\)](#)

OCIStmtSetPieceInfo()

用途

ピース単位操作のピース情報を設定します。

構文

```
sword OCIStmtSetPieceInfo ( dvoid          *hndlp,
                             ub4           type,
                             OCIError      *errhp,
                             CONST dvoid   *bufp,
                             ub4           alenp,
                             ub1           piece,
                             CONST dvoid   *indp,
                             ub2           *rcodep );
```

パラメータ

hndlp (IN/OUT)

バインド・ハンドルまたは定義ハンドルです。

type (IN)

ハンドルのタイプです。

errhp (OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

bufp (IN/OUT)

IN バインド変数の場合はデータ値またはピースを含む記憶域のポインタです。それ以外の
場合、*bufp* は OUT バインドおよび定義変数に対してピースまたはデータ値を取得するた
めの記憶域のポインタです。名前付きデータ型または REF の場合は、そのオブジェクトまたは
REF へのポインタが戻ります。

alenp (IN/OUT)

ピースまたは値の長さです。

piece (IN)

ピース・パラメータです。有効な値は次のとおりです。

- OCI_ONE_PIECE
- OCI_FIRST_PIECE
- OCI_NEXT_PIECE
- OCI_LAST_PIECE

このパラメータは、IN バインド変数でのみ使用されます。

indp (IN/OUT)

インジケータです。**sb2** 値へのポインタまたは名前付きデータ型 (SQLT_NTY) と REF (SQLT_REF) 用のインジケータ構造体へのポインタです。つまり **indp* は、データ型に応じて、**sb2** または **dvoid *** になります。

rcodep (IN/OUT)

リターン・コードです。

コメント

実行コールにより、動的 IN/OUT バインド値またはピースを取得する OCI_NEED_DATA が戻されると、OCIStmtSetPieceInfo() により、バッファ、長さ、現在処理中のピース、インジケータおよびこの列のリターン・コードなどのピース情報が設定されます。

関連項目： OCIStmtSetPieceInfo() の使用方法の詳細は、5-44 ページの「ランタイム・データ割当てとピース単位操作」を参照してください。

関連関数

OCIAttrGet(), OCIAttrSet(), OCIStmtExecute(), OCIStmtFetch(), OCIStmtGetPieceInfo()

LOB 関数

この項では、LOB 関数について説明します。

表 16-2 LOB 関数

関数	用途
OCIDurationBegin() (16-24 ページ)	一時 LOB のためのユーザー期間を開始します。
OCIDurationEnd() (16-25 ページ)	一時 LOB のためのユーザー期間を終了します。
OCILobAppend() (16-26 ページ)	1 つの LOB を別の LOB の後に追加します。
OCILobAssign() (16-28 ページ)	1 つの LOB ロケータを別の LOB ロケータに割り当てます。
OCILobCharSetForm() (16-30 ページ)	LOB ロケータからキャラクタ・セットを取得します。
OCILobCharSetId() (16-31 ページ)	LOB ロケータからキャラクタ・セット ID を取得します。
OCILobClose() (16-32 ページ)	オープンされている LOB をクローズします。
OCILobCopy() (16-34 ページ)	LOB の一部または全部を別の LOB にコピーします。
OCILobCreateTemporary() (16-36 ページ)	一時 LOB を作成します。
OCILobDisableBuffering() (16-38 ページ)	LOB バッファリングをオフにします。
OCILobEnableBuffering() (16-39 ページ)	LOB バッファリングをオンにします。
OCILobErase() (16-40 ページ)	LOB の一部を消去します。
OCILobFileClose() (16-42 ページ)	オープンされている FILE をクローズします。
OCILobFileCloseAll() (16-43 ページ)	オープンされているすべての FILE をクローズします。
OCILobFileExists() (16-44 ページ)	サーバー上のファイルの存在をチェックします。
OCILobFileNameGet() (16-45 ページ)	LOB ロケータからディレクトリ別名とファイル名を取得します。
OCILobFileIsOpen() (16-47 ページ)	サーバー上のファイルがこのロケータでオープンされているかどうかをチェックします。
OCILobFileOpen() (16-48 ページ)	FILE をオープンします。
OCILobFileNameSet() (16-49 ページ)	LOB ロケータにディレクトリ別名とファイル名を設定します。
OCILobFlushBuffer() (16-51 ページ)	LOB バッファをフラッシュします。
OCILobFreeTemporary() (16-53 ページ)	一時 LOB を解放します。
OCILobGetChunkSize() (16-54 ページ)	LOB のチャンク・サイズを取得します。
OCILobGetLength() (16-56 ページ)	LOB の長さを取得します。

表 16-2 LOB 関数（続き）

関数	用途
OCILobIsEqual() (16-57 ページ)	2 つの LOB ロケータが等しいか比較します。
OCILobIsOpen() (16-58 ページ)	LOB がオープンされているかどうかをチェックします。
OCILobIsTemporary() (16-60 ページ)	特定の LOB が一時 LOB かどうかを判断します。
OCILobLoadFromFile() (16-61 ページ)	FILE から LOB をロードします。
OCILobLocatorAssign() (16-63 ページ)	1 つの LOB ロケータを別の LOB ロケータに割り当てます。
OCILobLocatorIsInit() (16-65 ページ)	LOB ロケータが初期化されているかどうかをチェックします。
OCILobOpen() (16-67 ページ)	LOB をオープンします。
OCILobRead() (16-69 ページ)	LOB の一部を読み込みます。
OCILobTrim() (16-74 ページ)	LOB を切り捨てます。
OCILobWrite() (16-76 ページ)	LOB に書き込みます。
OCILobWriteAppend() (16-81 ページ)	LOB の末尾からデータを書き込みます。

OCI LOB コールのパラメータについて、次の点に注意してください。

- 固定幅のクライアント側キャラクタ・セットの場合、**offset** パラメータと **amount** パラメータは、CLOB と NCLOB では常に文字数、BLOB と BFILE では常にバイト数です。
- 可変幅のクライアント側キャラクタ・セットの場合は、一般に次のルールが適用されます。
 - **amount** (*amt*) パラメーター **amount** パラメータがサーバー側の LOB を参照する場合、その内容は文字数です。**amount** パラメータがクライアント側のバッファを参照する場合、その内容はバイト数です。詳細は、[OCILobGetLength\(\)](#)、[OCILobRead\(\)](#)、[OCILobWrite\(\)](#) などの各 LOB コールを参照してください。
 - **offset** (*offset*) パラメーター クライアント側のキャラクタ・セットが可変幅かどうかに関係なく、**offset** パラメータは、CLOB と NCLOB では常に文字数、BLOB と BFILE では常にバイト数です。
- LOB 操作では、多くの場合、クライアント側のキャラクタ・セットに関係なく、**amount** パラメータは CLOB と NCLOB の文字数です。これらの LOB 操作には [OCILobCopy\(\)](#)、[OCILobErase\(\)](#)、[OCILobGetLength\(\)](#)、[OCILobLoadFromFile\(\)](#) および [OCILobTrim\(\)](#) などがあります。これらのすべての操作では、サーバー上の LOB データの量を参照します。

ストリーム操作は、ピース単位での LOB の書込みや読込みを意味します。ストリームは、ポーリング・メカニズムを使用するか、またはユーザー定義コールバックを登録して実装できます。

OCIDurationBegin()

用途

一時 LOB のためのユーザー期間を開始します。

構文

```
sword OCIDurationBegin ( OCIEnv          *env,  
                          OCIError       *err,  
                          CONST OCISvcCtx *svc,  
                          OCIDuration    parent,  
                          OCIDuration    *duration );
```

パラメータ

env (IN/OUT)

NULL ポインタを渡します。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。`OCIErrorGet()` のコールによって診断情報を取得できます。

svc (IN)

OCI サービス・コンテキスト・ハンドルです。NULL 以外にしてください。

parent (IN)

親の継続時間の時間番号です。次のいずれかになります。

- `OCI_DURATION_STATEMENT`
- `OCI_DURATION_SESSION`

duration (OUT)

新しく作成されたユーザー期間固有の識別子です。

コメント

この関数によってユーザー期間が開始されます。Oracle8i OCI 以上では、一時 LOB の作成時にユーザー期間を使用できます。ユーザーは複数のアクティブなユーザー期間を同時に利用できます。ユーザー期間をネストする必要はありません。`dur` パラメータは、このコールによって作成された継続時間を識別するための一意の番号を戻すのに使用します。

関連項目： 7-18 ページ [「一時 LOB の継続時間」](#)

関連関数

[OCIDurationEnd\(\)](#)

OCIDurationEnd()

用途

一時 LOB のためのユーザー期間を終了します。

構文

```
sword OCIDurationEnd ( OCIEnv          *env,  
                      OCIError        *err,  
                      CONST OCISvcCtx  *svc,  
                      OCIDuration     duration,  
                      CONST OCISvcCtx  *svc );
```

パラメータ

env (IN/OUT)

NULL ポインタを渡します。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。 `OCIErrorGet()` のコールによって診断情報を取得できます。

duration (IN)

ユーザー期間を識別するための番号です。

svc (IN)

OCI サービス・コンテキストです（カートリッジ・サービスの場合は NULL として渡しますが、それ以外の場合は非 NULL として渡します）。

コメント

この関数によってユーザー期間が終了します。ユーザー期間に割り当てられていた一時 LOB が解放されます。

関連項目： 7-18 ページ「[一時 LOB の継続時間](#)」

関連関数

[OCIDurationBegin\(\)](#)

OCILobAppend()

用途

別の LOB の末尾に LOB 値を指定どおりに追加します。

構文

```
sword OCILobAppend ( OCISvcCtx      *svchp,
                     OCIError       *errhp,
                     OCILobLocator  *dst_locp,
                     OCILobLocator  *src_locp );
```

パラメータ

svchp (IN)

サービス・コンテキスト・ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

dst_locp (IN/OUT)

一意に宛先 LOB を参照する内部 LOB ロケータです。このロケータは、*svchp* に指定したサーバーから取得されたロケータにしてください。

src_locp (IN)

一意にソース LOB を参照する内部 LOB ロケータです。このロケータは、*svchp* に指定したサーバーから取得されたロケータにしてください。

コメント

別の LOB の末尾に LOB 値を指定どおりに追加します。データは、ソースから宛先の末尾にコピーされます。コピー元 LOB およびコピー先 LOB は、すでに存在している必要があります。宛先 LOB は、新たに書き込まれるデータにあわせて拡張されます。許される最大長 (4GB) を超えての宛先 LOB の拡張と NULL LOB のコピーはエラーになります。

ソースと宛先の LOB ロケータは同じ型にしてください (つまり、両方とも BLOB または両方とも CLOB)。LOB バッファリングは、ロケータのどちらの型に対しても使用可能にしないでください。この関数では、ソースまたは宛先としての FILE ロケータを受け入れません。

この LOB 操作をオープン・コールとクローズ・コールで囲む必要はありません。この操作を実行する前に LOB をオープンしていない場合、LOB 列のファンクション索引とドメイン索引はこのコール時に更新されます。この操作を実行する前に LOB をオープンしている場合は、トランザクションをコミットまたはロールバックする前に、その LOB をクローズする必要があります。内部 LOB をクローズすると、LOB 列のファンクション索引とドメイン索引が更新されます。

LOB 操作をオープン API とクローズ API で囲んでいない場合、ファンクション索引とドメイン索引は、LOB に書き込むたびに更新されます。これによって、パフォーマンスが低下する可能性があります。ファンクション索引とドメイン索引がある場合は、LOB への書き込み操作をオープン / クローズ文内に含めることをお勧めします。

関連関数

[OCILOBTrim\(\)](#)、[OCILOBWrite\(\)](#)、[OCILOBCopy\(\)](#)、[OCIErrorGet\(\)](#)、[OCILOBWriteAppend\(\)](#)

OCILobAssign()

用途

LOB/FILE ロケータを別のロケータに割り当てます。

構文

```
sword OCILobAssign ( OCIEEnv           *envhp,  
                    OCIError          *errhp,  
                    CONST OCILobLocator *src_locp,  
                    OCILobLocator      **dst_locpp );
```

パラメータ

envhp (IN/OUT)

OCI 環境ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrGet () に渡すエラー・ハンドルです。

src_locp (IN)

コピー元の LOB/FILE ロケータです。

dst_locpp (IN/OUT)

コピー先となる LOB/FILE ロケータです。コール元で、OCIDescriptorAlloc() をコールしてコピー先のロケータに領域を割り当てる必要があります。

コメント

ソース・ロケータを宛先ロケータに割り当てます。割当て後は、両方のロケータは同じ LOB 値を参照します。内部 LOB の場合は、宛先ロケータが表に格納されている時のみ、ソース・ロケータの LOB 値が宛先ロケータの LOB 値にコピーされます。したがって、宛先ロケータが格納されているオブジェクトのフラッシュを発行すると、LOB 値がコピーされます。

OCILobAssign() は一時 LOB には使用できません。OCI_INVALID_HANDLE エラーが生成されます。一時 LOB には、[OCILobLocatorAssign\(\)](#) を使用します。

FILE では、ファイルを参照するロケータのみが表にコピーされます。オペレーティング・システム・ファイル自体はコピーされません。

FILE ロケータを内部 LOB ロケータに割り当てたり、LOB ロケータを FILE ロケータに割り当てたりするとエラーになります。

ソース・ロケータがバッファリング可能な内部 LOB 用で LOB バッファリング・サブシステムを介した LOB データの変更に使用されており、そのバッファが書込み後フラッシュされていない場合には、ソース・ロケータが宛先ロケータに割り当てられない場合があります。

これは、LOB ごとに 1 つのロケータしか LOB バッファリング・サブシステムを介して LOB データを変更できないためです。

入力宛先ロケータの値は、`OCIDescriptorAlloc()` コールによってあらかじめ割り当てられている必要があります。たとえば、次のように宣言します。

```
OCILOBLocator    *source_loc = (OCILOBLocator *) 0;
OCILOBLocator    *dest_loc = (OCILOBLocator *) 0;
```

アプリケーションでは、`source_loc` ロケータが次のように割り当てられます。

```
if (OCIDescriptorAlloc((dvoid *) envhp, (dvoid **) &source_loc,
    (ub4) OCI_DTYPE_LOB, (size_t) 0, (dvoid **) 0))
    handle_error;
```

次に、表の LOB が `source_loc` に選択され、初期化されます。アプリケーションでは、`OCILOBAssign()` コールを発行して `source_loc` の値を `dest_loc` に割り当てる前に、宛先ロケータ `dest_loc` を割り当てる必要があります。次に例を示します。

```
if (OCIDescriptorAlloc((dvoid *) envhp, (dvoid **) &dest_loc,
    (ub4)OCI_DTYPE_LOB, (size_t) 0, (dvoid **) 0))
    handle_error;
if (OCILOBAssign(envhp, errhp, source_loc, &dest_loc))
    handle_error;
```

関連関数

`OCIErrorGet()`、`OCILOBIsEqual()`、`OCILOBLocatorAssign()`、
`OCILOBLocatorIsInit()`、`OCILOBEnableBuffering()`

OCILobCharSetForm()

用途

LOB ロケータのキャラクタ・セット・フォームを取得します。

構文

```
sword OCILobCharSetForm ( OCIEnv          *envhp,  
                          OCIError        *errhp,  
                          CONST OCILobLocator *locp,  
                          ub1              *csfrm );
```

パラメータ

envhp (IN/OUT)

OCI 環境ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

locp (IN)

キャラクタ・セット・フォームを取得するための LOB ロケータです。

csfrm (OUT)

入力 LOB ロケータのキャラクタ・セット・フォームです。入力ロケータの *locp* が BLOB または BFILE 用の場合、バイナリの LOB と FILE にはキャラクタ・セットという概念がないため、*csfrm* には 0 (ゼロ) が設定されます。コール元では、*csfrm* (**ub1**) 用の領域を割り当てる必要があります。

csfrm は、0 (ゼロ) 以外の次の 2 つの値をとることができます。

- SQLCS_IMPLICIT — データベース・キャラクタ・セット ID
- SQLCS_NCHAR — NCHAR キャラクタ・セット ID

デフォルト値は SQLCS_IMPLICIT です。

コメント

入力 LOB ロケータのキャラクタ・セット・フォームを *csfrm* 出力パラメータに戻します。この関数は、文字の LOB (つまり、CLOB および NCLOB) にのみ有効です。

関連関数

[OCIErrorGet\(\)](#)、[OCILobCharSetId\(\)](#)、[OCILobLocatorIsInit\(\)](#)

OCILobCharSetId()

用途

LOB ロケータのデータベース・キャラクタ・セット ID を取得します。

構文

```
sword OCILobCharSetId ( OCIEnv          *envhp,  
                        OCIError        *errhp,  
                        CONST OCILobLocator *locp,  
                        ub2              *csid );
```

パラメータ

envhp (IN/OUT)

OCI 環境ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCLErrorGet () に渡すエラー・ハンドルです。

locp (IN)

キャラクタ・セット ID を取得するための LOB ロケータです。

csid (OUT)

入力 LOB ロケータのデータベース・キャラクタ・セット ID です。入力ロケータが BLOB または BFILE 用の場合、バイナリの LOB/FILE にはキャラクタ・セットという概念がないため、*csid* には 0 (ゼロ) が設定されます。コール元は、*csid ub2* 用の領域を割り当てる必要があります。

コメント

入力 LOB ロケータのキャラクタ・セット ID を *csid* 出力パラメータに戻します。

この関数は、文字の LOB (つまり、CLOB、NCLOB) にのみ有効です。

関連関数

[OCLErrorGet\(\)](#)、[OCILobCharSetForm\(\)](#)、[OCILobLocatorIsInit\(\)](#)

OCILobClose()

用途

オープンしている LOB または FILE をクローズします。

構文

```
sword OCILobClose ( OCISvcCtx      *svchp,  
                    OCIError       *errhp,  
                    OCILobLocator  *locp );
```

パラメータ

svchp (IN)

サービス・コンテキスト・ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

locp (IN/OUT)

クローズする LOB です。ロケータは、内部 LOB または外部 LOB を参照できます。

コメント

オープンしている内部または外部 LOB をクローズします。BFILE は存在しているが、オープンしていない場合、エラーは戻りません。内部 LOB がオープンしていない場合は、エラーが戻されます。

LOB をクローズするときは、内部 LOB および外部 LOB からサーバーへのラウンドトリップが必要です。内部 LOB の場合は、LOB をクローズすると、クローズ・コールに依存している他のコードが実行されます。外部 LOB (BFILE) の場合は、LOB をクローズすると、サーバー側のオペレーティング・システム・ファイルが実際にクローズされます。

すべての LOB 操作をオープン・コールとクローズ・コールで囲む必要はありません。ただし、LOB をオープンしている場合は、トランザクションをコミットまたはロールバックする前に、その LOB をクローズする必要があります。内部 LOB をクローズすると、LOB 列のファンクション索引とドメイン索引が更新されます。トランザクションによってオープンされたすべての LOB は、トランザクションをコミットする前にクローズしておかないと、エラーが発生します。

このエラーが戻されると、LOB はオープンとしてマークされなくなりますが、トランザクションは正常にコミットされます。したがって、トランザクションで LOB データと LOB 以外のデータに対して行われた変更はすべてコミットされます。ただし、ドメイン索引とファンクション索引は更新されません。このエラーが戻された場合は、LOB 列のファンクション索引とドメイン索引を再作成してください。

LOB 操作をオープン API とクローズ API で囲んでいない場合、ファンクション索引とドメイン索引は、LOB に書き込むたびに更新されます。これによって、パフォーマンスが低下する可能性があるため、ファンクション索引とドメイン索引がある場合は、LOB への書き込み操作をオープン / クローズ文内に含めることをお勧めします。

関連項目： 7-11 ページ「[LOB のオープンおよびクローズのための関数](#)」

関連関数

[OCIErrorGet\(\)](#)、[OCILobOpen\(\)](#)、[OCILobIsOpen\(\)](#)

OCILobCopy()

用途

LOB 値の全体または一部を別の LOB 値にコピーします。

構文

```
sword OCILobCopy ( OCISvcCtx      *svchp,  
                  OCIError       *errhp,  
                  OCILobLocator   *dst_locp,  
                  OCILobLocator   *src_locp,  
                  ub4             amount,  
                  ub4             dst_offset,  
                  ub4             src_offset );
```

パラメータ

svchp (IN)

サービス・コンテキスト・ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

dst_locp (IN/OUT)

一意に宛先 LOB を参照する内部 LOB ロケータです。このロケータは、*svchp* に指定したサーバーから取得されたロケータにしてください。

src_locp (IN)

一意にソース LOB を参照する内部 LOB ロケータです。このロケータは、*svchp* に指定したサーバーから取得されたロケータにしてください。

amount (IN)

ソース LOB から宛先 LOB にコピーされる文字数（CLOB および NCLOB）またはバイト数（BLOB）です。

dst_offset (IN)

これは、宛先 LOB 用の絶対オフセットです。文字 LOB では、LOB の先頭から書込みを開始する位置までの文字数です。バイナリ LOB では、LOB の先頭から書込みを開始する位置までのバイト数です。オフセットは 1 から始まります。

src_offset (IN)

これは、ソース LOB の絶対オフセットです。文字 LOB では LOB の先頭からの文字数であり、バイナリ LOB ではバイト数です。オフセットは 1 から始まります。

コメント

内部 LOB 値全体または一部を別の内部 LOB 値に指定されたとおりにコピーします。データは、ソースから宛先にコピーされます。ソース (*src_locp*) LOB と宛先 (*dst_locp*) LOB はすでに存在する必要があります。

宛先のコピー開始位置にすでにデータが存在する場合は、ソース・データによって上書きされます。宛先のコピー開始位置がカレント・データの終了位置を超えている場合は、0 (ゼロ) バイトの充填文字 (BLOB の場合) または空白 (CLOB の場合) が、宛先 LOB のカレント・データの末尾と、ソースから新たに書き込まれたデータの先頭との間に書き込まれます。新規に書き込むデータが宛先 LOB の現行の長さよりも大きい場合、宛先 LOB は、そのデータにあわせて拡張されます。許される最大長 (つまり、4GB) を超える宛先 LOB の拡張と NULL LOB のコピーはエラーになります。

ソースと宛先の LOB ロケータは同じ型にしてください (つまり、両方とも BLOB または両方とも CLOB)。LOB バッファリングは、ロケータのどちらに対しても使用可能にしないでください。

この関数では、ソースまたは宛先としての FILE ロケータを受け入れません。

この LOB 操作をオープン・コールとクローズ・コールで囲む必要はありません。この操作を実行する前に LOB をオープンしていない場合、LOB 列のファンクション索引とドメイン索引はこのコール時に更新されます。この操作を実行する前に LOB をオープンしている場合は、トランザクションをコミットまたはロールバックする前に、その LOB をクローズする必要があります。内部 LOB をクローズすると、LOB 列のファンクション索引とドメイン索引が更新されます。

LOB 操作をオープン API とクローズ API で囲んでいない場合、ファンクション索引とドメイン索引は、LOB に書き込むたびに更新されます。これによって、パフォーマンスが低下する可能性があります。ファンクション索引とドメイン索引がある場合は、LOB への書き込み操作をオープン / クローズ文内に含めることをお勧めします。

注意： ソース LOB の長さを判断するには、`OCILobGetLength()` をコールします。

関連関数

[OCIErrorGet\(\)](#)、[OCILobRead\(\)](#)、[OCILobAppend\(\)](#)、[OCILobCopy\(\)](#)、[OCILobWrite\(\)](#)、[OCILobWriteAppend\(\)](#)

OCILobCreateTemporary()

用途

一時 LOB を作成します。

構文

```
sword OCILobCreateTemporary (OCISvcCtx          *svchp,  
                             OCIError           *errhp,  
                             OCILobLocator      *locp,  
                             ub2                csid,  
                             ub1                csfrm,  
                             ub1                lobtype,  
                             boolean            cache,  
                             OCIDuration       duration);
```

パラメータ

svchp (IN)

OCI サービス・コンテキスト・ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

locp (IN/OUT)

一時 LOB を指し示すロケータです。ロケータをこの関数に渡す前に、OCIDescriptorAlloc () を使用してロケータを割り当てる必要があります。このロケータは、LOB を指し示しているかどうかに関係なく上書きされます。

csid (IN)

LOB キャラクタ・セット ID です。Oracle8i 以上では OCI_DEFAULT を渡します。

csfrm (IN)

バッファ・データの LOB キャラクタ・セット・フォームです。Oracle8i 以上の場合は、OCI_DEFAULT を渡します。

lobtype (IN)

作成する LOB の型です。次の値が有効です。

- OCI_TEMP_BLOB — テンポラリ BLOB の場合
- OCI_TEMP_CLOB — テンポラリ CLOB の場合

cache (IN)

一時 LOB をキャッシュに読み込む必要がある場合は TRUE を、その必要がない場合は FALSE を渡します。NOCACHE 機能の場合、デフォルトは FALSE です。

duration (IN)

一時 LOB の継続時間です。次に示す値が有効です。

- OCI_DURATION_SESSION
- OCI_DURATION_CALL

コメント

この関数は、ユーザーの一時表領域に一時 LOB とそれに対応する索引を作成します。

この関数が完了すると、*lcp* パラメータは、長さが 0（ゼロ）の空の一時 LOB を指し示します。

一時 LOB の存続期間は、*duration* パラメータによって決まります。一時 LOB は、この期間の最後に解放されます。アプリケーションから `OCILobFreeTemporary()` コールを使用すると、一時 LOB をより早く解放できます。

LOB が BLOB の場合、*csid* パラメータおよび *csfrm* パラメータは無視されます。

関連項目： 一時 LOB とその継続時間の詳細は、7-17 ページの「[一時 LOB のサポート](#)」を参照してください。

関連関数

`OCILobFreeTemporary()`、`OCILobIsTemporary()`、`OCIDescriptorAlloc()`、`OCIErrorGet()`

OCILobDisableBuffering()

用途

入力ロケータの LOB バッファリングを使用禁止にします。

構文

```
sword OCILobDisableBuffering ( OCISvcCtx      *svchp,  
                                OCIError       *errhp,  
                                OCILobLocator  *locp );
```

パラメータ

svchp (IN)

サービス・コンテキスト・ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

locp (IN/OUT)

一意に LOB を参照する内部 LOB ロケータです。

コメント

入力内部 LOB ロケータの LOB バッファリングを使用禁止にします。次回入力ロケータを通して LOB に対してデータの読み込みまたは書き込みを行うときは、LOB バッファリング・サブシステムは使用されません。このコールでは、バッファリング・サブシステムで行われた変更は暗黙的にフラッシュされません。ユーザーは、OCILobFlushBuffer () をコールして変更を明示的にフラッシュする必要があります。

この関数は、FILE ロケータを受け入れません。

関連関数

[OCILobEnableBuffering\(\)](#)、[OCIErrorGet\(\)](#)、[OCILobFlushBuffer\(\)](#)

OCILobEnableBuffering()

用途

入力ロケータの LOB バッファリングを使用可能にします。

構文

```
sword OCILobEnableBuffering ( OCISvcCtx      *svchp,  
                              OCIError       *errhp,  
                              OCILobLocator  *locp );
```

パラメータ

svchp (IN)

サービス・コンテキスト・ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet() に渡すエラー・ハンドルです。

locp (IN/OUT)

一意に LOB を参照する内部 LOB ロケータです。

コメント

入力内部 LOB ロケータの LOB バッファリングを使用可能にします。次回入力ロケータを通して LOB に対してデータの読み込みまたは書き込みを行うときは、LOB バッファリング・サブシステムが使用されます。

ロケータの LOB バッファリングが使用可能な状態で、そのロケータがルーチン OCILobAppend()、OCILobCopy()、OCILobErase()、OCILobGetLength()、OCILobLoadFromFile()、OCILobTrim() または OCILobWriteAppend() のいずれかに渡されると、エラーが戻ります。

この関数は、FILE ロケータを受け入れません。

関連関数

[OCILobDisableBuffering\(\)](#)、[OCIErrorGet\(\)](#)、[OCILobWrite\(\)](#)、[OCILobRead\(\)](#)、[OCILobFlushBuffer\(\)](#)、[OCILobWriteAppend\(\)](#)

OCILobErase()

用途

内部 LOB データの一部を指定のオフセットの位置から消去します。

構文

```
sword OCILobErase ( OCISvcCtx      *svchp,  
                    OCIError       *errhp,  
                    OCILobLocator  *locp,  
                    ub4            *amount,  
                    ub4            offset );
```

パラメータ

svchp (IN)

サービス・コンテキスト・ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

locp (IN/OUT)

一意に LOB を参照する内部 LOB ロケータです。このロケータは、*svchp* に指定したサーバーから取得されたロケータにしてください。

amount (IN/OUT)

消去する文字数 (CLOB/NCLOB の場合) またはバイト数 (BLOB の場合) です。IN では、値は消去する文字数またはバイト数を表します。OUT では、値は消去された実際の文字数またはバイト数です。

offset (IN)

データの消去を開始する LOB 値の、先頭からの絶対オフセット (CLOB/NCLOB の場合は文字数、BLOB の場合はバイト数) です。オフセットは 1 から始まります。

コメント

実際に消去した文字数またはバイト数が戻ります。BLOB の場合、消去するということは、0 (ゼロ) バイトの充填文字で既存の LOB 値を上書きするということです。CLOB の場合は、既存の LOB 値が空白で上書きされます。

この関数は内部 LOB にのみ有効で、FILE には使用できません。

この LOB 操作をオープン・コールとクローズ・コールで囲む必要はありません。この操作を実行する前に LOB をオープンしていない場合、LOB 列のファンクション索引とドメイン索引はこのコール時に更新されます。この操作を実行する前に LOB をオープンしている場合は、トランザクションをコミットまたはロールバックする前に、その LOB をクローズする必要があります。内部 LOB をクローズすると、LOB 列のファンクション索引とドメイン索引が更新されます。

LOB 操作をオープン API とクローズ API で囲んでいない場合、ファンクション索引とドメイン索引は、LOB に書き込むたびに更新されます。これによって、パフォーマンスが低下する可能性があります。ファンクション索引とドメイン索引がある場合は、LOB への書き込み操作をオープン / クローズ文内に含めることをお勧めします。

関連関数

[OCIErrorGet\(\)](#)、[OCILobRead\(\)](#)、[OCILobAppend\(\)](#)、[OCILobCopy\(\)](#)、[OCILobWrite\(\)](#)、[OCILobWriteAppend\(\)](#)

OCILobFileClose()

用途

オープンしている FILE をクローズします。

構文

```
sword OCILobFileClose ( OCISvcCtx          *svchp,  
                        OCIError          *errhp,  
                        OCILobLocator      *filep );
```

パラメータ

svchp (IN)

サービス・コンテキスト・ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

filep (IN/OUT)

クローズする FILE を参照する FILE ロケータへのポインタです。

コメント

オープンしている FILE をクローズします。内部 LOB に対してこの関数をコールするとエラーになります。FILE が存在していてオープンしていないときは、エラーは戻りません。

この関数は、特定の FILE ロケータに対して初めてコールしたときにかぎり有効です。同じ FILE ロケータを使用してこの関数を続けてコールしても何も行われません。

関連項目： FILE の詳細は、『Oracle9i アプリケーション開発者ガイド - ラージ・オブジェクト』の BFILE の説明を参照してください。

関連関数

[OCIErrorGet\(\)](#)、[OCILobClose\(\)](#)、[OCILobFileCloseAll\(\)](#)、
[OCILobFileExists\(\)](#)、[OCILobFileIsOpen\(\)](#)、[OCILobFileOpen\(\)](#)、
[OCILobOpen\(\)](#)、[OCILobIsOpen\(\)](#)

OCILobFileCloseAll()

用途

指定のサービス・コンテキストでオープンしているすべての FILE をクローズします。

構文

```
sword OCILobFileCloseAll ( OCISvcCtx   *svchp,  
                           OCIError    *errhp );
```

パラメータ

svchp (IN)

サービス・コンテキスト・ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

コメント

指定のサービス・コンテキストでオープンしているすべての FILE をクローズします。

関連項目： FILE の詳細は、『Oracle9i アプリケーション開発者ガイド - ラージ・オブジェクト』の BFILE の説明を参照してください。

関連関数

[OCILobFileClose\(\)](#)、[OCIErrorGet\(\)](#)、[OCILobFileExists\(\)](#)、[OCILobFileIsOpen\(\)](#)

OCILobFileExists()

用途

サーバーのオペレーティング・システムに FILE が存在するかどうかを確認します。

構文

```
sword OCILobFileExists ( OCISvcCtx      *svchp,  
                          OCIError      *errhp,  
                          OCILobLocator *filep,  
                          boolean       *flag );
```

パラメータ

svchp (IN)

OCI サービス・コンテキスト・ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

filep (IN)

ファイルを参照する FILE ロケータへのポインタです。

flag (OUT)

サーバーに FILE が存在する場合は TRUE を、存在しない場合は FALSE を戻します。

コメント

サーバーのファイル・システムに FILE が存在するかどうかをチェックします。内部 LOB に対してこの関数をコールするとエラーになります。

関連項目： FILE の詳細は、『Oracle9i アプリケーション開発者ガイド - ラージ・オブジェクト』の BFILE の説明を参照してください。

関連関数

[OCIErrorGet\(\)](#)、[OCILobFileClose\(\)](#)、[OCILobFileCloseAll\(\)](#)、
[OCILobFileIsOpen\(\)](#)、[OCILobOpen\(\)](#)、[OCILobIsOpen\(\)](#)

OCILobFileGetName()

用途

FILE ロケータのディレクトリ別名およびファイル名を取得します。

構文

```
sword OCILobFileGetName ( OCIEnv                *envhp,  
                           OCIError              *errhp,  
                           CONST OCILobLocator    *filep,  
                           text                   *dir_alias,  
                           ub2                   *d_length,  
                           text                   *filename,  
                           ub2                   *f_length );
```

パラメータ

envhp (IN/OUT)

OCI 環境ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCLErrorGet () に渡すエラー・ハンドルです。

filep (IN)

ディレクトリ別名およびファイル名を取得するための FILE ロケータです。

dir_alias (OUT)

ディレクトリのエイリアスが配置されているバッファです。これは UTF-16 で可能です。ディレクトリのエイリアス用の十分な領域を割り当てる必要があります。ディレクトリ別名の最大長は 30 バイトです。

d_length (IN/OUT)

次の用途に使用できます (Unicode のコードポイントまたはバイト単位が可能です)。

- IN: 入力 *dir_alias* 文字列の長さ
- OUT: 戻される *dir_alias* 文字列の長さ

filename (OUT)

ファイル名が配置されているバッファです。ファイル名用の十分な領域を割り当てる必要があります。ファイル名の最大長は 255 バイトです。

f_length (IN/OUT)

次の用途に使用できます (バイト数単位)。

- IN: 入力 *filename* バッファの長さ
- OUT: 戻される *filename* 文字列の長さ

コメント

この FILE ロケータに対応付けられたディレクトリ別名とファイル名を戻します。環境ハンドルによって、Unicode かどうかが判断されます。内部 LOB に対してこの関数をコールするとエラーになります。

関連項目： FILE の詳細は、『Oracle9i アプリケーション開発者ガイド - ラージ・オブジェクト』の BFILE の説明を参照してください。

関連関数

[OCILobFileName\(\)](#)、[OCIErrorGet\(\)](#)

OCILobFileIsOpen()

用途

FILE がオープンしているかどうかを確認します。

構文

```
sword OCILobFileIsOpen ( OCISvcCtx      *svchp,  
                          OCIError      *errhp,  
                          OCILobLocator *filep,  
                          boolean       *flag );
```

パラメータ

svchp (IN)

OCI サービス・コンテキスト・ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

filep (IN)

確認する FILE ロケータへのポインタです。

flag (OUT)

この特定のロケータを使用して FILE がオープンされた場合は TRUE を、オープンされなかった場合は FALSE を戻します。

コメント

サーバー上のファイルが *filep* FILE ロケータを使用してオープンされたかどうかをチェックします。内部 LOB に対してこの関数をコールするとエラーになります。

OCILobFileOpen() または OCILobOpen() コマンドに入力 FILE ロケータが渡されなかった場合、ファイルはそのロケータによってオープンされていないとみなされます。ただし、別のロケータがそのファイルをオープンしていることがあります。オープンは、特定のロケータに対応付けられます。

関連項目： FILE の詳細は、『Oracle9i アプリケーション開発者ガイド - ラージ・オブジェクト』の BFILE の説明を参照してください。

関連関数

[OCIErrorGet\(\)](#)、[OCILobClose\(\)](#)、[OCILobFileCloseAll\(\)](#)、
[OCILobFileExists\(\)](#)、[OCILobFileClose\(\)](#)、[OCILobFileOpen\(\)](#)、[OCILobOpen\(\)](#)、
[OCILobIsOpen\(\)](#)

OCILobFileOpen()

用途

読取り専用アクセス用にサーバーのファイル・システム上の FILE をオープンします。

構文

```
sword OCILobFileOpen ( OCISvcCtx          *svchp,  
                        OCIError          *errhp,  
                        OCILobLocator      *filep,  
                        ub1                 mode );
```

パラメータ

svchp (IN)

サービス・コンテキスト・ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

filep (IN/OUT)

オープンする FILE です。ロケータが FILE を参照していない場合はエラーが発生します。

mode (IN)

ファイルをオープンするモードです。有効なモードは OCI_FILE_READONLY のみです。

コメント

サーバーのファイル・システムの FILE をオープンします。FILE は、読取り専用アクセス用にオープンできます。Oracle を介して FILE には書き込みできません。内部 LOB に対してこの関数をコールするとエラーになります。

この関数は、特定の FILE ロケータに対して初めてコールしたときにかぎり有効です。同じ FILE ロケータを使用してこの関数を続けてコールしても何も行われません。

関連項目： FILE の詳細は、『Oracle9i アプリケーション開発者ガイド - ラージ・オブジェクト』の BFILE の説明を参照してください。

関連関数

[OCIErrorGet\(\)](#)、[OCILobClose\(\)](#)、[OCILobFileCloseAll\(\)](#)、
[OCILobFileExists\(\)](#)、[OCILobFileClose\(\)](#)、[OCILobFileIsOpen\(\)](#)、
[OCILobOpen\(\)](#)、[OCILobIsOpen\(\)](#)

OCILobFileName()

用途

FILE ロケータ内にディレクトリ別名とファイル名を設定します。

構文

```
sword OCILobFileName ( OCIEnv          *envhp,
                      OCIError        *errhp,
                      OCILobLocator   **filepp,
                      CONST text      *dir_alias,
                      ub2              d_length,
                      CONST text      *filename,
                      ub2              f_length );
```

パラメータ

envhp (IN/OUT)

OCI 環境ハンドルです。UTF-16 設定が含まれます。

errhp (IN/OUT)

エラー発生時の診断情報のために OCLErrorGet () に渡すエラー・ハンドルです。

filepp (IN/OUT)

ディレクトリ別名およびファイル名を設定する FILE ロケータへのポインタです。

dir_alias (IN)

FILE ロケータに設定するディレクトリのエイリアス (OCI_UTF16 環境の UTF-16 で可能) が含まれたバッファです。

d_length (IN)

入力 *dir_alias* パラメータの長さです。バイト単位です。

filename (IN)

FILE ロケータに設定するファイル名 (OCI_UTF16 環境の UTF-16 で可能) が含まれたバッファです。

f_length (IN)

入力 *filename* パラメータの長さです。バイト単位です。

コメント

内部 LOB に対してこの関数をコールするとエラーになります。

関連項目： FILE の詳細は、『Oracle9i アプリケーション開発者ガイド - ラージ・オブジェクト』の BFILE の説明を参照してください。

関連関数

`OCILobFileName()`、`OCIErrorGet()`

OCILobFlushBuffer()

用途

この LOB のすべてのバッファをサーバーにフラッシュするか、または書き込みます。

構文

```
sword OCILobFlushBuffer ( OCISvcCtx      *svchp,  
                           OCIError       *errhp,  
                           OCILobLocator  *locp  
                           ub4            flag );
```

パラメータ

svchp (IN/OUT)

サービス・コンテキスト・ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

locp (IN/OUT)

LOB を一意に参照する内部ロケータです。

flag (IN)

OCI_LOB_BUFFER_FREE を設定すると、LOB のバッファ・リソースは、フラッシュの後に解放されます。次のコメントを参照してください。

コメント

入力ロケータによって参照される LOB に関連するバッファリング・サブシステムに加えられた変更を、サーバーにフラッシュします。このルーチンでは、バッファのデータをデータベースの LOB に実際に書き込みます。LOB バッファリングは入力 LOB ロケータに対してあらかじめ使用可能にしてください。

このフラッシュ操作のデフォルトでは、別のバッファ LOB 操作への再割当て用にバッファ・リソースは解放されません。ただし、バッファを明示的に解放する場合は、flag パラメータを OCI_LOB_BUFFER_FREE に設定できます。

クライアント・アプリケーションでフラッシュ後のバッファ値を読み込む予定で、あらかじめバッファの現在値が望む値であることがわかっている場合、サーバーからデータを再度読み込む必要はありません。

バッファの解放による影響はユーザーにはほとんど意識されません。ただし、LOB 内の同じ範囲に次にアクセスするときに、サーバーをラウンドトリップすることになり、バッファ・リソースの取得および LOB から読み込まれるデータによる初期化のコストもかかります。このオプションは、オンボード・メモリーが少ないクライアント環境用です。

関連関数

`OCILobEnableBuffering()`、`OCIErrorGet()`、`OCILobWrite()`、`OCILobRead()`、
`OCILobDisableBuffering()`、`OCILobWriteAppend()`

OCILobFreeTemporary()

用途

一時 LOB を解放します。

構文

```
sword OCILobFreeTemporary( OCISvcCtx          *svchp,  
                             OCIError          *errhp,  
                             OCILobLocator     *locp);
```

パラメータ

svchp (IN/OUT)

OCI サービス・コンテキスト・ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet() に渡すエラー・ハンドルです。

locp (IN/OUT)

解放する LOB を一意に参照するロケータです。

コメント

このロケータが指し示す一時 LOB の内容を解放します。ロケータ自体は、OCIDescriptorFree() がコールされるまで解放されません。

locp パラメータで渡される LOB ロケータが一時 LOB を指し示していない場合は、エラーを戻します。次のいずれかが原因である可能性があります。

- ロケータが永続 LOB を指し示しています。
- ロケータがすでに解放された一時 LOB を指し示しています。
- ロケータが何も指し示していません。

関連関数

[OCILobCreateTemporary\(\)](#)、[OCILobIsTemporary\(\)](#)、[OCIErrorGet\(\)](#)

OCILobGetChunkSize()

用途

LOB のチャンク・サイズを取得します。

構文

```
sword OCILobGetChunkSize ( OCISvcCtx      *svchp,  
                           OCIError       *errhp,  
                           OCILobLocator  *locp,  
                           ub4            *chunk_size );
```

パラメータ

svchp (IN)

サービス・コンテキスト・ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

locp (IN/OUT)

使用可能なチャンク・サイズを取得する内部 LOB です。

chunk_size (OUT)

内部 LOB 値を格納するために使用するチャンク領域の量です。これは、ユーザーが LOB 値の読み込み / 書き込み時に使用する量です。可能な場合は、チャンクの先頭などのチャンクの境界で書き込みを開始し、1 つのチャンクを 1 回で書き込みます。

chunk_size は、BLOB ではバイト数、CLOB と NCLOB では文字数で戻されます。可変幅のキャラクタ・セットの場合は、チャンクと同じ大きさの Unicode 文字数です。

コメント

内部 LOB が格納される表を作成するときに、チャンク係数を指定できます。チャンク係数は、Oracle ブロックの倍数で、LOB 値に対するアクセスまたは修正時に、LOB データ層によって使用されるチャンク・サイズに対応しています。チャンクの一部はシステム関連情報に使用され、残りのチャンクには LOB 値が格納されます。この関数では、LOB 値が格納される LOB チャンクで使用される領域の量を戻します。アプリケーションからこのチャンク・サイズの倍数を使用して読み込みまたは書き込み要求を発行すると、パフォーマンスが向上します。書き込みの場合は、他にも利点があります。LOB チャンクはバージョン管理されているので、すべての書き込みをチャンク単位で行うと、不要または重複したバージョンングが行われることがありません。同じチャンクに対して複数の書き込みコールを発行するかわりに、チャンクがいっぱいになるまで書き込みを蓄積することもできます。

関連項目： 7-10 ページ [「LOB の読み込み / 書き込みパフォーマンスを向上するための関数」](#)

関連関数

`OCIErrorGet()`、`OCILobRead()`、`OCILobAppend()`、`OCILobCopy()`、
`OCILobWrite()`、`OCILobWriteAppend()`

OCILobGetLength()

用途

LOB の長さを取得します。

構文

```
sword OCILobGetLength ( OCISvcCtx      *svchp,  
                        OCIError       *errhp,  
                        OCILobLocator  *locp,  
                        ub4            *lenp );
```

パラメータ

svchp (IN)

サービス・コンテキスト・ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

locp (IN)

一意に LOB を参照する LOB ロケータです。内部 LOB では、このロケータは、*svchp* に指定したサーバーから取得されたロケータにしてください。FILE の場合は、OCILOBFileNameSet ()、SELECT 文または OCIObjectPin () を使用して設定できます。

lenp (OUT)

出力時に LOB が NULL でない場合は、LOB の長さです。文字の LOB の場合は、LOB 内の文字数です。バイナリの LOB および BFILE の場合は、LOB 内のバイト数です。

コメント

LOB の長さを取得します。LOB が NULL の場合、長さは未定義です。FILE の長さには EOF (存在する場合) が含まれます。空の内部 LOB の長さは 0 (ゼロ) です。

クライアント側のキャラクタ・セットが可変幅かどうかに関係なく、出力の長さは、CLOB と NCLOB では文字数、BLOB と BFILE ではバイト数です。

注意： 以前の OCILobErase () または OCILobWrite () のコールによって LOB に書き込まれた 0 (ゼロ) バイトや空白の充填文字も、長さの一部になります。

関連関数

[OCIErrorGet \(\)](#)、[OCILobFileNameSet \(\)](#)、[OCILobRead \(\)](#)、[OCILobWrite \(\)](#)、[OCILobCopy \(\)](#)、[OCILobAppend \(\)](#)、[OCILobLoadFromFile \(\)](#)、[OCILobWriteAppend \(\)](#)

OCILobIsEqual()

用途

2 つの LOB/FILE ロケータの等価を比較します。

構文

```
sword OCILobIsEqual ( OCIEnv          *envhp,  
                      CONST OCILobLocator *x,  
                      CONST OCILobLocator *y,  
                      boolean           *is_equal );
```

パラメータ

envhp (IN)

OCI 環境ハンドルです。

x (IN)

比較する LOB ロケータです。

y (IN)

比較する LOB ロケータです。

is_equal (OUT)

LOB ロケータが等価の場合は TRUE、等価でない場合は FALSE です。

コメント

指定された LOB/FILE ロケータの等価を比較します。2 つの LOB/FILE ロケータは、どちらも同じ LOB/FILE 値を参照している場合のみ等価になります。

この関数では、2 つの NULL ロケータは等価とみなされません。

関連関数

[OCILobAssign\(\)](#)、[OCILobLocatorIsInit\(\)](#)

OCILobIsOpen()

用途

LOB または FILE がオープンされているかどうかを確認します。

構文

```
sword OCILobIsOpen ( OCISvcCtx      *svchp,  
                     OCIError      *errhp,  
                     OCILobLocator *locp,  
                     boolean        *flag );
```

パラメータ

svchp (IN)

サービス・コンテキスト・ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

locp (IN)

確認する LOB ロケータへのポインタです。ロケータは、内部 LOB または外部 LOB を参照できます。

flag (OUT)

内部 LOB がオープンされている場合、または入力ロケータを使用して BFILE がオープンされた場合は、TRUE を戻します。それ以外は FALSE を戻します。

コメント

内部 LOB がオープンされているかどうか、または BFILE が入力ロケータを使用してオープンされているかどうかをチェックします。

BFILES の場合

入力 BFILE ロケータが OCILobOpen () または OCILobFileOpen () に渡されていない場合は、BFILE ロケータによる BFILE のオープンが行わないとみなされます。ただし、別の BFILE ロケータによって BFILE がオープンされている場合があります。別のロケータを使用して、その BFILE に対して複数のオープンが実行されることがあります。つまり、オープンは BFILE の特定のロケータに関連付けられています。

内部 LOB の場合

オープンは、ロケータではなく LOB に関連付けられます。locator1 が LOB をオープンした場合は、locator2 もオープン時にその LOB を参照します。

内部 LOB の場合、このコールでは、サーバー上の状態によって、LOB がオープンしているかどうかを確認するため、サーバー・ラウンドトリップが必要です。外部 LOB (BFILE) の場合も、サーバー側のオペレーティング・システム・ファイルによってその LOB がオープンされているかどうかを確認するため、ラウンドトリップが必要です。

関連項目： 7-11 ページ「[LOB のオープンおよびクローズのための関数](#)」

関連関数

`OCIErrorGet()`、`OCILobClose()`、`OCILobFileCloseAll()`、
`OCILobFileExists()`、`OCILobFileClose()`、`OCILobFileIsOpen()`、
`OCILobFileOpen()`、`OCILobOpen()`

OCILobIsTemporary()

用途

ロケータが一時 LOB を指し示しているかどうかを確認します。

構文

```
sword OCILobIsTemporary(OCIEnv          *envhp,  
                        OCIError         *errhp,  
                        OCILobLocator    *locp,  
                        boolean          *is_temporary);
```

パラメータ

envhp (IN)

OCI 環境ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCLErrorGet () に渡すエラー・ハンドルです。

locp (IN)

確認するロケータです。

is_temporary (OUT)

ロケータが一時 LOB を指し示している場合は TRUE を、そうでない場合は FALSE を返します。

コメント

この関数は、ロケータが一時 LOB を指し示しているかどうかを調べるためにロケータを確認します。指し示している場合は、*is_temporary* が TRUE に設定されます。そうでない場合は、*is_temporary* が FALSE に設定されます。

関連関数

[OCILobCreateTemporary\(\)](#)、[OCILobFreeTemporary\(\)](#)

OCILobLoadFromFile()

用途

ファイルの全体または一部を内部 LOB にロード / コピーします。

構文

```
sword OCILobLoadFromFile ( OCISvcCtx      *svchp,  
                           OCIError       *errhp,  
                           OCILobLocator   *dst_locp,  
                           OCILobLocator   *src_locp,  
                           ub4             amount,  
                           ub4             dst_offset,  
                           ub4             src_offset );
```

パラメータ

svchp (IN)

サービス・コンテキスト・ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

dst_locp (IN/OUT)

BLOB 型、CLOB 型または NCLOB 型の宛先内部 LOB を一意に参照するロケータです。

src_locp (IN/OUT)

ソース FILE を一意に参照するロケータです。

amount (IN)

ロードされるバイト数です。

dst_offset (IN)

これは、宛先 LOB 用の絶対オフセットです。文字 LOB では、LOB の先頭から書込みを開始する位置までの文字数です。バイナリ LOB では、LOB の先頭から読取りを開始する位置までのバイト数です。オフセットは 1 から始まります。

src_offset (IN)

これは、ソース FILE にとっての絶対表示オフセットです。FILE の先頭からのバイト数です。オフセットは 1 から始まります。

コメント

FILE 値の一部またはすべてを指定どおりに内部 LOB にロードまたはコピーします。データは、ソース FILE から宛先の内部 LOB (BLOB または CLOB) にコピーされます。FILE データを CLOB または NCLOB にコピーするときに、キャラクタ・セットの変換は実行されません。バイナリ・データが BLOB にロードされるときにも、キャラクタ・セット変換は実行されません。このため、FILE データは、あらかじめデータベースの LOB と同じキャラクタ・セットにしてください。これを検証するエラー・チェックは実行されません。

ソース (*src_locp*) LOB と宛先 (*dst_locp*) LOB はすでに存在している必要があります。宛先のコピー開始位置にすでにデータが存在する場合は、ソース・データによって上書きされます。宛先のコピー開始位置がカレント・データの終了位置を超えている場合は、0 (ゼロ) バイトの充填文字 (BLOB の場合) または空白 (CLOB の場合) が、宛先 LOB のデータの末尾と、ソースから新たに書き込まれたデータの先頭との間に書き込まれます。新規に書き込むデータが宛先 LOB の現行の長さよりも大きい場合、宛先 LOB は、そのデータにあわせて拡張されます。

許される最大長 (4GB) を超えての宛先 LOB の拡張と NULL FILE のコピーはエラーになります。

この LOB 操作をオープン・コールとクローズ・コールで囲む必要はありません。この操作を実行する前に LOB をオープンしていない場合、LOB 列のファンクション索引とドメイン索引はこのコール時に更新されます。この操作を実行する前に LOB をオープンしている場合は、トランザクションをコミットまたはロールバックする前に、その LOB をクローズする必要があります。内部 LOB をクローズすると、LOB 列のファンクション索引とドメイン索引が更新されます。

LOB 操作をオープン API とクローズ API で囲んでいない場合、ファンクション索引とドメイン索引は、LOB に書き込むたびに更新されます。これによって、パフォーマンスが低下する可能性があります。ファンクション索引とドメイン索引がある場合は、LOB への書き込み操作をオープン / クローズ文内に含めることをお勧めします。

関連関数

[OCIErrorGet\(\)](#)、[OCILobAppend\(\)](#)、[OCILobWrite\(\)](#)、[OCILobTrim\(\)](#)、[OCILobCopy\(\)](#)、[OCILobGetLength\(\)](#)、[OCILobWriteAppend\(\)](#)

OCILobLocatorAssign()

用途

LOB/FILE ロケータを別のロケータに割り当てます。

構文

```
sword OCILobLocatorAssign ( OCISvcCtx          *svchp,  
                             OCIError          *errhp,  
                             CONST OCILobLocator *src_locp,  
                             OCILobLocator      **dst_locpp );
```

パラメータ

svchp (IN/OUT)

OCI サービス・コンテキスト・ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet() に渡すエラー・ハンドルです。

src_locp (IN)

コピー元の LOB/FILE ロケータです。

dst_locpp (IN/OUT)

コピー先となる LOB/FILE ロケータです。コール元で、OCIDescriptorAlloc() をコールして OCILobLocator に領域を割り当ててする必要があります。

コメント

このコールは、ソース・ロケータを宛先ロケータに割り当てます。割当て後は、両方のロケータは同じ LOB データを参照します。内部 LOB の場合は、宛先ロケータが表に格納されているときのみ、ソース・ロケータの LOB データが宛先ロケータの LOB データにコピーされます。このため、宛先ロケータが含まれているオブジェクトにフラッシュを発行すると、LOB データがコピーされます。FILE の場合、OS ファイルを参照するロケータのみが表にコピーされ、OS ファイルはコピーされません。

このコールは OCILobAssign() と似ていますが、OCILobLocatorAssign() は OCI 環境ハンドル・ポインタではなく、OCI サービス・ハンドル・ポインタを使用します。また、OCILobLocatorAssign() は一時 LOB に対して使用できますが、OCILobAssign() は使用できません。

注意： OCILobLocatorAssign() 関数が正常に終了しなかった場合、ターゲット・ロケータは以前の状態にはリストアされません。ターゲット・ロケータは、初期化しなおすまで、後続の操作で使用しないでください。

宛先ロケータが一時 LOB に対するロケータの場合、宛先一時 LOB は、ソース LOB を割り当てる前に解放されます。

ソース LOB ロケータから一時 LOB を参照すると、宛先ロケータが一時 LOB にも作成されます。ソース・ロケータと宛先ロケータは、理論的には異なる一時 LOB になります。OCI_DEFAULT モードでは、ソースの一時 LOB がディープ・コピーされ、その一時 LOB の新しいディープ・コピーを参照するために宛先ロケータが作成されます。したがって、OCILobLocatorAssign() をコールした後は、OCILobIsEqual() で FALSE が戻ります。ただし、OCI_OBJECT モードの場合は、ディープ・コピー数を最小化するために最適化が行われるため、ソース・ロケータと宛先ロケータは、いずれかの LOB ロケータを介して変更が行われるまで、同じ LOB を指し示します。したがって、最初の変更が行われるまで、OCILobLocatorAssign() 直後の OCILobIsEqual() では、TRUE が戻ります。これらのいずれの場合にも、OCILobLocatorAssign() 後のソースまたは宛先に対する変更は、他方（つまり、宛先またはソース）の LOB には反映されません。ソースおよび宛先が同じ LOB を指し示し、変更を他方にも反映する場合は、等号を使用して、2 つの LOB ロケータ・ポインタが同じ LOB ロケータを参照するようにします。

関連関数

[OCIErrorGet\(\)](#)、[OCILobAssign\(\)](#)、[OCILobIsEqual\(\)](#)、[OCILobLocatorIsInit\(\)](#)

OCILobLocatorIsInit()

用途

指定された LOB/FILE ロケータが初期化されているかどうかを確認します。

構文

```
sword OCILobLocatorIsInit ( OCIEnv          *envhp,  
                             OCIError        *errhp,  
                             CONST OCILobLocator *locp,  
                             boolean          *is_initialized );
```

パラメータ

envhp (IN/OUT)

OCI 環境ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCLErrorGet() に渡すエラー・ハンドルです。

locp (IN)

確認する LOB/FILE ロケータです。

is_initialized (OUT)

特定の LOB/FILE ロケータが初期化されていると TRUE、初期化されていないと FALSE を返します。

コメント

指定された LOB/FILE ロケータが初期化されているかどうかを確認します。

内部 LOB ロケータは、次のいずれかの方法で初期化できます。

- NULL 以外の LOB をロケータに選択します。
- OCIObjectPin() を使用して、LOB 属性が NULL 以外のオブジェクトを確保します。
- OCIAttrSet() を使用して、ロケータを Empty 値に設定します。

関連項目： A-41 ページ [「LOB ロケータ属性」](#)

FILE ロケータは、次のいずれかの方法で初期化できます。

- NULL 以外の FILE をロケータに選択します。
- `OCIObjectPin()` を使用して、FILE 属性が NULL 以外のオブジェクトを確保します。
- `OCILobFileName()` をコールします。

関連関数

[OCIErrorGet\(\)](#)、[OCILobIsEqual\(\)](#)

OCILobOpen()

用途

指定されたモードで LOB（内部または外部）をオープンします。

構文

```
sword OCILobOpen ( OCISvcCtx      *svchp,  
                   OCIError       *errhp,  
                   OCILobLocator  *locp,  
                   ub1             mode );
```

パラメータ

svchp (IN)

サービス・コンテキスト・ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

locp (IN/OUT)

オープンする LOB です。ロケータは、内部 LOB または外部 LOB を参照できます。

mode (IN)

LOB/BFILE をオープンするモードです。Oracle8i 以上の場合、LOB に対する有効なモードは、OCI_LOB_READONLY および OCI_LOB_READWRITE です。OCI_FILE_READONLY は、OCILobFileOpen () の入力モードとして使用します。入力ロケータが BFILE に対するロケータの場合、OCI_FILE_READONLY は OCILobOpen () とともに使用できます。

コメント

同じ LOB を 2 度オープンするとエラーが発生します。BFILE は、読取り / 書込みモードではオープンできません。LOB/BFILE が読取り専用モードでオープンされている場合は、LOB/BFILE に書込みを行うと、エラーが戻されます。

LOB のオープンでは、内部 LOB と外部 LOB のどちらに対してもサーバーへのラウンドトリップが必要です。内部 LOB をオープンすると、そのオープン・コールに応じて他のコードが実行されます。外部 LOB (BFILE) をオープンすると、サーバー側のオペレーティング・システム・ファイルがオープンされるため、ラウンドトリップが必要です。

LOB を操作するために、LOB をオープンする必要はありません。ファンクション索引、拡張索引作成機能またはコンテキストを使用し、複数のコールを行って LOB に対する更新または書込みを行う場合は、最初に OCILobOpen () をコールし、次に、その LOB を必要な回数だけ更新し、最後に OCILobClose () をコールします。この一連の操作によって、索引は、書込み操作ごとに更新されるのではなく、すべての書込み操作の最後に 1 回で更新されます。

すべての LOB 操作をオープン・コールとクローズ・コールで囲む必要はありません。ただし、LOB をオープンしている場合は、トランザクションをコミットまたはロールバックする前に、その LOB をクローズする必要があります。内部 LOB をクローズすると、LOB 列のファンクション索引とドメイン索引が更新されます。トランザクションによってオープンされたすべての LOB は、トランザクションをコミットする前にクローズしておかないと、エラーが発生します。

このエラーが戻されると、LOB はオープンとしてマークされなくなりますが、トランザクションは正常にコミットされます。したがって、トランザクションで LOB データと LOB 以外のデータに対して行われた変更はすべてコミットされます。ただし、ドメイン索引とファンクション索引は更新されません。このエラーが戻された場合は、LOB 列のファンクション索引とドメイン索引を再作成してください。

LOB 操作をオープン API とクローズ API で囲んでいない場合、ファンクション索引とドメイン索引は、LOB に書き込むたびに更新されます。これによって、パフォーマンスが低下する可能性があるため、ファンクション索引とドメイン索引がある場合は、LOB への書き込み操作をオープン / クローズ文内に含めることをお勧めします。

関連項目： 7-11 ページ「[LOB のオープンおよびクローズのための関数](#)」

関連関数

[OCIErrorGet\(\)](#)、[OCILobClose\(\)](#)、[OCILobFileCloseAll\(\)](#)、
[OCILobFileExists\(\)](#)、[OCILobFileClose\(\)](#)、[OCILobFileIsOpen\(\)](#)、
[OCILobFileOpen\(\)](#)、[OCILobIsOpen\(\)](#)

OCILobRead()

用途

LOB/FILE の一部をコールの指定どおりにバッファに読み込みます。

構文

```
sword OCILobRead ( OCISvcCtx          *svchp,
                   OCIError           *errhp,
                   OCILobLocator      *locp,
                   ub4                *amtp,
                   ub4                offset,
                   dvoid              *bufp,
                   ub4                bufl,
                   dvoid              *ctxp,
                   OCICallbackLobRead (cbfp)
                   ( dvoid              *ctxp,
                     CONST dvoid        *bufp,
                     ub4                len,
                     ub1                piece )
                   ub2                csid,
                   ub1                csfrm );
```

パラメータ

svchp (IN/OUT)

サービス・コンテキスト・ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

locp (IN)

一意に LOB/FILE を参照する LOB/FILE ロケータです。このロケータは、*svchp* に指定したサーバーから取得されたロケータにしてください。

amtp (IN/OUT)

次の表に示すように、*amtp* の値は、バイト数または文字数のいずれかで表される量です。

LOB/FILE	入力	固定幅のクライアント側 キャラクタ・セットによる 出力	可変幅のクライアント側 キャラクタ・セットによる 出力
BLOB および BFILE	バイト	バイト	バイト
CLOB および NCLOB	文字	文字	バイト (1)

(1) 入力量は、サーバー側の CLOB/NCLOB から読み込む文字数を示します。出力量は、バッファ *bufp* に読み込まれたバイト数を示します。

次の場合、**amtp* は読み込まれたデータの総量です。

- データがストリーム・モードで読み込まれない場合 (1 ピースのみ読み込まれ、ポーリングまたはコールバックは行われません)。
- データがコールバックによってストリーム・モードで読み込まれる場合

データがポーリングを使用してストリーム・モードで読み込まれる場合、**amtp* は読み込まれた最後のピースの長さです。

読み込まれる量がバッファの長さより多い場合、ストリーム・モードでの LOB の読み込みは、入力オフセットから LOB の終わりまでか、指定されたバイト数の読み込みまで、いずれか先に達するところまで行われます。入力時にこの値が 0 (ゼロ) の場合は、入力オフセットから LOB の末尾までストリーム・モードでデータが読み込まれます。

ストリーム・モード (ポーリングまたはコールバックのいずれかで実行) の場合、LOB 値は入力オフセットから連続して読み込まれます。

データがピース単位で読み込まれる場合、**amtp* には常に読み込まれた各ピースの長さが含まれます。

コールバック関数が定義されている場合は、パイプから *buf1* バイトが読み込まれるたびにそのコールバック関数が呼び出されます。各ピースが *bufp* に書き込まれます。

コールバック関数が定義されていない場合は、OCI_NEED_DATA エラー・コードが戻ります。アプリケーションでは、OCI_NEED_DATA エラー・コードが戻らなくなるまで何度も *OCILobRead()* をコールし、LOB のピースを読み込む必要があります。ピースのサイズを変えながら複数の場所を読み込む場合は、コールごとにバッファ・ポインタと長さを変えることができます。

offset (IN)

入力時、これは LOB 値の先頭からの絶対オフセットです。キャラクタ LOB (CLOB、NCLOB) の場合は、LOB の先頭からの文字数です。バイナリ LOB/FILE の場合はバイト数です。先頭位置は 1 です。

ポーリングまたはコールバックによるストリームを使用する場合は、最初のコールでオフセットを指定します。後続のポーリング・コールでは、**offset** パラメータは無視されます。コールバックを使用する場合、**offset** パラメータはありません。

bufp (IN/OUT)

ピースの読み込み先バッファへのポインタです。*buf1* のメモリー長が割り当てられるとみなされます。

buf1 (IN)

オクテットで示したバッファの長さです。*buf1* パラメータがバイト数で指定され、*amtp* パラメータが文字で指定されている場合、この値は、CLOB および NCLOB の *amtp* 値 (*csfrm*=SQLCS_NCHAR) とは異なります。

ctxp (IN)

コールバック関数用のコンテキスト・ポインタです。NULL でもかまいません。

cbfp (IN)

各ピースに対してコールされるように登録できるコールバックです。これが NULL の場合は、ピースごとに OCI_NEED_DATA が戻ります。

コールバック関数は、OCI_CONTINUE を戻して読み込みを続行する必要があります。これ以外のエラー・コードが戻った場合、LOB の読み込みは強制終了します。

ctxp (IN)

コールバック関数用のコンテキストです。NULL でもかまいません。

bufp (IN/OUT)

ピース用のバッファ・ポインタです。

len (IN)

bufp におけるカレント・ピースのバイト長です。

piece (IN)

ピースは、OCI_FIRST_PIECE、OCI_NEXT_PIECE または OCI_LAST_PIECE です。

csid (IN)

バッファ・データのキャラクタ・セット ID です。この値が 0 (ゼロ) の場合は、*csfrm* の値に応じて、*csid* にクライアントの NLS_LANG 値または NLS_CHAR 値が設定されます。サーバーとクライアントの設定が同一である場合を除き、サーバーのキャラクタ・セットとはみなされません。

csfrm (IN)

バッファ・データのキャラクタ・セット・フォームです。 *csfrm* パラメータは、LOB の型と一貫している必要があります。

csfrm は、0（ゼロ）以外の次の 2 つの値をとることができます。

- SQLCS_IMPLICIT — データベース・キャラクタ・セット ID
- SQLCS_NCHAR — NCHAR キャラクタ・セット ID

デフォルト値は SQLCS_IMPLICIT です。 *csfrm* が指定されていない場合は、デフォルトが使用されます。

コメント

LOB/FILE の一部をコールの指定どおりにバッファに読み込みます。 NULL 属性の LOB または FILE から読み込むとエラーになります。

注意： LOB を読み込むかまたは書き込むときは、指定されたキャラクタ・セット・フォーム (*csfrm*) がロケータのフォームと一致している必要があります。

FILE の場合は、すでにオペレーティング・システム・ファイルがサーバー上に存在しており、それらのファイルは、入力ロケータを使用して OCILobFileOpen() または OCILobOpen() によってオープンされている必要があります。オペレーティング・システム・ファイルの読取り権限が Oracle にあること、およびディレクトリ・オブジェクトの読み許可がユーザーにあることが必要です。

OCILobRead() に対してポーリング・モードを使用するとき、ユーザーは最初のコールでは *offset* および *amtp* の値を指定する必要がありますが、その後の OCILobRead() のポーリング・コールでは指定する必要はありません。

LOB が BLOB の場合、*csid* パラメータおよび *csfrm* パラメータは無視されます。

注意： OCILobRead() 操作を終了して文ハンドルを解放するには、OCIBreak() コールを使用します。

次のルールは、CLOB および NCLOB のクライアント側可変幅キャラクタ・セットに適用されます。

- ポーリング・モードを使用する場合は、最初の OCILobRead() コールで *amtp* パラメータと *offset* パラメータを指定してください。後続のポーリング・コールでは、これらのパラメータは無視されます。
- コールバックを使用する場合、*len* パラメータはコールバックへの入力で、バッファ内で格納されたバイト数を示します。バッファの空き領域を確認するため、コールバック処理中に *len* パラメータをチェックします。

次のルールは、CLOB および NCLOB のクライアント側の固定幅キャラクタ・セットおよび可変幅キャラクタ・セットに適用されます。

- CLOB 値または NCLOB 値を読み込む場合は、`OCILobRead()` をコールするたびに *amtp* パラメータを参照し、バッファに格納された量を確認します。戻り値が文字数の場合（クライアント側のキャラクタ・セットが固定幅の場合）は、この値をバイトに変換して、バッファに格納された量を判断します。コールバックを使用する場合は、常に *len* パラメータをチェックしてバッファに格納された量を確認します。この値は、常にバイト単位です。

UTF-16 形式でデータを読み込むには、*csid* パラメータを `OCI_UTF16ID` に設定します。*csid* パラメータが設定された場合は、このパラメータによって環境変数 `NLS_LANG` が上書きされます。

関連項目：

- Unicode 形式の詳細は、5-43 ページの「[PL/SQL REF CURSOR および NESTED TABLE](#)」を参照してください。
- BFILE の詳細は、『[Oracle9i アプリケーション開発者ガイド - ラージ・オブジェクト](#)』の BFILE の説明を参照してください。
- LOB の読み込みおよび書き込み方法を示したコード例については、Oracle インストレーションに含まれているデモ・プログラムを参照してください。追加情報は、[付録 B「OCI デモ・プログラム」](#)を参照してください。
- 一般的なピース単位の OCI 操作については、5-44 ページの「[ランタイム・データ割当てとピース単位操作](#)」を参照してください。

関連関数

`OCLErrorGet()`、`OCILobWrite()`、`OCILobFileSetName()`、`OCILobWriteAppend()`

OCILobTrim()

用途

LOB 値を短い長さに切り捨てます。

構文

```
sword OCILobTrim ( OCISvcCtx      *svchp,  
                   OCIError       *errhp,  
                   OCILobLocator  *locp,  
                   ub4            newlen );
```

パラメータ

svchp (IN)

サービス・コンテキスト・ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCLErrorGet () に渡すエラー・ハンドルです。

locp (IN/OUT)

一意に LOB を参照する内部 LOB ロケータです。このロケータは、*svchp* に指定したサーバーから取得されたロケータにしてください。

newlen (IN)

LOB 値の新しい長さ（現行の長さ以下）です。文字の LOB の場合は、LOB 内の文字数です。バイナリの LOB および BFILE の場合は、LOB 内のバイト数です。

コメント

この関数では、LOB データを指定の長さに切り捨てます。*newlen* が現行の LOB の長さより長い場合は、エラーが戻されます。この関数は内部 LOB でのみ有効です。FILE には使用できません。

この LOB 操作をオープン・コールとクローズ・コールで囲む必要はありません。この操作を実行する前に LOB をオープンしていない場合、LOB 列のファンクション索引とドメイン索引はこのコール時に更新されます。この操作を実行する前に LOB をオープンしている場合は、トランザクションをコミットまたはロールバックする前に、その LOB をクローズする必要があります。内部 LOB をクローズすると、LOB 列のファンクション索引とドメイン索引が更新されます。

LOB 操作をオープン API とクローズ API で囲んでいない場合、ファンクション索引とドメイン索引は、LOB に書き込むたびに更新されます。これによって、パフォーマンスが低下する可能性があります。ファンクション索引とドメイン索引がある場合は、LOB への書込み操作をオープン / クローズ文内に含めることをお勧めします。

関連関数

`OCIErrorGet()`、`OCILobRead()`、`OCILobAppend()`、`OCILobCopy()`、
`OCILobErase()`、`OCILobWrite()`、`OCILobWriteAppend()`

OCILobWrite()

用途

バッファを LOB に書き込みます。

構文

```
sword OCILobWrite ( OCISvcCtx      *svchp,
                    OCIError       *errhp,
                    OCILobLocator  *locp,
                    ub4            *amtp,
                    ub4            *offset,
                    dvoid          *bufp,
                    ub4            *buflen,
                    ub1            *piece,
                    dvoid          *ctxp,
                    OCICallbackLobWrite (cbfp)
                    /*_
                    dvoid          *ctxp,
                    dvoid          *bufp,
                    ub4            *lenp,
                    ub1            *piecep */
                    ub2            csid,
                    ub1            csfrm );
```

パラメータ

svchp (IN/OUT)

サービス・コンテキスト・ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

locp (IN/OUT)

一意に LOB を参照する内部 LOB ロケータです。このロケータは、*svchp* に指定したサーバーから取得されたロケータにしてください。

amtp (IN/OUT)

次の表に示すように、*amtp* の値は、バイト数または文字数のいずれかで表される量です。

LOB/FILE	固定幅のクライアント側 キャラクタ・セットを 使用する入力	可変幅のクライアント側 キャラクタ・セットを 使用する入力	出力
BLOB および BFILE	バイト	バイト	バイト
CLOB および NCLOB	文字	バイト (1)	文字

(1) 入力量は、ユーザーが LOB に書き込むデータのバイト数で、*bufp* 内のバイト数ではありません。*bufp* 内のバイト数は、*buflen* によって指定されます。データがピースごとに読み込まれる場合、書き込むバイトの量は、*buflen* よりも大きくできます。出力量は、サーバー側の CLOB/NCLOB に書き込む文字数を指します。

このパラメータは常に NULL 以外のポインタです。EOF までの書込みを指定する場合は、変数を宣言してゼロを設定し、そのアドレスをこのパラメータに渡します。

入力時に量が指定されていて、データがピース単位で書き込まれる場合、**amtp* にはコール終了時（最後のピースが書き込まれた時点）に各ピースの長さの合計が格納され、書込み途中は未定義になります。ピース単位の読み込みとは異なることに注意してください。指定された量がサーバーに送信されない場合は、エラーが戻されます。

amtp が 0（ゼロ）の場合は、ストリーム・モードが使用され、ユーザーが OCI_LAST_PIECE を指定するまでデータが書き込まれます。

offset (IN)

入力時、これは LOB 値の先頭からの絶対オフセットです。文字 LOB では LOB の先頭からの文字数であり、バイナリ LOB ではバイト数です。先頭位置は 1 です。

ポーリングまたはコールバックによるストリームを使用する場合は、最初のコールでオフセットを指定します。後続のポーリング・コールでは、*offset* パラメータは無視されます。コールバックを使用する場合、*offset* パラメータはありません。

bufp (IN)

ピースの書込み元バッファへのポインタです。バッファ内のデータの長さが *buflen* に渡された値であるとみなされます。データがポーリング・メソッドを使用してピース単位で書き込まれる場合でも、*bufp* は、このコールが呼び出されたときは、LOB の最初のピースを格納する必要があります。コールバックが使用される場合、データの提供に *bufp* は使用できません。使用するとエラーになります。

buflen (IN)

バッファ内のデータの長さ（バイト数）です。*buflen* パラメータがバイト数で指定され、*amtp* パラメータが文字数で指定されている場合、この値は、CLOB および NCLOB の *amtp* 値とは異なります。

注意： このパラメータは8ビット（1バイト）を仮定します。現行のプラットフォームでこれより長いバイトを使用している場合は、*buflen* の値をそれにあわせて調整する必要があります。

piece (IN)

書き込み中のバッファのピースです。このパラメータのデフォルト値は、バッファが単独のピースとして書き込まれることを示す `OCI_ONE_PIECE` です。

ピース単位またはコールバック・モードで可能な他の値としては、`OCI_FIRST_PIECE`、`OCI_NEXT_PIECE` および `OCI_LAST_PIECE` があります。

ctxp (IN)

コールバック関数用のコンテキストです。NULL でもかまいません。

cbfp (IN)

ピース単位書き込みで各ピースに対してコールされるように登録できるコールバックです。これが NULL の場合は、標準ポーリング・メソッドが使用されます。

コールバック関数は、`OCI_CONTINUE` を戻して書き込みを続行する必要があります。これ以外のエラー・コードが戻った場合、LOB の書き込みは異常終了します。コールバックは、次のパラメータを取ります。

ctxp (IN)

コールバック関数用のコンテキストです。NULL でもかまいません。

bufp (IN/OUT)

ピース用のバッファ・ポインタです。このパラメータは、`OCILobWrite()` ルーチンに入力として渡される *bufp* と同じです。

lenp (IN/OUT)

buffer (IN) 内のデータのバイト数、および *bufp* (OUT) 内のカレント・ピースのバイト数です。

piecep (OUT)

ピースは `OCI_NEXT_PIECE` または `OCI_LAST_PIECE` です。

csid (IN)

バッファ内のデータのキャラクタ・セット ID です。この値が 0（ゼロ）の場合は、*csfrm* の値に応じて、*csid* にクライアントの `NLS_LANG` 値または `NLS_CHAR` 値が設定されます。

csfrm (IN)

バッファ・データのキャラクタ・セット・フォームです。 *csfrm* パラメータは、LOB の型と一貫している必要があります。

csfrm は、0 (ゼロ) 以外の次の 2 つの値をとることができます。

- SQLCS_IMPLICIT – データベース・キャラクタ・セット ID
- SQLCS_NCHAR – NCHAR キャラクタ・セット ID

デフォルト値は SQLCS_IMPLICIT です。

コメント

バッファを指定どおりに内部 LOB に書き込みます。LOB にすでにデータが存在する場合は、バッファに格納されたデータによって上書きされます。バッファは、このコールによって単独のピースとして LOB に書き込むことも、コールバックまたは標準ポーリング・メソッドを使用してピース単位で提供することもできます。

注意： LOB を読み込むかまたは書き込むときは、指定されたキャラクタ・セット・フォーム (*csfrm*) がロケータのフォームと一致している必要があります。

OCILOBWrite() に対してポーリング・モードを使用している場合、ユーザーは最初のコールでは *offset* および *amtp* の値を指定する必要がありますが、その後の OCILOBWrite() のコールでは指定する必要はありません。

piece パラメータの値が OCI_FIRST_PIECE である場合、データによっては、コールバックまたはポーリングを介して提供される必要があります。

コールバック関数が *cbfp* パラメータに定義されている場合は、パイプに 1 ピースが書き込まれるとこのコールバック関数が呼び出され、次のピースが取得されます。各ピースが *bufp* から書き込まれます。コールバック関数が定義されていない場合、OCILOBWrite() は、OCI_NEED_DATA エラー・コードを戻します。LOB のピースの書込みを続けるには、アプリケーションで OCILOBWrite() を再度コールする必要があります。このモードでは、ピースが別々のサイズで複数の場所に読み込まれる場合、コールごとにバッファ・ポインタと長さを変えることができます。

piece パラメータの値が OCI_LAST_PIECE のときは、ポーリングまたはコールバック方式が使用されているかどうかに関係なく、ピース単位書込み操作は終了します。

(どの入力方法を使用した場合も) Oracle に渡されるデータの量が、*amtp* パラメータに指定された量より少ない場合は、ORA-22993 エラーが戻されます。

この関数は内部 LOB でのみ有効です。FILE は読取り専用なので、取り扱えません。LOB が BLOB の場合、*csid* パラメータおよび *csfrm* パラメータは無視されます。

クライアント側のキャラクタ・セットが可変幅の場合、CLOB および NCLOB については、入力はバイト数で、出力は文字数で表されます。入力量は、ユーザーが LOB に書き込むデータのバイト数で、*bufp* 内のバイト数ではありません。*bufp* 内のバイト数は、*buflen* によって指定されます。データがピースごとに読み込まれる場合は、書き込むバイトの量は、*buflen* よりも大きくできます。出力量は、サーバー側の CLOB/NCLOB に書き込む文字数を指します。

UTF-16 形式でデータを書き込むには、*csid* パラメータを OCI_UTF16ID に設定します。*csid* パラメータが設定された場合は、このパラメータによって環境変数 NLS_LANG が上書きされます。

この LOB 操作をオープン・コールとクローズ・コールで囲む必要はありません。この操作を実行する前に LOB をオープンしていない場合、LOB 列のファンクション索引とドメイン索引はこのコール時に更新されます。この操作を実行する前に LOB をオープンしている場合は、トランザクションをコミットまたはロールバックする前に、その LOB をクローズする必要があります。内部 LOB をクローズすると、LOB 列のファンクション索引とドメイン索引が更新されます。

LOB 操作をオープン API とクローズ API で囲んでいない場合、ファンクション索引とドメイン索引は、LOB に書き込むたびに更新されます。これによって、パフォーマンスが低下する可能性があります。ファンクション索引とドメイン索引がある場合は、LOB への書き込み操作をオープン / クローズ文内に含めることをお勧めします。

関連項目：

- Unicode 形式の詳細は、5-43 ページの「[PL/SQL REF CURSOR および NESTED TABLE](#)」を参照してください。
- LOB の読み込みおよび書き込み方法を示したコード例については、Oracle インストレーションに含まれているデモ・プログラムを参照してください。追加情報は、[付録 B「OCI デモ・プログラム」](#)を参照してください。
- 一般的なピース単位の OCI 操作については、5-44 ページの「[ランタイム・データ割当てとピース単位操作](#)」を参照してください。

関連関数

[OCIErrorGet\(\)](#)、[OCILobRead\(\)](#)、[OCILobAppend\(\)](#)、[OCILobCopy\(\)](#)、[OCILobWriteAppend\(\)](#)

OCILobWriteAppend()

用途

LOB の末尾からデータの書き込みを開始します。

構文

```
sword OCILobWriteAppend ( OCISvcCtx *svchp,
                          OCIError *errhp,
                          OCILobLocator *locp,
                          ub4 *amtp,
                          dvoid *bufp,
                          ub4 buflen,
                          ub1 piece,
                          dvoid *ctxp,
                          OCICallbackLobWrite (cbfp)
                          (/*_
                           dvoid      *ctxp,
                           dvoid      *bufp,
                           ub4        *lenp,
                           ub1        *piecep */)
                          ub2 csid,
                          ub1 csfrm);
```

パラメータ

svchp (IN)

サービス・コンテキスト・ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

locp (IN/OUT)

一意に LOB を参照する内部 LOB ロケータです。

amtp (IN/OUT)

次の表に示すように、*amtp* の値は、バイト数または文字数のいずれかで表される量です。

LOB/FILE	固定幅のクライアント側 キャラクタ・セットを 使用する入力	可変幅のクライアント側 キャラクタ・セットを 使用する入力	出力
BLOB および BFILE	バイト	バイト	バイト
CLOB および NCLOB	文字	バイト (1)	文字

(1) 入力量は、ユーザーが LOB に書き込むデータのバイト数で、*bufp* 内のバイト数ではありません。*bufp* 内のバイト数は、*buflen* によって指定されます。データがピースごとに読み込まれる場合は、書き込むバイトの量は、*buflen* よりも大きくできます。出力量は、サーバー側の CLOB/NCLOB に書き込む文字数を指します。

入力時に量が指定されていて、データがピース単位で書き込まれる場合、**amtp* にはコール終了時（最後のピースが書き込まれた時点）に各ピースの長さの合計が格納され、書き込み途中は未定義になります（ピース単位の読み込みとは異なることに注意してください）。指定された量がサーバーに送信されない場合は、エラーが戻されます。*amtp* が 0（ゼロ）の場合、ストリーム・モードが仮定され、ユーザーが OCI_LAST_PIECE を指定するまでデータが書き込まれます。

クライアント側のキャラクタ・セットが可変幅の場合、CLOB/NCLOB の入力量は文字ではなくバイトで表されます。

bufp (IN)

ピースの書き込み元バッファへのポインタです。バッファ内のデータの長さが *buflen* に渡された値であるとみなされます。データがピース単位で書き込まれる場合でも、この関数が呼び出されたときは、*bufp* に LOB の最初のピースを格納する必要があります。コールバックが使用される場合、データの提供に *bufp* は使用できません。使用するとエラーになります。

buflen (IN)

バッファ内のデータの長さ（バイト数）です。このパラメータは 8 ビット（1 バイト）を仮定します。現行のプラットフォームでこれより長いバイトを使用している場合は、*buflen* の値をそれにあわせて調整する必要があります。

piece (IN)

書き込み中のバッファのピースです。このパラメータのデフォルト値は、バッファが単独のピースとして書き込まれることを示す OCI_ONE_PIECE です。ピース単位またはコールバック・モードで可能な他の値としては、OCI_FIRST_PIECE、OCI_NEXT_PIECE および OCI_LAST_PIECE があります。

ctxp (IN)

コールバック関数用のコンテキストです。NULL でもかまいません。

cbfp (IN)

ピース単位書込みで各ピースに対してコールされるように登録できるコールバックです。これが NULL の場合は、標準ポーリング・メソッドが使用されます。コールバック関数は、OCI_CONTINUE を戻して書込みを続行する必要があります。これ以外のエラー・コードが戻った場合、LOB の書込みは異常終了します。コールバックは、次のパラメータを取りま

ctxp (IN)

コールバック関数用のコンテキストです。NULL でもかまいません。

bufp (IN/OUT)

ピース用のバッファ・ポインタです。

lenp (IN/OUT)

buffer (IN) 内のデータのバイト数、および bufp (OUT) 内のカレント・ピースのバイト数です。

piecep (OUT)

ピースは OCI_NEXT_PIECE または OCI_LAST_PIECE です。

csid (IN)

バッファ・データのキャラクタ・セット ID です。

csfrm (IN)

バッファ・データのキャラクタ・セット・フォームです。

csfrm は、0（ゼロ）以外の次の 2 つの値をとることができます。

- SQLCS_IMPLICIT — データベース・キャラクタ・セット ID
- SQLCS_NCHAR — NCHAR キャラクタ・セット ID

デフォルト値は SQLCS_IMPLICIT です。

コメント

バッファは、このコールによって単独のピースとして LOB に書き込むことも、コールバックまたは標準ポーリング・メソッドを使用してピース単位で提供することもできます。piece パラメータの値が OCI_FIRST_PIECE である場合、データによっては、コールバックまたはポーリングを介して提供される必要があります。コールバック関数が cbfp パラメータに定義されている場合は、パイプに 1 ピースが書き込まれるとこのコールバック関数が呼び出され、次のピースが取得されます。各ピースが bufp から書き込まれます。コールバック関数が定義されていない場合、OCILobWriteAppend() は、OCI_NEED_DATA エラー・コードを戻します。

LOB のピースの書込みを続けるには、アプリケーションで `OCILobWriteAppend()` を再度コールする必要があります。このモードでは、ピースが別々のサイズで複数の場所に読み込まれる場合、コールごとにバッファ・ポインタと長さを変えることができます。piece パラメータの値が `OCI_LAST_PIECE` のときは、ピース単位の書込みは終了します。

LOB バッファリングが有効な場合、`OCILobWriteAppend()` はサポートされません。

LOB が BLOB の場合、`csid` パラメータおよび `csfrm` パラメータは無視されます。

クライアント側のキャラクタ・セットが可変幅の場合、CLOB/NCLOB の入力量は文字ではなくバイトで表されます。

この LOB 操作をオープン・コールとクローズ・コールで囲む必要はありません。この操作を実行する前に LOB をオープンしていない場合、LOB 列のファンクション索引とドメイン索引はこのコール時に更新されます。この操作を実行する前に LOB をオープンしている場合は、トランザクションをコミットまたはロールバックする前に、その LOB をクローズする必要があります。内部 LOB をクローズすると、LOB 列のファンクション索引とドメイン索引が更新されます。

LOB 操作をオープン API とクローズ API で囲んでいない場合、ファンクション索引とドメイン索引は、LOB に書き込むたびに更新されます。これによって、パフォーマンスが低下する可能性があります。ファンクション索引とドメイン索引がある場合は、LOB への書込み操作をオープン / クローズ文内に含めることをお勧めします。

関連項目： 7-10 ページ [「LOB の読み / 書き込みパフォーマンスを向上するための関数」](#)

関連関数

`OCIErrorGet()`、`OCILobRead()`、`OCILobAppend()`、`OCILobCopy()`、`OCILobWrite()`

アドバンスト・キューイング関数およびパブリッシュ・サブスクライブ関数

この項では、アドバンスト・キューイング関数およびパブリッシュ・サブスクライブ関数について説明します。

表 16-3 アドバンスト・キューイング関数およびパブリッシュ・サブスクライブ関数

関数	用途
OCIAQDeq() (16-86 ページ)	アドバンスト・キューイング・デキュー
OCIAQEnq() (16-88 ページ)	アドバンスト・キューイング・エンキュー
OCIAQListen() (16-100 ページ)	リストのエージェントの代理として 1 つ以上のキューをリスニングします。
OCISubscriptionEnable() (16-103 ページ)	サブスクリプションについての通知を使用可能にします。
OCISubscriptionPost() (16-104 ページ)	通知を受信するようにサブスクリプションに転記します。
OCISubscriptionRegister() (16-106 ページ)	サブスクリプションを登録します。
OCISubscriptionUnRegister() (16-109 ページ)	サブスクリプションの登録を解除します。

OCIAQDeq()

用途

このコールは、OCI を使用したアドバンスト・キューイングのデキュー操作で使します。

構文

```
sword OCIAQDeq ( OCISvcCtx      *svch,
                  OCIError      *errh,
                  text           *queue_name,
                  OCIAQDeqOptions *dequeue_options,
                  OCIAQMsgProperties *message_properties,
                  OCIType        *payload_tdo,
                  dvoid          **payload,
                  dvoid          **payload_ind,
                  OCIRaw         **msgid,
                  ub4            flags );
```

パラメータ

svch (IN)

OCI サービス・コンテキストです。

errh (IN)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

queue_name (IN)

デキュー操作のためのターゲット・キューです。

dequeue_options (IN)

デキュー操作のオプションです。OCIAQDeqOptions 記述子に格納されます。

message_properties (OUT)

メッセージ用のメッセージ・プロパティです。OCIAQMsgProperties 記述子に格納されています。

payload_tdo (IN)

オブジェクト型の TDO (型記述子オブジェクト) です。RAW キューでは、このパラメータが SYS.RAW の TDO を指し示す必要があります。

payload (IN/OUT)

オブジェクト型のインスタンスであるプログラム変数バッファのポインタへのポインタです。RAW キューでは、このパラメータが OCIRaw のインスタンスを指し示す必要があります。

payload 用のメモリーは、オブジェクト・キャッシュに動的に割り当てられます。payload インスタンスが不要になったときは、アプリケーションから `OCIObjectFree()` をコールし、割当て解除もできます。プログラム変数バッファへのポインタ (`*payload`) が NULL で渡された場合、そのバッファは暗黙的にキャッシュに割り当てられます。

アプリケーションでは、`OCIAQDeq()` が最初にコールされたときに payload に NULL を渡し、その payload に、OCI でメモリーを割り当てるようにできます。後続の `OCIAQDeq()` コールでは、前に割り当てられたメモリーへのポインタが使用されます。

payload 用に TDO を取得するには、`OCITypeByName()` または `OCITypeByRef()` を使用します。

OCI によって、ユーザーが payload の属性 (テキストなど) を設定できる関数が提供されます。これらの属性の設定については、10-13 ページの「[オブジェクト属性の操作](#)」を参照してください。

payload_ind (IN/OUT)

オブジェクト型のパラレル・インジケータ情報構造を含むプログラム変数バッファ・ポインタへのポインタです。

payload_ind へのメモリー割当てルールは、前述の payload に対するルールと同じです。

msgid (OUT)

メッセージ ID です。

flags (IN)

現行では使用されていません。OCI_DEFAULT として渡されます。

コメント

このコールを使用するには、ユーザーが `aq_user_role` または `dbms_aq` パッケージを実行する権限を所有している必要があります。このコールを使用するには、OCI 環境をオブジェクト・モードで初期化する必要があります (`OCIInitialize()` を使用)。

関連項目：

- OCI およびアドバンスト・キューイングの詳細は、9-48 ページの「[OCI およびアドバンスト・キューイング](#)」を参照してください。
- アドバンスト・キューイングの追加情報は、『Oracle9i アプリケーション開発者ガイド-アドバンスト・キューイング』を参照してください。

例

コード例は、16-88 ページの「[OCIAQEnq\(\)](#)」の説明を参照してください。

関連関数

[OCIAQEnq\(\)](#)、[OCIAQListen\(\)](#)、[OCIInitialize\(\)](#)

OCIAQEnq()

用途

このコールは、アドバンスト・キューイング・エンキューに使用します。

構文

```
sword OCIAQEnq ( OCISvcCtx      *svch,  
                 OCIError       *errh,  
                 text            *queue_name,  
                 OCIAQEnqOptions *enqueue_options,  
                 OCIAQMsgProperties *message_properties,  
                 OCIType         *payload_tdo,  
                 dvoid           **payload,  
                 dvoid           **payload_ind,  
                 OCIRaw          **msgid,  
                 ub4             flags );
```

パラメータ

svch (IN)

OCI サービス・コンテキストです。

errh (IN)

エラー発生時の診断情報のために `OCLErrorGet()` に渡すエラー・ハンドルです。

queue_name (IN)

エンキュー操作のためのターゲット・キューです。

enqueue_options (IN)

エンキュー操作のオプション。**OCIAQEnqOptions** 記述子に格納されています。

message_properties (IN)

メッセージ用のメッセージ・プロパティです。**OCIAQMsgProperties** 記述子に格納されています。

payload_tdo (IN)

オブジェクト型の TDO（型記述子オブジェクト）です。RAW キューでは、このパラメータが `SYS.RAW` の TDO を指し示す必要があります。

payload (IN)

オブジェクト型のインスタンスを指し示すポインタへのポインタです。RAW キューでは、このパラメータが **OCIRaw** のインスタンスを指し示す必要があります。

OCI によって、ユーザーが `payload` の属性（テキストなど）を設定できる関数が提供されます。

関連項目： これらの属性の設定については、10-13 ページの「[オブジェクト属性の操作](#)」を参照してください。

payload_ind (IN)

オブジェクト型のパラレル・インジケータ情報構造を含むプログラム変数バッファ・ポインタへのポインタです。

msgid (OUT)

メッセージ ID です。

flags (IN)

現行では使用されていません。OCI_DEFAULT として渡されます。

コメント

このコールを使用するには、ユーザーが `aq_user_role` または `dbms_aq` パッケージを実行する権限を所有する必要があります。

このコールを使用するには、OCI 環境をオブジェクト・モードで初期化する必要があります (`OCIInitialize()` を使用)。

関連項目：

- OCI およびアドバンスト・キューイングの詳細は、9-48 ページの「[OCI およびアドバンスト・キューイング](#)」を参照してください。
- アドバンスト・キューイングの詳細は、『Oracle9i アプリケーション開発者ガイド-アドバンスト・キューイング』を参照してください。

payload 用に TDO を取得するには、`OCITypeByName()` または `OCITypeByRef()` を使用します。

例

次に 4 つの異なる状況を想定し、それぞれについて `OCIAQEnq()` と `OCIAQDeq()` を使用する方法を説明します。

関連項目： これらの例では、データベースが、『Oracle9i アプリケーション開発者ガイド-アドバンスト・キューイング』の「Oracle アドバンスト・キューイングの使用例」にある説明どおりに設定されているものと想定しています。

例 1 – payload オブジェクトのエンキューおよびデキュー

```
struct message
{
    OCIStrng    *subject;
    OCIStrng    *data;
};
typedef struct message message;

struct null_message
{
    OCIInd      null_adt;
    OCIInd      null_subject;
    OCIInd      null_data;
};
typedef struct null_message null_message;

int main()
{
    OCIEnv *envhp;
    OCIServer *srvhp;
    OCIError *errhp;
    OCISvcCtx *svchp;
    dvoid *tmp;
    OCIType *mesg_tdo = (OCIType *) 0;
    message msg;
    null_message nmsg;
    message *mesg = &msg;
    null_message *nmesg = &nmsg;
    message *deqmesg = (message *)0;
    null_message *ndeqmesg = (null_message *)0;

    OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0, (dvoid * (*)()) 0,
                  (dvoid * (*)()) 0, (void (*)()) 0 );

    OCIHandleAlloc( (dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
                    52, (dvoid **) &tmp);

    OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp );

    OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
                    52, (dvoid **) &tmp);
    OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
                    52, (dvoid **) &tmp);

    OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);
```

```

OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
                52, (dvoid **) &tmp);
OCIAttrSet( (dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *)srvhp, (ub4) 0,
            (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);

OCILogon(envhp, errhp, &svchp, "AQ", strlen("AQ"), "AQ", strlen("AQ"), 0, 0);

/* obtain TDO of message_type */
OCITypeByName(envhp, errhp, svchp, (CONST text *)"AQ", strlen("AQ"),
              (CONST text *)"MESSAGE_TYPE", strlen("MESSAGE_TYPE"),
              (text *)0, 0, OCI_DURATION_SESSION, OCI_TYPEGET_ALL, &mesg_tdo);

/* prepare the message payload */
mesg->subject = (OCIString *)0;
mesg->data = (OCIString *)0;
OCIStringAssignText(envhp, errhp, (CONST text *)"NORMAL MESSAGE",
                    strlen("NORMAL MESSAGE"), &mesg->subject);
OCIStringAssignText(envhp, errhp, (CONST text *)"OCI ENQUEUE",
                    strlen("OCI ENQUEUE"), &mesg->data);
nmesg->null_adt = nmesg->null_subject = nmesg->null_data = OCI_IND_NOTNULL;

/* enqueue into the msg_queue */
OCIAQEnq(svchp, errhp, (CONST text *)"msg_queue", 0, 0,
          mesg_tdo, (dvoid **)&mesg, (dvoid **)&nmesg, 0, 0);
OCITransCommit(svchp, errhp, (ub4) 0);

/* dequeue from the msg_queue */
OCIAQDeq(svchp, errhp, (CONST text *)"msg_queue", 0, 0,
          mesg_tdo, (dvoid **)&deqmesg, (dvoid **)&ndeqmesg, 0, 0);
printf("Subject: %s\n", OCIStringPtr(envhp, deqmesg->subject));
printf("Text: %s\n", OCIStringPtr(envhp, deqmesg->data));
OCITransCommit(svchp, errhp, (ub4) 0);
}

```

例 2 – 関連 ID を使用したエンキューおよびデキュー

```

struct message
{
    OCIString    *subject;
    OCIString    *data;
};

typedef struct message message;

struct null_message
{
    OCIInd      null_adt;
    OCIInd      null_subject;
    OCIInd      null_data;
};

```

```
};
typedef struct null_message null_message;

int main()
{
    OCIEnv *envhp;
    OCIServer *srvhp;
    OCIError *errhp;
    OCISvcCtx *svchp;
    dvoid *tmp;
    OCIType *mesg_tdo = (OCIType *) 0;
    message msg;
    null_message nmsg;
    message *mesg = &msg;
    null_message *nmesg = &nmsg;
    message *deqmesg = (message *)0;
    null_message *ndeqmesg = (null_message *)0;
    OCIRaw*firstmsg = (OCIRaw *)0;
    OCIAQMsgProperties *msgprop = (OCIAQMsgProperties *)0;
    OCIAQDeqOptions *deqopt = (OCIAQDeqOptions *)0;
    text correlation1[30], correlation2[30];

    OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0, (dvoid * (*)()) 0,
                  (dvoid * (*)()) 0, (void (*)()) 0 );
    OCIHandleAlloc( (dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
                    52, (dvoid **) &tmp);

    OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp );

    OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
                    52, (dvoid **) &tmp);
    OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
                    52, (dvoid **) &tmp);

    OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

    OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
                    52, (dvoid **) &tmp);
    OCIAttrSet( (dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *)srvhp, (ub4) 0,
                (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);

    OCILogon(envhp, errhp, &svchp, "AQ", strlen("AQ"), "AQ", strlen("AQ"), 0, 0);

    /* allocate message properties descriptor */
    OCIDescriptorAlloc(envhp, (dvoid **)&msgprop,
                       OCI_DTYPE_AQMSG_PROPERTIES, 0, (dvoid **)0);
    strcpy(correlation1, "1st message");
```

```
OCIAttrSet(msgprop, OCI_DTYPE_AQMSG_PROPERTIES, (dvoid *)&correlation1,
            strlen(correlation1), OCI_ATTR_CORRELATION, errhp);

/* obtain TDO of message_type */
OCITypeByName(envhp, errhp, svchp, (CONST text *)"AQ", strlen("AQ"),
              (CONST text *)"MESSAGE_TYPE", strlen("MESSAGE_TYPE"),
              (text *)0, 0, OCI_DURATION_SESSION, OCI_TYPEGET_ALL, &mesg_tdo);

/* prepare the message payload */
mesg->subject = (OCIStrng *)0;
mesg->data = (OCIStrng *)0;
OCIStrngAssignText(envhp, errhp, (CONST text *)"NORMAL ENQUEUE1",
                  strlen("NORMAL ENQUEUE1"), &mesg->subject);
OCIStrngAssignText(envhp, errhp, (CONST text *)"OCI ENQUEUE",
                  strlen("OCI ENQUEUE"), &mesg->data);
nmesg->null_adt = nmesg->null_subject = nmesg->null_data = OCI_IND_NOTNULL;

/* enqueue into the msg_queue, store the message id into firstmsg */
OCIAQEnq(svchp, errhp, (CONST text *)"msg_queue", 0, msgprop,
         mesg_tdo, (dvoid **)&mesg, (dvoid **)&nmesg, &firstmsg, 0);

/* enqueue into the msg_queue with a different correlation id */
strcpy(correlation2, "2nd message");
OCIAttrSet(msgprop, OCI_DTYPE_AQMSG_PROPERTIES, (dvoid *)&correlation2,
            strlen(correlation2), OCI_ATTR_CORRELATION, errhp);
OCIStrngAssignText(envhp, errhp, (CONST text *)"NORMAL ENQUEUE2",
                  strlen("NORMAL ENQUEUE2"), &mesg->subject);
OCIAQEnq(svchp, errhp, (CONST text *)"msg_queue", 0, msgprop,
         mesg_tdo, (dvoid **)&mesg, (dvoid **)&nmesg, 0, 0);

OCITransCommit(svchp, errhp, (ub4) 0);

/* first dequeue by correlation id "2nd message" */
/* allocate dequeue options descriptor and set the correlation option */
OCIDescriptorAlloc(envhp, (dvoid **)&deqopt,
                  OCI_DTYPE_AQDEQ_OPTIONS, 0, (dvoid **)&0);
OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (dvoid *)&correlation2,
            strlen(correlation2), OCI_ATTR_CORRELATION, errhp);

/* dequeue from the msg_queue */
OCIAQDeq(svchp, errhp, (CONST text *)"msg_queue", deqopt, 0,
         mesg_tdo, (dvoid **)&deqmesg, (dvoid **)&ndeqmesg, 0, 0);
printf("Subject: %s\n", OCIStrngPtr(envhp, deqmesg->subject));
printf("Text: %s\n", OCIStrngPtr(envhp, deqmesg->data));
OCITransCommit(svchp, errhp, (ub4) 0);
```

```
/* second dequeue by message id */
OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (dvoid *)&firstmsg,
OCIRawSize(envhp, firstmsg), OCI_ATTR_DEQ_MSGID, errhp);
/* clear correlation id option */
OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS,
            (dvoid *)correlation2, 0, OCI_ATTR_CORRELATION, errhp);

/* dequeue from the msg_queue */
OCIAQDeq(svchp, errhp, (CONST text *)"msg_queue", deqopt, 0,
        msg_tdo, (dvoid **)&deqmesg, (dvoid **)&ndeqmesg, 0, 0);
printf("Subject: %s\n", OCIStrPtr(envhp, deqmesg->subject));
printf("Text: %s\n", OCIStrPtr(envhp, deqmesg->data));
OCITransCommit(svchp, errhp, (ub4) 0);
}
```

例 3 – RAW キューのエンキューおよびデキュー

```
int main()
{
    OCIEnv *envhp;
    OCIServer *srvhp;
    OCIError *errhp;
    OCISvcCtx *svchp;
    dvoid *tmp;
    OCIType *msg_tdo = (OCIType *) 0;
    char msg_text[100];
    OCIRaw *mesg = (OCIRaw *) 0;
    OCIRaw*deqmesg = (OCIRaw *) 0;
    OCIInd ind = 0;
    dvoid *indptr = (dvoid *)&ind;
    inti;

    OCIInitialize((ub4) OCI_OBJECT, (dvoid *) 0, (dvoid * (*)()) 0,
                  (dvoid * (*)()) 0, (void (*)()) 0 );
    OCIHandleAlloc( (dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
                    52, (dvoid **) &tmp);

    OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp );

    OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
                    52, (dvoid **) &tmp);
    OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
                    52, (dvoid **) &tmp);

    OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

    OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
                    52, (dvoid **) &tmp);
```

```

OCIAttrSet( (dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *)srvhp, (ub4) 0,
            (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);

OCILogon(envhp, errhp, &svchp, "AQ", strlen("AQ"), "AQ", strlen("AQ"), 0, 0);

/* obtain the TDO of the RAW data type */
OCITypeByName(envhp, errhp, svchp, (CONST text *)"SYS", strlen("SYS"),
            (CONST text *)"RAW", strlen("RAW"),
            (text *)0, 0, OCI_DURATION_SESSION, OCI_TYPEGET_ALL, &mesg_tdo);

/* prepare the message payload */
strcpy(msg_text, "Enqueue to a RAW queue");
OCIRawAssignBytes(envhp, errhp, msg_text, strlen(msg_text), &mesg);

/* enqueue the message into raw_msg_queue */
OCIAQEnq(svchp, errhp, (CONST text *)"raw_msg_queue", 0, 0,
        mesg_tdo, (dvoid **)&mesg, (dvoid **)&indp, 0, 0);
OCITransCommit(svchp, errhp, (ub4) 0);

/* dequeue the same message into C variable deqmesg */
OCIAQDeq(svchp, errhp, (CONST text *)"raw_msg_queue", 0, 0,
        mesg_tdo, (dvoid **)&deqmesg, (dvoid **)&indp, 0, 0);
for (i = 0; i < OCIRawSize(envhp, deqmesg); i++)
    printf("%c", *(OCIRawPtr(envhp, deqmesg) + i));
OCITransCommit(svchp, errhp, (ub4) 0);
}

```

例 4 – OCIAQAgent を使用したエンキューおよびデキュー

```

struct message
{
    OCIStrng    *subject;
    OCIStrng    *data;
};
typedef struct message message;

struct null_message
{
    OCIInd      null_adt;
    OCIInd      null_subject;
    OCIInd      null_data;
};
typedef struct null_message null_message;

```

```
int main()
{
    OCIEnv *envhp;
    OCIServer *srvhp;
    OCIError *errhp;
    OCISvcCtx *svchp;
    dvoid *tmp;
    OCIType *mesg_tdo = (OCIType *) 0;
    message msg;
    null_message nmsg;
    message *mesg = &msg;
    null_message *nmesg = &nmsg;
    message *deqmesg = (message *)0;
    null_message *ndeqmesg = (null_message *)0;
    OCIAQMsgProperties *msgprop = (OCIAQMsgProperties *)0;
    OCIAQAgent *agents[2];
    OCIAQDeqOptions *deqopt = (OCIAQDeqOptions *)0;
    ub4wait = OCI_DEQ_NO_WAIT;
    ub4 navigation = OCI_DEQ_FIRST_MSG;

    OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0, (dvoid * (*)()) 0,
                  (dvoid * (*)()) 0, (void (*)()) 0 );

    OCIHandleAlloc( (dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
                    52, (dvoid **) &tmp);

    OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp );

    OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
                    52, (dvoid **) &tmp);
    OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
                    52, (dvoid **) &tmp);

    OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

    OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
                    52, (dvoid **) &tmp);

    OCIAttrSet( (dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *)srvhp, (ub4) 0,
                (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);

    OCILogon(envhp, errhp, &svchp, "AQ", strlen("AQ"), "AQ", strlen("AQ"), 0, 0);
```



```
/* obtain TDO of message_type */
OCITypeByName(envhp, errhp, svchp, (CONST text *)"AQ", strlen("AQ"),
              (CONST text *)"MESSAGE_TYPE", strlen("MESSAGE_TYPE"),
              (text *)0, 0, OCI_DURATION_SESSION, OCI_TYPEGET_ALL, &mesg_tdo);

/* prepare the message payload */
mesg->subject = (OCIString *)0;
mesg->data = (OCIString *)0;
OCIStringAssignText(envhp, errhp,
                   (CONST text *)"MESSAGE 1", strlen("MESSAGE 1"),
                   &mesg->subject);
OCIStringAssignText(envhp, errhp,
                   (CONST text *)"mesg for queue subscribers",
                   strlen("mesg for queue subscribers"), &mesg->data);
nmesg->null_adt = nmesg->null_subject = nmesg->null_data = OCI_IND_NOTNULL;

/* enqueue MESSAGE 1 for subscribers to the queue that is for RED and GREEN */
OCIAQEnq(svchp, errhp, (CONST text *)"msg_queue_multiple", 0, 0,
         mesg_tdo, (dvoid **)&mesg, (dvoid **)&nmesg, 0, 0);

/* enqueue MESSAGE 2 for specified recipients that is for RED and BLUE */
/* prepare message payload */
OCIStringAssignText(envhp, errhp,
                   (CONST text *)"MESSAGE 2", strlen("MESSAGE 2"),
                   &mesg->subject);
OCIStringAssignText(envhp, errhp,
                   (CONST text *)"mesg for two recipients",
                   strlen("mesg for two recipients"), &mesg->data);

/* allocate AQ message properties and agent descriptors */
OCIDescriptorAlloc(envhp, (dvoid **)&msgprop,
                  OCI_DTYPE_AQMSG_PROPERTIES, 0, (dvoid **)&0);
OCIDescriptorAlloc(envhp, (dvoid **)&agents[0],
                  OCI_DTYPE_AQAGENT, 0, (dvoid **)&0);
OCIDescriptorAlloc(envhp, (dvoid **)&agents[1],
                  OCI_DTYPE_AQAGENT, 0, (dvoid **)&0);

/* prepare the recipient list, RED and BLUE */
OCIAttrSet(agents[0], OCI_DTYPE_AQAGENT, "RED", strlen("RED"),
          OCI_ATTR_AGENT_NAME, errhp);
OCIAttrSet(agents[1], OCI_DTYPE_AQAGENT, "BLUE", strlen("BLUE"),
          OCI_ATTR_AGENT_NAME, errhp);
OCIAttrSet(msgprop, OCI_DTYPE_AQMSG_PROPERTIES, (dvoid *)agents, 2,
          OCI_ATTR_RECIPIENT_LIST, errhp);
```

```
OCIAQEnq(svchp, errhp, (CONST text *)"msg_queue_multiple", 0, msgprop,
          msg_tdo, (dvoid **)&mesg, (dvoid **)&nmesg, 0, 0);
OCITransCommit(svchp, errhp, (ub4) 0);

/* now dequeue the messages using different consumer names */
/* allocate dequeue options descriptor to set the dequeue options */
OCIDescriptorAlloc(envhp, (dvoid **)&deqopt, OCI_DTYPE_AQDEQ_OPTIONS, 0,
                   (dvoid **)0);

/* set wait parameter to NO_WAIT so that the dequeue returns immediately */
OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (dvoid *)&wait, 0,
           OCI_ATTR_WAIT, errhp);

/* set navigation to FIRST_MESSAGE so that the dequeue resets the position */
/* after a new consumer_name is set in the dequeue options */
OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (dvoid *)&navigation, 0,
           OCI_ATTR_NAVIGATION, errhp);

/* dequeue from the msg_queue_multiple as consumer BLUE */
OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (dvoid *)"BLUE", strlen("BLUE"),
           OCI_ATTR_CONSUMER_NAME, errhp);
while (OCIAQDeq(svchp, errhp, (CONST text *)"msg_queue_multiple", deqopt, 0,
               msg_tdo, (dvoid **)&deqmesg, (dvoid **)&ndeqmesg, 0, 0)
== OCI_SUCCESS)
{
    printf("Subject: %s\n", OCIStrPtr(envhp, deqmesg->subject));
    printf("Text: %s\n", OCIStrPtr(envhp, deqmesg->data));
}
OCITransCommit(svchp, errhp, (ub4) 0);

/* dequeue from the msg_queue_multiple as consumer RED */
OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (dvoid *)"RED", strlen("RED"),
           OCI_ATTR_CONSUMER_NAME, errhp);
while (OCIAQDeq(svchp, errhp, (CONST text *)"msg_queue_multiple", deqopt, 0,
               msg_tdo, (dvoid **)&deqmesg, (dvoid **)&ndeqmesg, 0, 0)
== OCI_SUCCESS)
{
    printf("Subject: %s\n", OCIStrPtr(envhp, deqmesg->subject));
    printf("Text: %s\n", OCIStrPtr(envhp, deqmesg->data));
}
OCITransCommit(svchp, errhp, (ub4) 0);
```

```
/* dequeue from the msg_queue_multiple as consumer GREEN */
OCIAttrSet(degopt, OCI_DTYPE_AQDEQ_OPTIONS, (dvoid *) "GREEN", strlen("GREEN"),
           OCI_ATTR_CONSUMER_NAME, errhp);
while (OCIAQDeq(svchp, errhp, (CONST text *) "msg_queue_multiple", degopt, 0,
               msg_tdo, (dvoid **)&degmesg, (dvoid **)&ndeqmesg, 0, 0)
== OCI_SUCCESS)
{
    printf("Subject: %s\n", OCIStrPtr(envhp, deqmesg->subject));
    printf("Text: %s\n", OCIStrPtr(envhp, deqmesg->data));
}
OCITransCommit(svchp, errhp, (ub4) 0);
}
```

関連関数

[OCIAQDeq\(\)](#)、[OCIAQListen\(\)](#)、[OCIInitialize\(\)](#)

OCIAQListen()

用途

リストのエージェントのかわりに 1 つ以上のキューをリスニングします。

構文

```
sword OCIAQListen (OCISvcCtx      *svchpp,
                  OCIError        *errhp,
                  OCIAQAgent      **agent_list,
                  ub4             num_agents,
                  sb4             wait,
                  OCIAQAgent      **agent,
                  ub4             flags);
```

パラメータ

svchpp (IN/OUT)

サービス・コンテキスト・ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCLErrorGet () に渡すエラー・ハンドルです。

agent_list (IN)

メッセージを監視するエージェントのリストです。

num_agents (IN)

エージェント・リスト内のエージェント数です。

wait (IN)

リスニング・コールのタイムアウトです。

agent (OUT)

メッセージの出力先のエージェントです。OCIAgent は OCI 記述子です。

flags (IN)

現行では使用されていません。OCI_DEFAULT として渡されます。

コメント

メッセージが存在し、リスト内のエージェントに対してそのメッセージを使用する準備ができたときに戻されるブロッキング・コールです。待機時間が終了してもメッセージが検出されない場合は、エラーが戻されます。

関連関数

[OCIAQEnq\(\)](#)、[OCIAQDeq\(\)](#)、[OCISvcCtxToLda\(\)](#)、[OCISubscriptionEnable\(\)](#)、
[OCISubscriptionPost\(\)](#)、[OCISubscriptionRegister\(\)](#)、
[OCISubscriptionUnRegister\(\)](#)

OCISubscriptionDisable()

用途

サブスクリプションの登録を使用禁止にして、すべての通知をオフにします。

構文

```
ub4 OCISubscriptionDisable ( OCISubscription  *subscrhp,  
                             OCIError          *errhp  
                             ub4                mode );
```

パラメータ

subscrhp (IN)

OCI_ATTR_SUBSCR_NAME および OCI_ATTR_SUBSCR_NAMESPACE 属性が設定されたサブスクリプション・ハンドルです。

関連項目： 詳細は、A-54 ページの「[サブスクリプション・ハンドル属性](#)」を参照してください。

errhp (OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

mode (IN)

コール固有モードです。有効な値は次のとおりです。

- OCI_DEFAULT — デフォルトのコールを実行します。サブスクリプションが使用可能になるまで、このサブスクリプションの通知をすべて破棄します。

コメント

このコールは、一時的に通知をオフにするときに使用します。コードの重要なセクションを実行しているときに使用して、アプリケーションが中断されないようにします。

この操作をするときは、接続または認証は必要ありません。このコールの前に、サブスクリプション・ハンドルによって指定されたサブスクリプションの登録を行ってください。

OCISubscriptionDisable() を実行した後は、OCISubscriptionEnable() を実行するまですべての通知が破棄されます。

関連関数

[OCIAQListen\(\)](#)、[OCISubscriptionEnable\(\)](#)、[OCISubscriptionPost\(\)](#)、[OCISubscriptionRegister\(\)](#)、[OCISubscriptionUnRegister\(\)](#)

OCISubscriptionEnable()

用途

使用禁止にしたサブスクリプションの登録を使用可能にします。すべての通知がオンになります。

構文

```
ub4 OCISubscriptionEnable ( OCISubscription  *subscrhp,  
                             OCIError        *errhp  
                             ub4              mode );
```

パラメータ

subscrhp (IN)

OCI_ATTR_SUBSCR_NAME および OCI_ATTR_SUBSCR_NAMESPACE 属性が設定されたサブスクリプション・ハンドルです。

関連項目： 詳細は、A-54 ページの「[サブスクリプション・ハンドル属性](#)」を参照してください。

errhp (OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

mode (IN)

コール固有モードです。有効な値は次のとおりです。

- OCI_DEFAULT – デフォルトのコールを実行します。次の使用可能化が実行されるまで、このサブスクリプションの通知はすべてバッファリングされます。

コメント

このコールは、サブスクリプションの登録を使用禁止にした後に、通知をオンにするときに使用します。

この操作をするときは、接続または認証は必要ありません。このコールを実行する前に、指定されたサブスクリプションの登録を行ってください。

関連関数

[OCIAQListen\(\)](#)、[OCISvcCtxToLda\(\)](#)、[OCISubscriptionPost\(\)](#)、[OCISubscriptionRegister\(\)](#)、[OCISubscriptionUnRegister\(\)](#)

OCISubscriptionPost()

用途

サブスクリプションに転記し、そのサブスクリプションに対して登録されたすべてのクライアントが通知を受信できるようにします。

構文

```
ub4 OCISubscriptionPost ( OCISvcCtx          *svchp,
                          OCISubscription    **subscrhpp,
                          ub2                  count,
                          OCIError           *errhp,
                          ub4                  mode );
```

パラメータ

svchp (IN)

OCI サービス・コンテキストです（バージョン7より後）。このサービス・コンテキストには、有効な認証されたユーザー・ハンドルが必要です。

subscrhpp (IN)

サブスクリプション・ハンドルの配列です。この配列の各要素は、OCI_ATTR_SUBSCR_NAME および OCI_ATTR_SUBSCR_NAMESPACE 属性が設定されたサブスクリプション・ハンドルにしてください。

関連項目： 詳細は、A-54 ページの「[サブスクリプション・ハンドル属性](#)」を参照してください。

このコールの前に、各サブスクリプション・ハンドルに対して OCI_ATTR_SUBSCR_PAYLOAD 属性を設定する必要があります。設定されていない場合、ペイロードは NULL であるとみなされ、重要な登録を行ったクライアントが通知を受信したときに、ペイロードが配布されません。OCIAttrSet () コールでは、ペイロードへの参照は記録されますが、内容はコピーされないため、コール元は転送が行われるまでペイロードを保存する必要があります。

count (IN)

サブスクリプション・ハンドル配列の要素数です。

errhp (OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

mode (IN)

コール固有モードです。有効な値は次のとおりです。

- OCL_DEFAULT – デフォルトのコールを実行します。

コメント

サブスクリプションに転記されるときに、必要に応じてサブスクリプション名およびペイロードの識別を行うことができます。ペイロードが関連付けられない場合は、ペイロードの長さは 0 に設定されます。

このコールでは、ベストエフォート型の保証を提供しています。通知は、登録されたクライアントに対して最大で 1 回行われます。

このコールは、主に軽量の通知に使用し、複数のシステム・イベントが発生する場合に役立ちます。アプリケーションでより確実な保証が必要な場合は、アドバンスト・キューイング機能を使用し、キューにエンキューします。

関連関数

[OCIAQListen\(\)](#)、[OCISvcCtxToLda\(\)](#)、[OCISubscriptionEnable\(\)](#)、[OCISubscriptionRegister\(\)](#)、[OCISubscriptionUnRegister\(\)](#)

OCISubscriptionRegister()

用途

メッセージ通知に対するコールバックを登録します。

構文

```
ub4 OCISubscriptionRegister ( OCISvcCtx          *svchp,
                              OCISubscription **subscrhpp,
                              ub2              count,
                              OCIError        *errhp,
                              ub4              mode );
```

パラメータ

svchp (IN)

OCI サービス・コンテキストです（バージョン7より後）。このサービス・コンテキストには、有効な認証されたユーザー・ハンドルが必要です。

subscrhpp (IN)

サブスクリプション・ハンドルの配列です。この配列の各要素は、次のすべての属性が設定されたサブスクリプション・ハンドルである必要があります。

- OCI_ATTR_SUBSCR_NAME
- OCI_ATTR_SUBSCR_NAMESPACE
- OCI_ATTR_SUBSCR_RECPTPROTO

設定されていない場合は、エラーが戻されます。

また、次の属性

- OCI_ATTR_SUBSCR_CBACK
- OCI_ATTR_SUBSCR_CTX
- OCI_ATTR_SUBSCR_RECPT

のいずれか1つを設定する必要があります。

関連項目： ハンドル属性の詳細は、A-54 ページの「[サブスクリプション・ハンドル属性](#)」を参照してください。

subscrhpp[i] によって示された登録に対して通知を受信すると、*subscrhpp[i]* に対してコンテキスト (OCI_ATTR_SUBSCR_CTX) が設定された状態で、*subscrhpp[i]* に対して設定されたユーザー定義コールバック関数 (OCI_ATTR_SUBSCR_CBACK) が呼び出されるか、*subscrhpp[i]* に対して設定された (OCI_ATTR_SUBSCR_RECPT) に電子メールが送信されます。あるいは、*subscrhpp[i]* のサブスクライバにプロシージャに対する適切な権限がある場合は、*subscrhpp[i]* に対して設定された PL/SQL プロシージャ (OCI_ATTR_SUBSCR_RECPT) がデータベースで呼び出されます。

count (IN)

サブスクリプション・ハンドル配列の要素数です。

errhp (OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

mode (IN)

コール固有モードです。有効な値は次のとおりです。

- OCI_DEFAULT — デフォルトのコールを実行します。登録が切断されているとみなされるように指定します。
- OCI_NOTIFY_CONNECTED — クライアントが接続されている場合にのみ、通知を受信します (このリリースではサポートされていません)。

新しいクライアント・プロセスが発生したとき、または古いクライアント・プロセスのダウンおよび復旧が発生したときは、重要なすべてのサブスクリプションに対する登録が必要です。クライアントが接続されたままサーバーがダウンすると、その後復旧しても、クライアントは、切断された登録の通知を引き続き受信し、接続された登録の通知は受信しません。サーバーのダウンおよび復旧が発生すると、その登録が失われるためです。

コメント

このコールは、サブスクリプションの登録に対して呼び出します。重要なサブスクリプション名および呼び出す関連したコールバックを識別します。重要な複数のサブスクリプションを同時に登録できます。

このインタフェースは、非同期モードのメッセージ配信の場合にのみ有効です。非同期モードでは、サブスクライバが登録コールの発行およびコールバックの指定を行います。サブスクリプション条件と一致するメッセージを受信すると、コールバックが呼び出されます。次に、コールバックから明示的な `message_receive` (デキュー) が発行され、メッセージが取り出されます。

ユーザーは、登録時に名前空間属性を OCI_SUBSCR_NAMESPACE_AQ に設定して、サブスクリプション・ハンドルを指定する必要があります。

サブスクリプション名は、シングル・コンシューマ・キューの登録の場合は文字列 SCHEMA.QUEUE で、マルチ・コンシューマ・キューの登録の場合は文字列 CONSUMER_NAME:SCHEMA.QUEUE です。この文字列は大文字にします。

各名前空間には、独自の権限モデルがあります。登録を行っているユーザーに、指定されたサブスクリプションの名前空間で登録する権限がない場合は、エラーが戻されます。

関連関数

`OCIAQListen()`、`OCISvcCtxToLda()`、`OCISubscriptionEnable()`、
`OCISubscriptionPost()`、`OCISubscriptionUnRegister()`

OCISubscriptionUnRegister()

用途

サブスクリプションの登録を解除して、通知をオフにします。

構文

```
ub4 OCISubscriptionUnRegister ( OCISvcCtx      *svchp,  
                                OCISubscription *subscrhp,  
                                OCIError       *errhp,  
                                ub4            mode );
```

パラメータ

svchp (IN)

OCI サービス・コンテキストです（バージョン7より後）。このサービス・コンテキストには、有効な認証されたユーザー・ハンドルが必要です。

subscrhp (IN)

OCI_ATTR_SUBSCR_NAME および OCI_ATTR_SUBSCR_NAMESPACE 属性が設定されたサブスクリプション・ハンドルです。

関連項目： 詳細は、A-54 ページの「[サブスクリプション・ハンドル属性](#)」を参照してください。

errhp (OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

mode (IN)

コール固有モードです。有効な値は次のとおりです。

- OCI_DEFAULT — デフォルトのコールを実行します。

コメント

サブスクリプションに対する登録を解除すると、その後は特定のサブスクリプションに関する通知を受信できなくなります。通知を再開する場合は、サブスクリプションに対して再登録を行ってください。

別の登録が行われると、重要なクライアントのリストからユーザーが削除されるため、解除前に配信されていた通知はすべて配信されなくなります。

関連関数

[OCIAQListen\(\)](#)、[OCISvcCtxToLda\(\)](#)、[OCISubscriptionEnable\(\)](#)、[OCISubscriptionPost\(\)](#)、[OCISubscriptionRegister\(\)](#)

ダイレクト・パス・ロード関数

この項では、ダイレクト・パス・ロード関数について説明します。

表 16-4 ダイレクト・パス・ロード関数

関数	用途
<code>OCIDirPathAbort()</code> (16-111 ページ)	ダイレクト・パス処理を異常終了します。
<code>OCIDirPathColArrayEntryGet()</code> (16-112 ページ)	列配列内の特定のエントリを取得します。
<code>OCIDirPathColArrayEntrySet()</code> (16-114 ページ)	列配列内の特定のエントリを特定の値に設定します。
<code>OCIDirPathColArrayRowGet()</code> (16-116 ページ)	特定の行番号の基本行ポインタを取得します。
<code>OCIDirPathColArrayReset()</code> (16-117 ページ)	行配列の状態をリセットします。
<code>OCIDirPathColArrayToStream()</code> (16-118 ページ)	列配列からダイレクト・パス・ストリーム形式に変換します。
<code>OCIDirPathDataSave()</code> (16-120 ページ)	データ・セーブポイントを実行するか、またはロードされたデータをコミットしてロード操作を終了します。
<code>OCIDirPathFinish()</code> (16-121 ページ)	ロードされたデータを終了およびコミットします。
<code>OCIDirPathFlushRow()</code> (16-122 ページ)	ダイレクト・パス・ストリーム形式に変換されたデータをロードします。
<code>OCIDirPathLoadStream()</code> (16-123 ページ)	ダイレクト・パス・ストリーム形式に変換されたデータをロードします。
<code>OCIDirPathPrepare()</code> (16-124 ページ)	行の変換またはロードを行うために、ダイレクト・パス・インタフェースを準備します。
<code>OCIDirPathStreamReset()</code> (16-125 ページ)	ダイレクト・パス・ストリームの状態をリセットします。

OCIDirPathAbort()

用途

ダイレクト・パス処理を異常終了します。

構文

```
sword OCIDirPathAbort ( OCIDirPathCtx      *dpctx,  
                        OCIError            *errhp );
```

パラメータ

dpctx (IN)

ダイレクト・パス・コンテキスト・ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

コメント

ダイレクト・パス処理のかわりにサーバーによって維持されていたすべての状態は、異常終了によって破棄されます。ダイレクト・パス・ロードの場合、異常終了前にロードされたデータは問合せで参照できなくなりますが、ロードされていなかったデータが、まだセグメントの領域を使用している可能性があります。索引メンテナンス処理などのロード完了処理は行われません。

関連関数

[OCIDirPathFinish\(\)](#)、[OCIDirPathFlushRow\(\)](#)、[OCIDirPathPrepare\(\)](#)、[OCIDirPathLoadStream\(\)](#)、[OCIDirPathStreamReset\(\)](#)、[OCIDirPathDataSave\(\)](#)

OCIDirPathColArrayEntryGet()

用途

列配列内の指定のエントリを取得します。

構文

```
sword OCIDirPathColArrayEntryGet ( OCIDirPathColArray  *dpca,
                                   OCIError             *errhp,
                                   ub4                   rownum,
                                   ub2                   colIdx,
                                   ub1                   **cvalpp,
                                   ub4                   *clenp,
                                   ub1                   *cflgp );
```

パラメータ

- dpca (IN/OUT)**
ダイレクト・パス列配列ハンドルです。
- errhp (IN)**
エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。
- rownum (IN)**
0 (ゼロ) ベースの行オフセットです。
- colIdx (IN)**
列の識別子 (索引) です。この列 ID は OCIDirPathColAttrSet () によって戻されます。
- cvalpp (IN/OUT)**
列データを指し示すポインタへのポインタです。
- clenp (IN/OUT)**
列データの長さへのポインタです。
- cflgp (IN/OUT)**
列フラグへのポインタです。
次のいずれかの値が戻されます。
- OCI_DIRPATH_COL_COMPLETE — 列のすべてのデータが存在します。
 - OCI_DIRPATH_COL_NULL — 列が NULL です。
 - OCI_DIRPATH_COL_PARTIAL — 一部の列データが提供されます。

コメント

cflgp が OCI_DIRPATH_COL_NULL に設定されている場合、*cvalp* パラメータおよび *clenp* パラメータは、この操作では設定されません。

関連関数

[OCIDirPathColArrayEntrySet\(\)](#)、[OCIDirPathColArrayRowGet\(\)](#)、[OCIDirPathColArrayReset\(\)](#)、[OCIDirPathColArrayToStream\(\)](#)

OCIDirPathColArrayEntrySet()

用途

列配列内の指定のエントリを提供される値に設定します。

構文

```
sword OCIDirPathColArrayEntrySet ( OCIDirPathColArray  *dpca,
                                     OCIError           *errhp,
                                     ub4                  rownum,
                                     ub2                  colIdx,
                                     ub1                  *cvalp,
                                     ub4                  clen,
                                     ub1                  cflg );
```

パラメータ

- dpca (IN/OUT)**
ダイレクト・パス列配列ハンドルです。
- errhp (IN)**
エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。
- rownum (IN)**
0 (ゼロ) ベースの行オフセットです。
- colIdx (IN)**
列の識別子 (索引) です。この列 ID は OCIDirPathColAttrSet () によって戻されます。
- cvalp (IN)**
列データへのポインタです。
- clen (IN)**
列データの長さです。
- cflg (IN)**
列フラグです。次のいずれかの値が戻されます。
- OCI_DIRPATH_COL_COMPLETE — 列のすべてのデータが存在します。
 - OCI_DIRPATH_COL_NULL — 列が NULL です。
 - OCI_DIRPATH_COL_PARTIAL — 一部の列データが提供されます。

コメント

cflg が OCI_DIRPATH_COL_NULL に設定されている場合、*cval* パラメータおよび *clen* パラメータは使用されません。

例

次の例では、列配列内の最初の行のデータのソースに *addr*、長さに *len* を設定します。この例では、列は、*colId* によって識別されます。

```
err = OCIDirPathColArrayEntrySet(dpca, errhp, (ub2)0, colId, addr, len,  
OCI_DIRPATH_COL_COMPLETE);
```

関連関数

[OCIDirPathColArrayRowGet\(\)](#)、[OCIDirPathColArrayRowGet\(\)](#)、
[OCIDirPathColArrayReset\(\)](#)、[OCIDirPathColArrayToStream\(\)](#)

OCIDirPathColArrayRowGet()

用途

特定の行番号の列配列行ポインタを取得します。

構文

```
sword OCIDirPathColArrayRowGet ( OCIDirPathColArray *dpca,  
                                OCIError *errhp,  
                                ub4 rownum,  
                                ub1 ***cvalppp,  
                                ub4 **clenpp,  
                                ub1 **cflgpp );
```

パラメータ

dpca (IN/OUT)

ダイレクト・パス列配列ハンドルです。

errhp (IN)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

rownum (IN)

0 (ゼロ) ベースの行オフセットです。

cvalppp (IN/OUT)

列データのポインタのベクトルへのポインタです。

clenpp (IN/OUT)

列データの長さのベクトルへのポインタです。

cflgpp (IN/OUT)

列フラグのベクトルへのポインタです。

コメント

指定行の列配列エントリにポインタが戻ります。ポインタが戻ると、アプリケーションでは簡単なポインタ計算をして特定行の列間で反復処理を行います。このインタフェースを使用すると、各列で OCIDirPathColArrayEntrySet () をコールする場合と比べて、行の列配列エントリの取得または設定を効率的に行うことができます。アプリケーションから、列配列の境界を超えてメモリーを間接参照しないでください。列配列の次元は、列配列の属性として取得できます。

関連関数

[OCIDirPathColArrayRowGet \(\)](#)、[OCIDirPathColArrayEntrySet \(\)](#)、[OCIDirPathColArrayReset \(\)](#)、[OCIDirPathColArrayToStream \(\)](#)

OCIDirPathColArrayReset()

用途

列配列の状態をリセットします。

構文

```
sword OCIDirPathColArrayReset ( OCIDirPathColArray  *dpca,  
                                OCIError             *errhp );
```

パラメータ

dpca (IN)

ダイレクト・パス列配列ハンドルです。

errhp (IN)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

コメント

列配列の状態は、大きな列内でピース単位操作を行うとき、および列のロード中にエラーが発生したときに、リセットする必要があります。

関連関数

[OCIDirPathColArrayEntryGet\(\)](#)、[OCIDirPathColArrayEntrySet\(\)](#)、
[OCIDirPathColArrayRowGet\(\)](#)、[OCIDirPathColArrayToStream\(\)](#)

OCIDirPathColArrayToStream()

用途

列配列形式からダイレクト・パス・ストリーム形式に変換します。

構文

```
sword OCIDirPathColArrayToStream ( OCIDirPathColArray      *dpca,  
                                   OCIDirPathCtx  const   *dpctx,  
                                   OCIDirPathStream *dpstr,  
                                   OCIError          *errhp,  
                                   ub4                rowcnt,  
                                   ub4                rowoff );
```

パラメータ

dpca (IN)

ダイレクト・パス列配列ハンドルです。

dpctx (IN)

ロードされているオブジェクトのダイレクト・パス・コンテキスト・ハンドルです。

dpstr (IN/OUT)

ダイレクト・パス・ストリーム・ハンドルです。

errhp (IN)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

rowcnt (IN)

列配列内の行数です。

rowoff (IN)

列配列内の開始索引です。

コメント

このインタフェースは、OCIDirPathColAttrSet () によって指定された外部形式の列配列表現データを、ダイレクト・パス・ストリーム形式に変換するときに使用します。変換された形式は、OCIDirPathLoadStream () を使用したロードに適しています。

ダイレクト・パス・ストリーム形式の列データは、Oracle の内部表現のデータに変換されます。すべての変換は、この 2 タスク・インタフェースのクライアント側で行われ、変換エラーは、このインタフェースへのコールと同時に発生します。エラーが発生した行および列に関する情報は、列配列ハンドルの属性として取得できます。

スレッド環境では、同時に行われる OCIDirPathColArrayToStream () 操作によって同じダイレクト・パス・コンテキスト・ハンドルが参照されることがあります。ただし、このインタフェースでは、ダイレクト・パス・コンテキスト・ハンドルは変更されません。

このコールのリターン・コードは次のとおりです。

- **OCI_SUCCESS** — 列配列内のすべてのデータがストリーム形式に正常に変換されました。列配列属性 **OCI_ATTR_ROW_COUNT** は処理された行数です。
- **OCI_ERROR** — 変換中にエラーが発生しました。エラー・ハンドルのエラー情報が格納されます。列配列属性 **OCI_ATTR_ROW_COUNT** は処理された行の数です。属性 **OCI_ATTR_COL_COUNT** にはエラーの原因となった列の列配列の索引が格納されています。
- **OCI_CONTINUE** — 列配列内の一部のデータが、ストリーム形式に変換できませんでした。ストリーム・バッファの領域が不足しているため、すべての列配列データを格納できません。コール元は、データをロードしてファイルに保存するか、または別のストリームを使用して **OCIDirPathArrayToStream()** を再度コールし、残りの列配列データを変換する必要があります。列配列には、変換を再開する場所を示す内部状態が格納されています。列配列属性 **OCI_ATTR_ROW_COUNT** は処理された行数です。
- **OCI_NEED_DATA** — 列配列内のすべてのデータが正常に変換されましたが、列の一部が見つかりました。コール元では、変換後のストリームをロードし、その行の残りの部分を変換する必要があります。必要に応じて操作を反復します。列配列属性 **OCI_ATTR_ROW_COUNT** は処理された行の数です。属性 **OCI_ATTR_COL_COUNT** には、一部がマークされた列の列配列の索引が格納されています。

関連関数

[OCIDirPathColArrayEntryGet\(\)](#)、[OCIDirPathColArrayEntrySet\(\)](#)、
[OCIDirPathColArrayRowGet\(\)](#)、[OCIDirPathColArrayReset\(\)](#)

OCIDirPathDataSave()

用途

要求されたアクションに応じて、データ・セーブポイントを実行するか、またはロードされたデータをコミットしてダイレクト・パス・ロード操作を終了します。

構文

```
sword OCIDirPathDataSave ( OCIDirPathCtx          *dpctx,  
                           OCIError                *errhp,  
                           ub4                      action );
```

パラメータ

dpctx (IN)

ロードされるオブジェクトのダイレクト・パス・コンテキスト・ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

action (IN)

OCIDirPathDataSave () の次のアクション・パラメータの値です。

- OCI_DIRPATH_DATASAVE_SAVEONLY – データ・セーブポイントのみ実行します。
- OCI_DIRPATH_DATASAVE_FINISH – ロードされたデータをコミットしてダイレクト終了関数をコールします。

コメント

戻り値の OCI_SUCCESS は、バックエンドによるデータ・セーブポイントまたは終了ロジックが正常に実行されたことを示します。

LOB では、データ・セーブポイントは実行できません。

終了ロジックの実行は、割り当てられたリソースが解放されないため、ロードの正常終了とは異なります。

関連関数

[OCIDirPathAbort\(\)](#)、[OCIDirPathFinish\(\)](#)、[OCIDirPathFlushRow\(\)](#)、[OCIDirPathPrepare\(\)](#)、[OCIDirPathStreamReset\(\)](#)

OCIDirPathFinish()

用途

ダイレクト・パス・ロード操作を終了します。

構文

```
sword OCIDirPathFinish (   OCIDirPathCtx      *dpctx,  
                           OCIError           *errhp );
```

パラメータ

dpctx (IN)

ロードされるオブジェクトのダイレクト・パス・コンテキスト・ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

コメント

ロードが完了して、ロードされたデータがコミットされるときに、ダイレクト・パス終了関数がコールされます。

戻り値 OCI_SUCCESS は、バックエンドによるロードが正常に終了したことを示します。

関連関数

[OCIDirPathAbort\(\)](#)、[OCIDirPathDataSave\(\)](#)、[OCIDirPathFlushRow\(\)](#)、[OCIDirPathPrepare\(\)](#)、[OCIDirPathStreamReset\(\)](#)

OCIDirPathFlushRow()

用途

サーバーから部分的にロードされた行をフラッシュします。

構文

```
sword OCIDirPathFlushRow (   OCIDirPathCtx          *dpctx,  
                             OCIError                *errhp );
```

パラメータ

dpctx (IN)

ロードされるオブジェクトのダイレクト・パス・コンテキスト・ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

コメント

この関数は、行の一部がロードされたときに必要ですが、アプリケーションで次に処理するピースで変換エラーが発生します。現在、部分状態の行のみが廃棄されます。サーバーが、現在、ダイレクト・パス・コンテキストに関連するオブジェクトに対する部分状態の行を処理していない場合、この関数は基本的に何も行いません。

関連関数

[OCIDirPathAbort\(\)](#)、[OCIDirPathFinish\(\)](#)、[OCIDirPathPrepare\(\)](#)、[OCIDirPathLoadStream\(\)](#)

OCIDirPathLoadStream()

用途

ダイレクト・パス・ストリーム形式に変換されたデータをロードします。

構文

```
sword OCIDirPathLoadStream (   OCIDirPathCtx          *dpctx,
                               OCIDirPathStream        *dpstr,
                               OCIError                 *errhp );
```

パラメータ

dpctx (IN)

ロードされるオブジェクトのダイレクト・パス・コンテキスト・ハンドルです。

dpstr (IN)

ロードするストリームのダイレクト・パス・ストリーム・ハンドルです。

errhp (IN)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

コメント

インタフェースからエラーが戻されたときは、ストリームのソースとなった列配列内の列に関する情報を、ダイレクト・パス・ストリームの属性として取得できます。また、エラーが発生したストリーム内のオフセットも、ストリームの属性として取得できます。

この関数のリターン・コードは次のとおりです。

- OCI_SUCCESS – ストリーム内のすべてのデータが正常にロードされました。
- OCI_ERROR – データのロード中にエラーが発生しました。問題は、パーティションのマッピング・エラー、NULL 制約の違反、ファンクション索引評価エラー、またはエクステントを割り当てられないなどの領域条件の不足である可能性があります。ダイレクト・パス・ストリームの属性 OCI_ATTR_STREAM_OFFSET は、違反している行に対応するストリームのオフセットです。OCI_ATTR_ROW_COUNT は処理された行数です。
- OCI_NEED_DATA – 最後の行が完全な行ではありませんでした。コール元は、行の残りをロードする必要があります。ストリームのソースが列配列である場合は、属性 OCI_ATTR_ROW_COUNT は処理された完全な行の数になります。
- OCI_NO_DATA – 空のストリームまたは完全に処理されているストリームをロードしようとしてしました。

関連関数

[OCIDirPathAbort\(\)](#)、[OCIDirPathDataSave\(\)](#)、[OCIDirPathFinish\(\)](#)、[OCIDirPathPrepare\(\)](#)、[OCIDirPathStreamReset\(\)](#)

OCIDirPathPrepare()

用途

行の変換またはロードを行う前に、ダイレクト・パス・ロード・インタフェースを準備します。

構文

```
sword OCIDirPathPrepare (   OCIDirPathCtx          *dpctx,  
                             OCISvcCtx                *svchp,  
                             OCIError                  *errhp );
```

パラメータ

dpctx (IN)

ロードされるオブジェクトのダイレクト・パス・コンテキスト・ハンドルです。

svchp (IN)

サービス・コンテキストです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

コメント

操作するオブジェクトの名前、列データの外部属性、およびすべてのロード・オプションの設定後、行の変換またはロードの前に OCIDirPathPrepare () を使用してダイレクト・パス・インタフェースを準備する必要があります。

戻り値 OCI_SUCCESS は、ダイレクト・パス・ロード操作を行うために、バックエンドが適切に初期化されたことを示しています。0（ゼロ）以外の戻り値は、エラーを示します。戻される可能性のあるエラーは次のとおりです。

- コンテキストが無効です。
- サーバーに接続されていません。
- オブジェクト名が設定されていません。
- すでに準備されています（2度準備することはできません）。
- オブジェクトがダイレクト・パス処理に適切ではありません。

関連関数

[OCIDirPathAbort\(\)](#)、[OCIDirPathDataSave\(\)](#)、[OCIDirPathFinish\(\)](#)、[OCIDirPathFlushRow\(\)](#)、[OCIDirPathStreamReset\(\)](#)

OCIDirPathStreamReset()

用途

ダイレクト・パス・ストリームの状態をリセットします。

構文

```
sword OCIDirPathStreamReset ( OCIDirPathStream      *dpstr,  
                               OCIError              *errhp );
```

パラメータ

dpstr (IN)

ダイレクト・パス・ストリーム・ハンドルです。

errhp (IN)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

コメント

ダイレクト・パス・ストリームでは、次の OCIDirPathColArrayToStream() がストリームに書き込みを開始する場所を示す状態を維持します。通常、データはストリームの末尾に追加されます。ストリームが正常にロードされた後にコール元から新しいストリームを開始する場合、またはストリーム内のデータを廃棄する場合は、このコールを使用してストリームをリセットする必要があります。

関連関数

[OCIDirPathAbort\(\)](#)、[OCIDirPathDataSave\(\)](#)、[OCIDirPathFinish\(\)](#)、[OCIDirPathFlushRow\(\)](#)、[OCIDirPathPrepare\(\)](#)

スレッド管理関数

この項では、スレッド管理関数について説明します。

表 16-5 スレッド管理関数

関数	用途
OCIThreadClose() (16-128 ページ)	スレッド・ハンドルをクローズします。
OCIThreadCreate() (16-129 ページ)	新しいスレッドを作成します。
OCIThreadHandleGet() (16-131 ページ)	コールが行われたスレッドの OCIThreadHandle を取り出します。
OCIThreadHndDestroy() (16-132 ページ)	スレッド・ハンドルの破棄および割当て解除を行います。
OCIThreadHndInit() (16-133 ページ)	スレッド・ハンドルの割当ておよび初期化を行います。
OCIThreadIdDestroy() (16-134 ページ)	スレッド ID の破棄および割当て解除を行います。
OCIThreadIdGet() (16-135 ページ)	コールが行われたスレッドの OCIThreadId を取り出します。
OCIThreadIdInit() (16-136 ページ)	スレッド ID の割当ておよび初期化を行います。
OCIThreadIdNull() (16-137 ページ)	特定の OCIThreadId が NULL スレッド ID かどうかを判断します。
OCIThreadIdSame() (16-138 ページ)	2 つの OCIThreadId が同じスレッドを表しているかどうかを判断します。
OCIThreadIdSet() (16-139 ページ)	OCIThreadId を別の OCIThreadId に設定します。
OCIThreadIdSetNull() (16-140 ページ)	特定の OCIThreadId に NULL スレッド ID を設定します。
OCIThreadInit() (16-141 ページ)	OCIThread コンテキストを初期化します。
OCIThreadIsMulti() (16-142 ページ)	アプリケーションがマルチスレッド環境とシングル・スレッド環境のどちらで動作しているかを、コール元に通知します。
OCIThreadJoin() (16-143 ページ)	コール側スレッドから別のスレッドに結合できるようにします。
OCIThreadKeyDestroy() (16-144 ページ)	key が指し示しているキーの破棄および割当て解除を行います。
OCIThreadKeyGet() (16-145 ページ)	キーに対してコール側スレッドのカレント値を取得します。
OCIThreadKeyInit() (16-146 ページ)	キーを作成します。
OCIThreadKeySet() (16-147 ページ)	キーに対してコール側スレッドの値を設定します。
OCIThreadMutexAcquire() (16-148 ページ)	コールが行われたスレッドに対して mutex を取得します。
OCIThreadMutexDestroy() (16-149 ページ)	mutex の破棄および割当て解除を行います。
OCIThreadMutexInit() (16-150 ページ)	mutex の割当ておよび初期化を行います。

表 16-5 スレッド管理関数（続き）

関数	用途
OCIThreadMutexRelease() （16-151 ページ）	mutex を解放します。
OCIThreadProcessInit() （16-152 ページ）	OCIThread プロセスの初期化を行います。
OCIThreadTerm() （16-153 ページ）	OCIThread コンテキストを解放します。

OCIThreadClose()

用途

スレッド・ハンドルをクローズします。

構文

```
sword OCIThreadClose ( dvoid          *hndl,
                       OCIError       *err,
                       OCIThreadHandle *tHnd );
```

パラメータ

hndl (IN/OUT)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet() のコールによって診断情報を取得できます。

tHnd (IN/OUT)

クローズする OCIThread スレッド・ハンドルです。

コメント

`tHnd` は、OCIThreadHndInit() を使用して初期化する必要があります。同一 OCIThreadCreate() コールから戻されたスレッド・ハンドルおよびスレッド ID は、OCIThreadClose() コール後はどちらも無効になります。

関連関数

[OCIThreadCreate\(\)](#)

OCIThreadCreate()

用途

新しいスレッドを作成します。

構文

```
sword OCIThreadCreate ( dvoid          *hndl,
                        OCIError       *err,
                        void (*start)   (dvoid
                        dvoid          *arg,
                        OCIThreadId     *tid,
                        OCIThreadHandle *tHnd );
```

パラメータ

hndl (IN/OUT)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet () のコールによって診断情報を取得できます。

start (IN)

新しいスレッドが実行を開始する関数です。

arg (IN)

`start` が指し示している関数に渡す引数です。

tid (IN/OUT)

NULL でない場合は、新しいスレッドに ID を取得します。

tHnd (IN/OUT)

NULL でない場合は、新しいスレッドにハンドルを取得します。

コメント

arg から渡された引数を使用して、*start* が指し示している関数コールを実行すると、新しいスレッドが開始します。その関数が復帰すると、新しいスレッドは終了します。関数は値を戻さず、パラメータ **dvoid** を受け入れます。*tHnd* が **NULL** 以外の場合にかぎり、`OCIThreadCreate()` コールは `OCIThreadClose()` コールに一致する必要があります。

tHnd が **NULL** の場合は、生成されるスレッドの終了のタイミングが不明のため、**tid* 内に配置されたスレッド ID はコール側スレッドでは無効になります。

tid は `OCIThreadIdInit()` によって初期化します。また、*tHnd* は `OCIThreadHndInit()` によって初期化します。

関連関数

[OCIThreadClose\(\)](#)、[OCIThreadIdInit\(\)](#)、[OCIThreadHndInit\(\)](#)

OCIThreadHandleGet()

用途

コールが行われたスレッドの **OCIThreadHandle** を取り出します。

構文

```
sword OCIThreadHandleGet ( dvoid          *hndl,
                           OCIError       *err,
                           OCIThreadHandle *tHnd );
```

パラメータ

hndl (IN/OUT)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、**OCI_ERROR** が戻されます。OCIErrorGet () のコールによって診断情報を取得できます。

tHnd (IN/OUT)

NULL 以外の場合は、スレッドのスレッド・ハンドルを配置する場所です。

コメント

tHnd は、OCIThreadHndInit () を使用して初期化する必要があります。

この関数によって取り出されるスレッド・ハンドル *tHnd* は、使用後に OCIThreadClose () を使用してクローズし、OCIThreadHndDestroy () を使用して破棄する必要があります。

関連関数

[OCIThreadHndDestroy\(\)](#)、[OCIThreadHndInit\(\)](#)

OCIThreadHndDestroy()

用途

スレッド・ハンドルの破棄および割当て解除を行います。

構文

```
sword OCIThreadHndDestroy ( dvoid          *hndl,
                             OCIError       *err,
                             OCIThreadHandle **thnd );
```

パラメータ

hndl (IN/OUT)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet () のコールによって診断情報を取得できます。

thnd (IN/OUT)

破棄するスレッド・ハンドルへのポインタのアドレスです。

コメント

`thnd` は、OCIThreadHndInit () を使用して初期化する必要があります。

関連関数

[OCIThreadHandleGet \(\)](#)、[OCIThreadHndInit \(\)](#)

OCIThreadHndInit()

用途

スレッド・ハンドルの割当ておよび初期化を行います。

構文

```
sword OCIThreadHndInit ( dvoid          *hndl,  
                        OCIError        *err,  
                        OCIThreadHandle **thnd );
```

パラメータ

hndl (IN/OUT)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、OCI_ERROR が戻されます。OCIErrorGet () のコールによって診断情報を取得できます。

thnd (OUT)

初期化するスレッド・ハンドルへのポインタのアドレスです。

関連関数

[OCIThreadHandleGet \(\)](#)、[OCIThreadHndDestroy \(\)](#)

OCIThreadIdDestroy()

用途

スレッド ID の破棄および割当て解除を行います。

構文

```
sword OCIThreadIdDestroy (dvoid          *hndl,
                          OCIError       *err,
                          OCIThreadId    **tid );
```

パラメータ

hndl (IN/OUT)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーが発生して `OCI_ERROR` が戻された場合、そのエラーは `err` に記録され、`OCIErrorGet()` のコールによって診断情報を取得できます。

tid (IN/OUT)

破棄するスレッド ID へのポインタです。

コメント

`tid` は、`OCIThreadIdInit()` を使用して初期化する必要があります。

関連関数

[OCIThreadIdGet\(\)](#)、[OCIThreadIdInit\(\)](#)、[OCIThreadIdNull\(\)](#)、[OCIThreadIdSame\(\)](#)、[OCIThreadIdSet\(\)](#)、[OCIThreadIdSetNull\(\)](#)

OCIThreadIdGet()

用途

コールされるスレッドの **OCIThreadId** を取り出します。

構文

```
sword OCIThreadIdGet ( dvoid          *hndl,  
                      OCIError       *err,  
                      OCIThreadId    *tid );
```

パラメータ

hndl (IN/OUT)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、**OCI_ERROR** が戻されます。OCIErrorGet () のコールによって診断情報を取得できます。

tid (OUT)

コール側スレッドの ID を配置する場所を指し示す必要があります。

コメント

tid は、OCIThreadIdInit () を使用して初期化する必要があります。OCIThread がシングル・スレッド環境で使用された場合は、**tid** が指し示している場所に、OCIThreadIdGet () によって常に同じ値が配置されます。正確な値であることも重要ですが、この値が NULL スレッド ID とは異なり、常に同じ値であるということが重要です。

関連関数

[OCIThreadIdDestroy\(\)](#)、[OCIThreadIdInit\(\)](#)、[OCIThreadIdNull\(\)](#)、[OCIThreadIdSame\(\)](#)、[OCIThreadIdSet\(\)](#)、[OCIThreadIdSetNull\(\)](#)

OCIThreadIdInit()

用途

スレッド ID *tid* の割当ておよび初期化を行います。

構文

```
sword OCIThreadIdInit ( dvoid          *hndl,  
                        OCIError       *err,  
                        OCIThreadId    **tid );
```

パラメータ

hndl (IN/OUT)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーが発生して `OCI_ERROR` が戻された場合、そのエラーは *err* に記録され、`OCIErrorGet()` のコールによって診断情報を取得できます。

tid (OUT)

初期化するスレッド ID へのポインタです。

関連関数

[OCIThreadIdDestroy\(\)](#)、[OCIThreadIdGet\(\)](#)、[OCIThreadIdNull\(\)](#)、
[OCIThreadIdSame\(\)](#)、[OCIThreadIdSet\(\)](#)、[OCIThreadIdSetNull\(\)](#)

OCIThreadIdNull()

用途

指定された **OCIThreadId** が NULL スレッド ID かどうかを判断します。

構文

```
sword OCIThreadIdNull ( dvoid          *hndl,  
                        OCIError       *err,  
                        OCIThreadId    *tid,  
                        boolean        *result );
```

パラメータ

hndl (IN/OUT)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、**OCI_ERROR** が戻されます。OCIErrorGet() のコールによって診断情報を取得できます。

tid (IN)

チェックする **OCIThreadId** へのポインタです。

result (IN/OUT)

結果へのポインタです。

コメント

tid が NULL スレッド ID の場合、*result* は TRUE に設定されます。それ以外の場合、*result* は FALSE に設定されます。*tid* は OCIThreadIdInit() で初期化する必要があります。

関連関数

[OCIThreadIdDestroy\(\)](#)、[OCIThreadIdGet\(\)](#)、[OCIThreadIdInit\(\)](#)、[OCIThreadIdSame\(\)](#)、[OCIThreadIdSet\(\)](#)、[OCIThreadIdSetNull\(\)](#)

OCIThreadIdSame()

用途

2 つの **OCIThreadId** が同じスレッドを表しているかどうかを判断します。

構文

```
sword OCIThreadIdSame ( dvoid          *hdl,  
                        OCIError       *err,  
                        OCIThreadId    *tid1,  
                        OCIThreadId    *tid2,  
                        boolean        *result );
```

パラメータ

hdl (IN/OUT)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、OCI_ERROR が戻されます。OCIErrorGet () のコールによって診断情報を取得できます。

tid1 (IN)

最初の **OCIThreadId** へのポインタです。

tid2 (IN)

2 番目の **OCIThreadId** へのポインタです。

result (IN/OUT)

結果へのポインタです。

コメント

tid1 および *tid2* が同じスレッドを表す場合、*result* は TRUE に設定されます。それ以外の場合、*result* は FALSE に設定されます。*tid1* および *tid2* の両方が NULL スレッド ID である場合、*result* は TRUE に設定されます。*tid1* および *tid2* は、OCIThreadIdInit () で初期化する必要があります。

関連関数

[OCIThreadIdDestroy\(\)](#)、[OCIThreadIdGet\(\)](#)、[OCIThreadIdInit\(\)](#)、[OCIThreadIdNull\(\)](#)、[OCIThreadIdSet\(\)](#)、[OCIThreadIdSetNull\(\)](#)

OCIThreadIdSet()

用途

OCIThreadId を別の **OCIThreadId** に設定します。

構文

```
sword OCIThreadIdSet ( dvoid          *hndl,  
                      OCIError       *err,  
                      OCIThreadId    *tidDest,  
                      OCIThreadId    *tidSrc );
```

パラメータ

hndl (IN/OUT)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーが発生して **OCI_ERROR** が戻された場合、そのエラーは *err* に記録され、**OCIErrorGet()** のコールによって診断情報を取得できます。

tidDest (OUT)

設定後の **OCIThreadId** の場所を指し示す必要があります。

tidSrc (IN)

設定前の **OCIThreadId** を指し示す必要があります。

コメント

tid は、**OCIThreadIdInit()** を使用して初期化する必要があります。

関連関数

[OCIThreadIdDestroy\(\)](#)、[OCIThreadIdGet\(\)](#)、[OCIThreadIdInit\(\)](#)、[OCIThreadIdNull\(\)](#)、[OCIThreadIdSame\(\)](#)、[OCIThreadIdSetNull\(\)](#)

OCIThreadIdSetNull()

用途

指定された **OCIThreadId** に NULL スレッド ID を設定します。

構文

```
sword OCIThreadIdSetNull ( dvoid          *hndl,  
                           OCIError      *err,  
                           OCIThreadId   *tid );
```

パラメータ

hndl (IN/OUT)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、**OCI_ERROR** が戻されます。OCIErrorGet() のコールによって診断情報を取得できます。

tid (OUT)

NULL スレッド ID を設定する **OCIThreadId** を指し示す必要があります。

コメント

tid は、OCIThreadIdInit() を使用して初期化する必要があります。

関連関数

[OCIThreadIdDestroy\(\)](#)、[OCIThreadIdGet\(\)](#)、[OCIThreadIdInit\(\)](#)、[OCIThreadIdNull\(\)](#)、[OCIThreadIdSame\(\)](#)、[OCIThreadIdSet\(\)](#)

OCIThreadInit()

用途

OCIThread コンテキストを初期化します。

構文

```
sword OCIThreadInit ( dvoid      *hndl,  
                     OCIError   *err );
```

パラメータ

hndl (IN/OUT)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーが発生して OCI_ERROR が戻された場合、そのエラーは err に記録され、OCIErrorGet() のコールによって診断情報を取得できます。

コメント

戻されたポインタが指し示しているメモリーを、OCIThread クライアントから検査しないでください。同時に OCIThreadInit() コールを実行してください。

OCIThreadProcessInit() と異なり、他のコールより先に初期コールを行う必要はありません。

OCIThreadInit() の 1 回目のコールでは、OCI スレッド・コンテキストが初期化されます。また、そのコンテキストへのポインタを、システム情報も含めて保存します。2 回目以降の OCIThreadInit() コールでは、同じコンテキストが戻されます。

各 OCIThreadInit() コールは、結果的に OCIThreadTerm() コールに合致する必要があります。

関連関数

[OCIThreadTerm\(\)](#)

OCIThreadIsMulti()

用途

アプリケーションがマルチスレッド環境とシングル・スレッド環境のどちらで動作しているかを、コール元に通知します。

構文

```
boolean OCIThreadIsMulti ( );
```

戻り値

TRUE — 環境がマルチスレッドの場合

FALSE — 環境がシングル・スレッドの場合

関連関数

[OCIThreadIdDestroy\(\)](#)、[OCIThreadIdGet\(\)](#)、[OCIThreadIdInit\(\)](#)、[OCIThreadIdNull\(\)](#)、[OCIThreadIdSame\(\)](#)、[OCIThreadIdSet\(\)](#)

OCIThreadJoin()

用途

コール側スレッドが別のスレッドと結合できるようにします。

構文

```
sword OCIThreadJoin ( dvoid          *hdl,  
                     OCIError       *err,  
                     OCIThreadHandle *tHnd );
```

パラメータ

hdl (IN/OUT)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、**OCI_ERROR** が戻されます。OCIErrorGet () のコールによって診断情報を取得できます。

tHnd (IN)

結合するスレッドの **OCIThreadHandle** です。

コメント

この関数は、指定されたスレッドが終了するまでコール元をブロックします。

tHnd は、OCIThreadHndInit () を使用して初期化する必要があります。複数のスレッドをすべて同一のスレッドに結合した場合、結果は保証されていません。

関連関数

[OCIThreadIdDestroy\(\)](#)、[OCIThreadIdGet\(\)](#)、[OCIThreadIdInit\(\)](#)、[OCIThreadIdNull\(\)](#)、[OCIThreadIdSame\(\)](#)、[OCIThreadIdSet\(\)](#)

OCIThreadKeyDestroy()

用途

key が指し示しているキーの破棄および割当て解除を行います。

構文

```
sword OCIThreadKeyDestroy ( dvoid          *hndl,
                             OCIError      *err,
                             OCIThreadKey  **key );
```

パラメータ

hndl (IN/OUT)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーが発生して `OCI_ERROR` が戻された場合、そのエラーは *err* に記録され、`OCIErrorGet()` のコールによって診断情報を取得できます。

key (IN/OUT)

キーを破棄する `OCIThreadKey` です。

コメント

キー作成ルーチンに渡されるデストラクタ関数のコールバックとは異なります。この新しい破棄関数 `OCIThreadKeyDestroy()` は、*key* を作成したときに、`OCI_THREAD` によって取得されたリソースの終了に使用します。`OCIThreadKeyInit()` の `OCIThreadKeyDestFunc` コールバックは、キー値のデストラクタです。キー自体の操作は行いません。

この関数は、キーの使用が終了したときにコールする必要があります。キー破棄関数をコールしないと、メモリー・リークが発生する可能性があります。

関連関数

[OCIThreadKeyGet\(\)](#)、[OCIThreadKeyInit\(\)](#)、[OCIThreadKeySet\(\)](#)

OCIThreadKeyGet()

用途

キーに対してコール側スレッドのカレント値を取得します。

構文

```
sword OCIThreadKeyGet ( dvoid          *hndl,
                        OCIError       *err,
                        OCIThreadKey   *key,
                        dvoid          **pValue );
```

パラメータ

hndl (IN/OUT)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーが発生して `OCI_ERROR` が戻された場合、そのエラーは `err` に記録され、`OCIErrorGet()` のコールによって診断情報を取得できます。

key (IN)

キーです。

pValue (IN/OUT)

スレッド固有のキー値を配置する場所です。

コメント

`OCIThreadKeyInit()` を使用して作成されなかったキーに対して、この関数は使用しないでください。

コール側スレッドからキーに値が割り当てられていない場合は、`pValue` が指し示す場所に `NULL` が配置されます。

関連関数

[OCIThreadKeyDestroy\(\)](#)、[OCIThreadKeyInit\(\)](#)、[OCIThreadKeySet\(\)](#)

OCIThreadKeyInit()

用途

キーを作成します。

構文

```
sword OCIThreadKeyInit (dvoid                *hdl,  
                        OCIError             *err,  
                        OCIThreadKey         **key,  
                        OCIThreadKeyDestFunc destFn );
```

パラメータ

hdl (IN/OUT)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーが発生して `OCI_ERROR` が戻された場合、そのエラーは `err` に記録され、`OCIErrorGet()` のコールによって診断情報を取得できます。

key (OUT)

新しいキーの作成を行う `OCIThreadKey` です。

destFn (IN)

キーのデストラクタです。NULL でもかまいません。

コメント

このルーチンをコールするたびに、他のキーと異なる新しいキーの割当ておよび生成が行われます。この関数が正常に実行されると、割当ておよび初期化が行われたキーへのポインタが戻されます。このキーは、`OCIThreadKeyGet()` および `OCIThreadKeySet()` とともに使用します。すべてのスレッドで、このキーの初期値は NULL です。

この関数を、`key` パラメータに同じ値を指定して、複数回コールしないでください。

`destFn` パラメータが NULL 以外の場合は、キーの値が NULL 以外のスレッドが終了するたびに、`destFn` によって指し示されているルーチンがコールされます。そのルーチンがコールされるときのパラメータは、1 つです。そのパラメータは、スレッド終了時のそのスレッドに対するキーの値です。キーにデストラクタ関数が不要な場合は、`destFn` に NULL を渡します。

関連関数

[OCIThreadKeyDestroy\(\)](#)、[OCIThreadKeyGet\(\)](#)、[OCIThreadKeySet\(\)](#)

OCIThreadKeySet()

用途

キーに対してコール側スレッドの値を設定します。

構文

```
sword OCIThreadKeySet ( dvoid          *hndl,
                        OCIError       *err,
                        OCIThreadKey   *key,
                        dvoid          *value );
```

パラメータ

hndl (IN/OUT)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーが発生して `OCI_ERROR` が戻された場合、そのエラーは `err` に記録され、`OCIErrorGet()` のコールによって診断情報を取得できます。

key (IN/OUT)

キーです。

value (IN)

キーに設定するスレッド固有の値です。

コメント

`OCIThreadKeyInit()` を使用して作成されなかったキーに対して、この関数は使用しないでください。

関連関数

[OCIThreadKeyDestroy\(\)](#)、[OCIThreadKeyGet\(\)](#)、[OCIThreadKeyInit\(\)](#)

OCIThreadMutexAcquire()

用途

コールが行われたスレッドに対して **mutex** を取得します。

構文

```
sword OCIThreadMutexAcquire ( dvoid          *hndl,  
                              OCIError      *err,  
                              OCIThreadMutex *mutex );
```

パラメータ

hndl (IN/OUT)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、**OCI_ERROR** が戻されます。OCIErrorGet() のコールによって診断情報を取得できます。

mutex (IN/OUT)

取得する **mutex** です。

コメント

mutex が別のスレッドによって保持されている場合は、**mutex** が取得できるまでコール側スレッドはブロックされます。

初期化されていない **mutex** は取得しないでください。

スレッドによってすでに保持されている **mutex** を取得するときに、この関数が使用されるかどうかは定義されていません。

関連関数

[OCIThreadMutexDestroy\(\)](#)、[OCIThreadMutexInit\(\)](#)、[OCIThreadMutexRelease\(\)](#)

OCIThreadMutexDestroy()

用途

`mutex` の破棄および割当て解除を行います。

構文

```
sword OCIThreadMutexDestroy ( dvoid          *hndl,  
                              OCIError       *err,  
                              OCIThreadMutex **mutex );
```

パラメータ

hndl (IN/OUT)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーが発生して `OCI_ERROR` が戻された場合、そのエラーは `err` に記録され、`OCIErrorGet()` のコールによって診断情報を取得できます。

mutex (IN/OUT)

破棄する `mutex` です。

コメント

不要になった各 `mutex` は、破棄する必要があります。

初期化されていない `mutex`、またはスレッドに現在保持されている `mutex` を破棄しないでください。`mutex` の破棄は、その `mutex` に対する他の操作と同時に行わないでください。また、`mutex` の破棄後は、その `mutex` を使用しないでください。

関連関数

[OCIThreadMutexAcquire\(\)](#)、[OCIThreadMutexInit\(\)](#)、
[OCIThreadMutexRelease\(\)](#)

OCIThreadMutexInit()

用途

`mutex` の割当ておよび初期化を行います。

構文

```
sword OCIThreadMutexInit ( dvoid          *hndl,
                           OCIError       *err,
                           OCIThreadMutex **mutex );
```

パラメータ

hndl (IN/OUT)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーが発生して `OCI_ERROR` が戻された場合、そのエラーは `err` に記録され、`OCIErrorGet()` のコールによって診断情報を取得できます。

mutex (OUT)

初期化する `mutex` です。

コメント

すべての `mutex` は、使用する前に初期化する必要があります。

複数のスレッドから同時に 1 つの `mutex` を初期化しないでください。また、`mutex` は、破壊されるまで初期化しなおさないでください (`OCIThreadMutexDestroy()` を参照)。

関連関数

[OCIThreadMutexDestroy\(\)](#)、[OCIThreadMutexAcquire\(\)](#)、[OCIThreadMutexRelease\(\)](#)

OCIThreadMutexRelease()

用途

mutex を解放します。

構文

```
sword OCIThreadMutexRelease ( dvoid          *hndl,  
                             OCIError       *err,  
                             OCIThreadMutex *mutex );
```

パラメータ

hndl (IN/OUT)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーが発生して OCI_ERROR が戻された場合、そのエラーは err に記録され、OCIErrorGet() のコールによって診断情報を取得できます。

mutex (IN/OUT)

解放する mutex です。

コメント

mutex にブロックされているスレッドが存在する場合は、スレッドの 1 つが mutex を取得し、スレッドのブロックが解除されます。

初期化されていない mutex を解放しないでください。また、スレッドから、そのスレッドが保持していない mutex を解放しないでください。

関連関数

[OCIThreadMutexDestroy\(\)](#)、[OCIThreadMutexInit\(\)](#)、[OCIThreadMutexAcquire\(\)](#)

OCIThreadProcessInit()

用途

OCIThread プロセスの初期化を実行します。

構文

```
void OCIThreadProcessInit ( );
```

コメント

この関数をコールする必要があるかどうかは、OCI Thread の使用方法によって決まります。

シングル・スレッド・アプリケーションでは、この関数はコールする必要はありません。コールする場合、1 回目のコールは、他の OCIThread 関数に対するコールよりも先に行う必要があります。2 回目以降のコールには制限はありません。コールしても何も処理されません。

マルチスレッド・アプリケーションでは、この関数は必ずコールする必要があります。1 回目のコールは、他の OCIThread コールよりも先に行う必要があります。つまり、1 回目のコールと同時に、他の OCIThread 関数のコール（このコールも含みます）を行うことはできません。

2 回目以降のコールには制限はありません。コールしても何も処理されません。

関連関数

[OCIThreadIdDestroy\(\)](#)、[OCIThreadIdGet\(\)](#)、[OCIThreadIdInit\(\)](#)、[OCIThreadIdNull\(\)](#)、[OCIThreadIdSame\(\)](#)、[OCIThreadIdSet\(\)](#)

OCIThreadTerm()

用途

OCIThread コンテキストを解放します。

構文

```
sword OCIThreadTerm ( dvoid      *hdl,  
                      OCIError   *err );
```

パラメータ

hdl (IN/OUT)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーが発生して OCI_ERROR が戻された場合、そのエラーは err に記録され、OCIErrorGet () のコールによって診断情報を取得できます。

コメント

この関数は、OCIThreadInit () への各コールに対して 1 回のみコールする必要があります。

同時に OCIThreadTerm () コールを行ってください。OCIThreadTerm () のコール回数が OCIThreadInit () のコール回数と異なる場合、処理は行われません。コール回数が同じ場合は、OCIThread 層を終了し、コンテキストに割り当てられていたメモリーを解放します。この操作の後に、そのコンテキストを再使用しないでください。OCIThreadInit () をコールして新しいコンテキストを取得する必要があります。

関連関数

[OCIThreadInit\(\)](#)

トランザクション関数

この項では、トランザクション関数について説明します。

表 16-6 トランザクション関数

関数	用途
OCITransCommit() (16-155 ページ)	サービス・コンテキスト上のトランザクションをコミットします。
OCITransDetach() (16-158 ページ)	サービス・コンテキストからトランザクションを連結解除します。
OCITransForget() (16-159 ページ)	準備したグローバル・トランザクションを放棄します。
OCITransMultiPrepare() (16-160 ページ)	単一セルに複数のブランチがあるトランザクションを準備します。
OCITransPrepare() (16-161 ページ)	グローバル・トランザクションをコミットのために準備します。
OCITransRollback() (16-162 ページ)	トランザクションをロールバックします。
OCITransStart() (16-163 ページ)	サービス・コンテキスト上のトランザクションを開始します。

OCITransCommit()

用途

指定のサービス・コンテキストに対応付けられたトランザクションをコミットします。

構文

```
sword OCITransCommit ( OCISvcCtx      *svchp,  
                        OCIError      *errhp,  
                        ub4            flags );
```

パラメータ

svchp (IN)

サービス・コンテキスト・ハンドルです。

errhp (IN)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

flags (IN)

グローバル・トランザクションの 1 フェーズ・コミットを最適化するためのフラグです。

トランザクションが分散型でない場合、**flags** パラメータは無視され、その値として OCI_DEFAULT を渡すことができます。2 フェーズ・コミットでは、グローバル・トランザクションを管理している OCI アプリケーションは、**flags** に対して OCI_TRANS_TWOPHASE の値を渡す必要があります。デフォルトは 1 フェーズ・コミットです。

コメント

サービス・コンテキストに現在対応付けられているトランザクションをコミットします。トランザクションがサーバーによるコミットが不可能なグローバル・トランザクションである場合、このコールは、トランザクションの状態をデータベースから取り出し、エラー・ハンドルを使用してユーザーに戻します。

複数のトランザクションを定義している場合、この関数は、サービス・コンテキストに現在対応付けられているトランザクションを処理します。データベースの変更時に作成される暗黙的ローカル・トランザクションのみを処理している場合は、その暗黙的トランザクションがコミットされます。

アプリケーションをオブジェクト・モードで実行している場合は、このトランザクションに対してオブジェクト・キャッシュで変更または更新されたオブジェクトもフラッシュされ、コミットされます。

正常な状況では、OCITransCommit () は、トランザクションがコミットされたかロールバックされたかを示す状態を戻します。グローバル・トランザクションでは、トランザクションがインダウトの状態、つまり、コミットも異常終了もされていない状態の場合もあり

得ます。この場合、OCITransCommit() は、トランザクションの状態をサーバーから取り出します。そのステータスが戻ります。

例

次の例は、8-3 ページの「[単純なローカル・トランザクション](#)」に記述されている単純なローカル・トランザクションの使用方法を説明しています。

```
int main()
{
    OCIEnv *envhp;
    OCIError *errhp;
    OCISvcCtx *svchp;
    OCIStmt *stmthp;
    dvoid *tmp;
    text sqlstmt[128];

    OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0, (dvoid * (*)()) 0,
                  (dvoid * (*)()) 0, (void (*)()) 0 );

    OCIHandleAlloc( (dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
                    0, (dvoid **) &tmp);

    OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, (size_t)0, (dvoid **) &tmp );

    OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
                    (size_t)0, (dvoid **) &tmp);
    OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SERVER,
                    (size_t)0, (dvoid **) &tmp);

    OCIErrorAttach( svchp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

    OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &stmthp, (ub4) OCI_HTYPE_STMT,
                    (size_t)0, (dvoid **) &tmp);

    OCIHandleAlloc((dvoid *)envhp, (dvoid **)&stmthp, OCI_HTYPE_STMT, 0, 0);

    OCIAttrSet((dvoid *)svchp, OCI_HTYPE_SVCCTX, (dvoid *)errhp, 0,
                OCI_ATTR_SERVER, errhp);

    OCILogon(envhp, errhp, &svchp, "SCOTT", strlen("SCOTT"),
              "TIGER", strlen("TIGER"), 0, 0);
```

```
/* update scott.emp empno=7902, increment salary */
sprintf((char *)sqlstmt, "UPDATE EMP SET SAL = SAL + 1 WHERE EMPNO = 7902");
OCIStmtPrepare(stmthp, errhp, sqlstmt, strlen(sqlstmt), OCI_NTV_SYNTAX, 0);
OCIStmtExecute(svchp, stmthp, errhp, 1, 0, 0, 0, 0);
OCITransCommit(svchp, errhp, (ub4) 0);

/* update scott.emp empno=7902, increment salary again, but rollback */
OCIStmtExecute(svchp, stmthp, errhp, 1, 0, 0, 0, 0);
OCITransRollback(svchp, errhp, (ub4) 0);
}
```

関連関数

[OCITransRollback\(\)](#)

OCITransDetach()

用途

トランザクションの連結を解除します。

構文

```
sword OCITransDetach ( OCISvcCtx   *svchp,  
                       OCIError    *errhp,  
                       ub4          flags );
```

パラメータ

svchp (IN)

サービス・コンテキスト・ハンドルです。

errhp (IN)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

flags (IN)

このパラメータ用に OCI_DEFAULT の値を渡す必要があります。

コメント

サービス・コンテキスト・ハンドルからグローバル・トランザクションを連結解除します。このコールが終了した時点で、サービス・コンテキスト・ハンドルに現在連結されているトランザクションが非活動状態になります。このトランザクションは、後で OCITransStart () をコールしながら OCI_TRANS_RESUME のフラグ値を指定するときに再開できます。

トランザクションの連結が解除されると、トランザクションの開始時に OCITransStart () のタイムアウト・パラメータに指定された値を使用して、サーバーの PMON プロセスによって削除されるまでブランチを非活動状態にしておける時間が判断されます。

注意： トランザクションが同一の認可を持っている場合、トランザクションはその連結を解除したプロセス以外のプロセスによっても再開できます。トランザクションが実際に開始される前にこの関数がコールされると、この関数は何も行いません。

OCITransDetach () の使用方法を説明するコード例については、「OCITransStart ()」の説明を参照してください。

関連関数

[OCITransStart \(\)](#)

OCITransForget()

用途

完了したグローバル・トランザクションをサーバーに放棄させます。

構文

```
sword OCITransForget ( OCISvcCtx      *svchp,  
                       OCIError      *errhp,  
                       ub4            flags );
```

パラメータ

svchp (IN)

トランザクションが常駐するサービス・コンテキスト・ハンドルです。

errhp (IN)

エラー発生時の診断情報のために OCIErrorGet() に渡すエラー・ハンドルです。

flags (IN)

このパラメータ用に OCI_DEFAULT を渡す必要があります。

コメント

完了したグローバル・トランザクションを放棄します。サーバーにより、システムのペンディング・トランザクション表からトランザクションの状態が削除されます。

放棄されるトランザクションの XID をトランザクション・ハンドルの属性として設定します (OCI_ATTR_XID)。

関連関数

[OCITransCommit\(\)](#)、[OCITransRollback\(\)](#)

OCITransMultiPrepare()

用途

単一セルに複数のブランチがあるトランザクションを準備します。

構文

```
sword OCITransMultiPrepare ( OCISvcCtx   *svchp,
                              ub4          numBranches,
                              OCITrans    **txns,
                              OCIError    **errhp);
```

パラメータ

svchp (IN)

サービス・コンテキスト・ハンドルです。

numBranches (IN)

ブランチの数を指定します。この値は、次の 2 つのパラメータの配列サイズでもあります。

txns (IN)

準備するブランチのトランザクション・ハンドルの配列です。これらにはすべて `OCI_ATTR_XID` が設定されます。グローバル・トランザクション ID は同じにしてください。

errhp (IN)

エラー・ハンドルの配列です。`OCI_SUCCESS` が戻されない場合、このパラメータによってどのブランチがどのエラーを受信したか示されます。

コメント

指定のグローバル・トランザクションをコミットできるように準備します。このコールは、分散トランザクションに対してのみ有効です。このコールは、コール元がトランザクションですべてのブランチを準備する場合にかぎり使用する拡張パフォーマンス機能です。

関連関数

[OCITransPrepare\(\)](#)

OCITransPrepare()

用途

トランザクションをコミットできるように準備します。

構文

```
sword OCITransPrepare ( OCISvcCtx      *svchp,  
                        OCIError       *errhp,  
                        ub4             flags );
```

パラメータ

svchp (IN)

サービス・コンテキスト・ハンドルです。

errhp (IN)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

flags (IN)

このパラメータ用に OCI_DEFAULT を渡す必要があります。

コメント

指定のグローバル・トランザクションをコミットできるように準備します。

このコールは、グローバル・トランザクションに対してのみ有効です。

このコールは、トランザクションが変更されていない場合は OCI_SUCCESS_WITH_INFO を戻します。エラー・ハンドルは、トランザクションが読取り専用であることを示します。flags パラメータは、現在使用されていません。

関連関数

[OCITransCommit \(\)](#)、[OCITransForget \(\)](#)

OCITransRollback()

用途

カレント・トランザクションをロールバックします。

構文

```
sword OCITransRollback ( dvoid          *svchp,
                        OCIError        *errhp,
                        ub4              flags );
```

パラメータ

svchp (IN)

サービス・コンテキスト・ハンドルです。サービス・コンテキスト・ハンドル内に現在設定されているトランザクションがロールバックされます。

errhp (IN)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

flags (IN)

このパラメータ用に OCI_DEFAULT の値を渡す必要があります。

コメント

カレント・トランザクション（最後の OCITransCommit ()、または OCISessionBegin () 以降に実行された一連の文として定義されます）がロールバックされます。

アプリケーションがオブジェクト・モードで実行されている場合は、このトランザクションで修正または更新されたオブジェクト・キャッシュ内のオブジェクトも同様にロールバックされます。

現在アクティブでないグローバル・トランザクションをロールバックしようとすると、エラーになります。

例

OCITransRollback () の使用方法を説明するコード例については、[「OCITransCommit \(\)」](#) の説明を参照してください。

関連関数

[OCITransCommit \(\)](#)

OCITransStart()

用途

トランザクションの開始を設定します。

構文

```
sword OCITransStart ( OCISvcCtx      *svchp,  
                      OCIError      *errhp,  
                      uword          timeout,  
                      ub4             flags );
```

パラメータ

svchp (IN/OUT)

サービス・コンテキスト・ハンドルです。**flags** パラメータに新規トランザクションの開始が指定されている場合は、このコールの終了時点で、サービス・コンテキスト・ハンドル内のトランザクション・コンテキストが初期化されます。

errhp (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、**OCI_ERROR** が戻されます。OCIErrorGet() のコールによって診断情報を取得できます。

timeout (IN)

OCI_TRANS_RESUME が指定されたときに、トランザクションが再開できるようになるまで待機する時間（秒）です。**OCI_TRANS_NEW** が指定されると、非活動状態が **timeout** パラメータの指示する秒数続いたトランザクションは、システムによって自動的に中止されます。トランザクションは、**OCITransDetach()** による連結解除から、OCITransStart() による再開までの間、非活動状態になります。

flags (IN)

新規トランザクションが開始中か、または既存のトランザクションが再開中であるかを指定します。シリアル化可能または読取り専用ステータスも指定します。複数の値を指定できます。デフォルトでは、読取りトランザクションまたは書込みトランザクションが開始されず。フラグ値は次のとおりです。

- **OCI_TRANS_NEW** — 新規トランザクション・ブランチを開始します。デフォルトでは、密結合された移行可能なブランチが開始されます。
- **OCI_TRANS_TIGHT** — 密結合ブランチを明示的に指定します。
- **OCI_TRANS_LOOSE** — 疎結合ブランチを指定します。
- **OCI_TRANS_RESUME** — 既存のトランザクション・ブランチを再開します。
- **OCI_TRANS_READONLY** — 読取り専用トランザクションを開始します。

- OCI_TRANS_SERIALIZABLE – シリアル化可能トランザクションを開始します。
- OCI_TRANS_SEPARABLE – トランザクションは、各コール後に分離されます。

このフラグによって、トランザクションが標準トランザクションを使用して開始されたことを警告します。リリース 9.0.1 のサーバーでは、分離されたトランザクションはサポートされていません。

コードまたはトランザクション・サービスにエラーがある場合は、エラー・メッセージが表示されます。このエラーは、すでに準備されていたトランザクションで処理を試みたことを示します。

コメント

この関数は、グローバル・トランザクションまたはシリアル化可能トランザクションの開始を設定します。flags パラメータに新規トランザクションの開始が指定されている場合は、このコールの終了時点で、サービス・コンテキスト・ハンドルに現在対応付けられているトランザクション・コンテキストが初期化されます。

トランザクションの XID がトランザクション・ハンドルの属性として設定されます (OCI_ATTR_XID)。

例

次の例では、OCI トランザクション・コールを使用してグローバルなトランザクションを操作する方法を説明します。

例 1 – 様々なブランチで動作するシングル・セッション

この概念については、8-6 ページの [図 8-2 「複数のブランチ操作のセッション」](#) を参照してください。

```
int main()
{
    OCIEnv *envhp;
    OCIServer *srvhp;
    OCIError *errhp;
    OCISvcCtx *svchp;
    OCISession *usrhp;
    OCISstmt *stmthp1, *stmthp2;
    OCITrans *txnhp1, *txnhp2;
    dvoid *tmp;
    XID gxid;
    text sqlstmt[128];

    OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0, (dvoid * (*)()) 0,
                  (dvoid * (*)()) 0, (void (*)()) 0 );

    OCIHandleAlloc( (dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
                    0, (dvoid **) &tmp);
```

```
OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp );

OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
                52, (dvoid **) &tmp);
OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
                52, (dvoid **) &tmp);

OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
                52, (dvoid **) &tmp);

OCIHandleAlloc((dvoid *)envhp, (dvoid **)&stmthp1, OCI_HTYPE_STMT, 0, 0);
OCIHandleAlloc((dvoid *)envhp, (dvoid **)&stmthp2, OCI_HTYPE_STMT, 0, 0);

OCIAttrSet((dvoid *)svchp, OCI_HTYPE_SVCCTX, (dvoid *)srvhp, 0,
            OCI_ATTR_SERVER, errhp);

/* set the external name and internal name in server handle */
OCIAttrSet((dvoid *)srvhp, OCI_HTYPE_SERVER, (dvoid *) "demo", 0,
            OCI_ATTR_EXTERNAL_NAME, errhp);
OCIAttrSet((dvoid *)srvhp, OCI_HTYPE_SERVER, (dvoid *) "txn demo", 0,
            OCI_ATTR_INTERNAL_NAME, errhp);

/* allocate a user context handle */
OCIHandleAlloc((dvoid *)envhp, (dvoid **)&usrhp, (ub4) OCI_HTYPE_SESSION,
                (size_t) 0, (dvoid **) 0);

OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION, (dvoid *)"scott",
            (ub4)strlen("scott"), OCI_ATTR_USERNAME, errhp);
OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION, (dvoid *)"tiger",
            (ub4)strlen("tiger"),OCI_ATTR_PASSWORD, errhp);

OCISessionBegin (svchp, errhp, usrhp, OCI_CRED_RDBMS, 0);

OCIAttrSet((dvoid *)svchp, (ub4)OCI_HTYPE_SVCCTX,
            (dvoid *)usrhp, (ub4)0, OCI_ATTR_SESSION, errhp);

/* allocate transaction handle 1 and set it in the service handle */
OCIHandleAlloc((dvoid *)envhp, (dvoid **)&txnhp1, OCI_HTYPE_TRANS, 0, 0);
OCIAttrSet((dvoid *)svchp, OCI_HTYPE_SVCCTX, (dvoid *)txnhp1, 0,
            OCI_ATTR_TRANS, errhp);
```

```
/* start a transaction with global transaction id = [1000, 123, 1] */
gxid.formatID = 1000; /* format id = 1000 */
gxid.gtrid_length = 3; /* gtrid = 123 */
gxid.data[0] = 1; gxid.data[1] = 2; gxid.data[2] = 3;
gxid.bqual_length = 1; /* bqual = 1 */
gxid.data[3] = 1;

OCIAttrSet((dvoid *)txnhp1, OCI_HTYPE_TRANS, (dvoid *)&gxid, sizeof(XID),
           OCI_ATTR_XID, errhp);

/* start global transaction 1 with 60 second time to live when detached */
OCITransStart(svchp, errhp, 60, OCI_TRANS_NEW);

/* update scott.emp empno=7902, increment salary */
sprintf((char *)sqlstmt, "UPDATE EMP SET SAL = SAL + 1 WHERE EMPNO = 7902");
OCIStmtPrepare(stmthp1, errhp, sqlstmt, strlen(sqlstmt), OCI_NTV_SYNTAX, 0);
OCIStmtExecute(svchp, stmthp1, errhp, 1, 0, 0, 0, 0);

/* detach the transaction */
OCITransDetach(svchp, errhp, 0);

/* allocate transaction handle 2 and set it in the service handle */
OCIHandleAlloc((dvoid *)envhp, (dvoid **)&txnhp2, OCI_HTYPE_TRANS, 0, 0);
OCIAttrSet((dvoid *)svchp, OCI_HTYPE_SVCCTX, (dvoid *)txnhp2, 0,
           OCI_ATTR_TRANS, errhp);

/* start a transaction with global transaction id = [1000, 124, 1] */
gxid.formatID = 1000; /* format id = 1000 */
gxid.gtrid_length = 3; /* gtrid = 124 */
gxid.data[0] = 1; gxid.data[1] = 2; gxid.data[2] = 4;
gxid.bqual_length = 1; /* bqual = 1 */
gxid.data[3] = 1;

OCIAttrSet((dvoid *)txnhp2, OCI_HTYPE_TRANS, (dvoid *)&gxid, sizeof(XID),
           OCI_ATTR_XID, errhp);

/* start global transaction 2 with 90 second time to live when detached */
OCITransStart(svchp, errhp, 90, OCI_TRANS_NEW);

/* update scott.emp empno=7934, increment salary */
sprintf((char *)sqlstmt, "UPDATE EMP SET SAL = SAL + 1 WHERE EMPNO = 7934");
OCIStmtPrepare(stmthp2, errhp, sqlstmt, strlen(sqlstmt), OCI_NTV_SYNTAX, 0);
OCIStmtExecute(svchp, stmthp2, errhp, 1, 0, 0, 0, 0);

/* detach the transaction */
OCITransDetach(svchp, errhp, 0);
```

```

/* Resume transaction 1, increment salary and commit it */
/* Set transaction handle 1 into the service handle */
OCIAttrSet((dvoid *)svchp, OCI_HTYPE_SVCCTX, (dvoid *)txnhp1, 0,
           OCI_ATTR_TRANS, errhp);

/* attach to transaction 1, wait for 10 seconds if the transaction is busy */
/* The wait is clearly not required in this example because no other */
/* process/thread is using the transaction. It is only for illustration */
OCITransStart(svchp, errhp, 10, OCI_TRANS_RESUME);
OCISmtExecute(svchp, stmthp1, errhp, 1, 0, 0, 0, 0);
OCITransCommit(svchp, errhp, (ub4) 0);

/* attach to transaction 2 and commit it */
/* set transaction handle2 into the service handle */
OCIAttrSet((dvoid *)svchp, OCI_HTYPE_SVCCTX, (dvoid *)txnhp2, 0,
           OCI_ATTR_TRANS, errhp);
OCITransCommit(svchp, errhp, (ub4) 0);
}

```

例 2 - 同一のトランザクションを共有する複数のブランチ上で動作するシングル・セッション

```

int main()
{
    OCIEnv *envhp;
    OCIError *errhp;
    OCISvcCtx *svchp;
    OCISession *usrhp;
    OCISmt *stmthp;
    OCITrans *txnhp1, *txnhp2;
    dvoid *tmp;
    XID gxid;
    text sqlstmt[128];

    OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0, (dvoid * (*)()) 0,
                 (dvoid * (*)()) 0, (void (*)()) 0 );

    OCIHandleAlloc( (dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
                   0, (dvoid **) &tmp);

    OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp );

    OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
                   52, (dvoid **) &tmp);
    OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SERVER,
                   52, (dvoid **) &tmp);
}

```

```
OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
                52, (dvoid **) &tmp);

OCIHandleAlloc((dvoid *)envhp, (dvoid **)&stmthp, OCI_HTYPE_STMT, 0, 0);

OCIAttrSet((dvoid *)svchp, OCI_HTYPE_SVCCTX, (dvoid *)srvhp, 0,
            OCI_ATTR_SERVER, errhp);

/* set the external name and internal name in server handle */
OCIAttrSet((dvoid *)srvhp, OCI_HTYPE_SERVER, (dvoid *) "demo", 0,
            OCI_ATTR_EXTERNAL_NAME, errhp);
OCIAttrSet((dvoid *)srvhp, OCI_HTYPE_SERVER, (dvoid *) "txn demo2", 0,
            OCI_ATTR_INTERNAL_NAME, errhp);

/* allocate a user context handle */
OCIHandleAlloc((dvoid *)envhp, (dvoid **)&usrhp, (ub4) OCI_HTYPE_SESSION,
                (size_t) 0, (dvoid **) 0);

OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION, (dvoid *)"scott",
            (ub4)strlen("scott"), OCI_ATTR_USERNAME, errhp);
OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION, (dvoid *)"tiger",
            (ub4)strlen("tiger"),OCI_ATTR_PASSWORD, errhp);

OCISessionBegin (svchp, errhp, usrhp, OCI_CRED_RDBMS, 0);

OCIAttrSet((dvoid *)svchp, (ub4)OCI_HTYPE_SVCCTX,
            (dvoid *)usrhp, (ub4)0, OCI_ATTR_SESSION, errhp);

/* allocate transaction handle 1 and set it in the service handle */
OCIHandleAlloc((dvoid *)envhp, (dvoid **)&txnhp1, OCI_HTYPE_TRANS, 0, 0);
OCIAttrSet((dvoid *)svchp, OCI_HTYPE_SVCCTX, (dvoid *)txnhp1, 0,
            OCI_ATTR_TRANS, errhp);

/* start a transaction with global transaction id = [1000, 123, 1] */
gxid.formatID = 1000; /* format id = 1000 */
gxid.gtrid_length = 3; /* gtrid = 123 */
gxid.data[0] = 1; gxid.data[1] = 2; gxid.data[2] = 3;
gxid.bqual_length = 1; /* bqual = 1 */
gxid.data[3] = 1;

OCIAttrSet((dvoid *)txnhp1, OCI_HTYPE_TRANS, (dvoid *)&gxid, sizeof(XID),
            OCI_ATTR_XID, errhp);
```



```
/* start global transaction 1 with 60 second time to live when detached */
OCITransStart(svchp, errhp, 60, OCI_TRANS_NEW);

/* update scott.emp empno=7902, increment salary */
sprintf((char *)sqlstmt, "UPDATE EMP SET SAL = SAL + 1 WHERE EMPNO = 7902");
OCIStmtPrepare(stmthp, errhp, sqlstmt, strlen(sqlstmt), OCI_NTV_SYNTAX, 0);
OCIStmtExecute(svchp, stmthp, errhp, 1, 0, 0, 0, 0);

/* detach the transaction */
OCITransDetach(svchp, errhp, 0);

/* allocate transaction handle 2 and set it in the service handle */
OCIHandleAlloc((dvoid *)envhp, (dvoid **)&txnhp2, OCI_HTYPE_TRANS, 0, 0);
OCIAttrSet((dvoid *)svchp, OCI_HTYPE_SVCCTX, (dvoid *)txnhp2, 0,
           OCI_ATTR_TRANS, errhp);

/* start a transaction with global transaction id = [1000, 123, 2] */
/* The global transaction will be tightly coupled with earlier transaction */
/* There is not much practical value in doing this but the example */
/* illustrates the use of tightly-coupled transaction branches */
/* In a practical case the second transaction that tightly couples with */
/* the first can be executed from a different process/thread */

gxid.formatID = 1000; /* format id = 1000 */
gxid.gtrid_length = 3; /* gtrid = 123 */
gxid.data[0] = 1; gxid.data[1] = 2; gxid.data[2] = 3;
gxid.bqual_length = 1; /* bqual = 2 */
gxid.data[3] = 2;

OCIAttrSet((dvoid *)txnhp2, OCI_HTYPE_TRANS, (dvoid *)&gxid, sizeof(XID),
           OCI_ATTR_XID, errhp);

/* start global transaction 2 with 90 second time to live when detached */
OCITransStart(svchp, errhp, 90, OCI_TRANS_NEW);

/* update scott.emp empno=7902, increment salary */
/* This is possible even if the earlier transaction has locked this row */
/* because the two global transactions are tightly coupled */
OCIStmtExecute(svchp, stmthp, errhp, 1, 0, 0, 0, 0);

/* detach the transaction */
OCITransDetach(svchp, errhp, 0);

/* Resume transaction 1 and prepare it. This will return */
/* OCI_SUCCESS_WITH_INFO because all branches except the last branch */
/* are treated as read-only transactions for tightly-coupled transactions */
```

```
OCIAttrSet((dvoid *)svchp, OCI_HTYPE_SVCCTX, (dvoid *)txnhp1, 0,
           OCI_ATTR_TRANS, errhp);
if (OCITransPrepare(svchp, errhp, (ub4) 0) == OCI_SUCCESS_WITH_INFO)
{
    text errbuf[512];
    ub4 buflen;
    sb4 errcode;

    OCIErrorGet ((dvoid *) errhp, (ub4) 1, (text *) NULL, &errcode,
                 errbuf, (ub4) sizeof(errbuf), (ub4) OCI_HTYPE_ERROR);
    printf("OCITransPrepare - %s\n", errbuf);
}

/* attach to transaction 2 and commit it */
/* set transaction handle2 into the service handle */
OCIAttrSet((dvoid *)svchp, OCI_HTYPE_SVCCTX, (dvoid *)txnhp2, 0,
           OCI_ATTR_TRANS, errhp);
OCITransCommit(svchp, errhp, (ub4) 0);
}
```

関連関数

[OCITransDetach\(\)](#)

その他の関数

この項では、その他の OCI 関数について説明します。

表 16-7 その他の関数

関数	用途
OCIBreak() (16-172 ページ)	即時非同期ブレイクを実行します。
OCIErrorGet() (16-173 ページ)	エラーを戻します。
OCILdaToSvcCtx() (16-175 ページ)	Lda_Def をサービス・コンテキスト・ハンドルに切り替えます。
OCINlsEnvironmentVariableGet() (16-176 ページ)	NLS_LANG からキャラクタ・セット ID を、NLS_NCHAR から各国語キャラクタ・セット ID を戻します。
OCIPasswordChange() (16-178 ページ)	パスワードを変更します。
OCIReset() (16-180 ページ)	OCIBreak() の後にコールされて、非同期操作およびプロトコルをリセットします。
OCIRowidToChar() (16-181 ページ)	ユニバーサル ROWID を拡張文字 (BASE64) 表現に変換します。
OCIServerVersion() (16-182 ページ)	Oracle バージョン文字列を取得します。
OCISvcCtxToLda() (16-183 ページ)	サービス・コンテキスト・ハンドルを Lda_Def に切り替えます。
OCIUserCallbackGet() (16-184 ページ)	ハンドルに対して登録されたコールバックを識別します。
OCIUserCallbackRegister() (16-186 ページ)	ユーザー作成コールバック関数を登録します。

OCIBreak()

用途

このコールにより、サーバーに対応付けられて現在実行中である OCI 関数は即時（非同期）終了します。

構文

```
sword OCIBreak ( dvoid      *hndlp,  
                  OCIError   *errhp );
```

パラメータ

hndlp (IN/OUT)

サービス・コンテキスト・ハンドルまたはサーバー・コンテキスト・ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために OCIErrorGet() に渡すエラー・ハンドルです。

コメント

このコールにより、サーバーに対応付けられて現在実行中である OCI 関数は即時（非同期）終了します。これは通常、サーバーで長時間実行されている処理中の OCI コールを停止するために使用します。

注意： サーバーが NT システムの場合、OCIBreak() はサポートされていません。

このコールは、中止させる関数を識別するためのパラメータとして、サービス・コンテキスト・ハンドルまたはサーバー・コンテキスト・ハンドルを取得できます。

関連関数

[OCIReset\(\)](#)

OCLErrorGet()

用途

指定されたバッファ内のエラー・メッセージと Oracle エラーを戻します。

構文

```
sword OCLErrorGet ( dvoid      *hndlp,  
                   ub4        recordno,  
                   text       *sqlstate,  
                   sb4        *errcodep,  
                   text       *bufp,  
                   ub4        bufsiz,  
                   ub4        type );
```

パラメータ

hndlp (IN)

エラー・ハンドルまたは環境ハンドル (OCIEnvCreate() および OCIHandleAlloc() に関するエラー用) です。エラー・ハンドルの場合がほとんどです。

recordno (IN)

アプリケーションが情報を検索する検索先の状態レコードを示します。1 から始まります。

sqlstate (OUT)

リリース 8.x 以上ではサポートされていません。

errcodep (OUT)

戻されたエラー・コードです。

bufp (OUT)

戻されたエラー・メッセージ・テキストです。

bufsiz (IN)

エラー・メッセージの取得に使用するバッファのサイズです。バイト単位です。

type (IN)

ハンドルのタイプ (OCI_HTYPE_ERR または OCI_HTYPE_ENV) です。

コメント

与えられたバッファ内のエラー・メッセージと Oracle エラー・コードを戻します。この関数は、SQL 文はサポートしていません。*hndlp* は、ほとんどの場合、エラー・ハンドルですが、環境ハンドルの場合もあります。メッセージは、常に、環境ハンドルで設定されたエンコーディングで取得する必要があります。この関数は、1 つのエラーに対して複数の診断レコードがある場合、複数回コールできます。

エラー・ハンドルは、最初は OCIHandleAlloc() のコールで割り当てられます。

例

次のコード例は、エラー処理ルーチンで `OCIErrorGet()` を使用方法の例です。このルーチンは、OCI 関数から戻されるステータス・コードの型を出力し、エラーが発生した場合は、`OCIErrorGet()` を使用して、出力するメッセージのテキストを取り出します。

```
static void checkerr(errhp, status)
OCIError *errhp;
sword status;
{ text errbuf[512];
  ub4 buflen;
  ub4 errcode;
  switch (status)
  { case OCI_SUCCESS:
    break;
    case OCI_SUCCESS_WITH_INFO:
    printf("ErrorOCI_SUCCESS_WITH_INFO\n");
    break;
    case OCI_NEED_DATA:
    printf("ErrorOCI_NEED_DATA\n");
    break;
    case OCI_NO_DATA:
    printf("ErrorOCI_NO_DATA\n");
    break;
    case OCI_ERROR:
    OCIErrorGet ((dvoid *) errhp, (ub4) 1, (text *) NULL, &errcode,
    errbuf, (ub4) sizeof(errbuf), (ub4) OCI_HTYPE_ERROR);
    printf("Error%s\n", errbuf);
    break;
    case OCI_INVALID_HANDLE:
    printf("ErrorOCI_INVALID_HANDLE\n");
    break;
    case OCI_STILL_EXECUTING:
    printf("ErrorOCI_STILL_EXECUTE\n");
    break;
    case OCI_CONTINUE:
    printf("ErrorOCI_CONTINUE\n");
    break;
    default:
    break;
  }
}
```

関連関数

[OCIHandleAlloc\(\)](#)

OCILdaToSvcCtx()

用途

バージョン 7 の **Lda_Def** をバージョン 8 以上のサービス・コンテキスト・ハンドルに変換します。

構文

```
sword OCILdaToSvcCtx ( OCISvcCtx  **svchpp,
                      OCIError    *errhp,
                      Lda_Def     *ldap );
```

パラメータ

svchpp (IN/OUT)

サービス・コンテキスト・ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために `OCIErrorGet()` に渡すエラー・ハンドルです。

ldap (IN/OUT)

`OCISvcCtxToLda()` によりこのサービス・コンテキストから戻された Oracle7 ログイン・データ領域 (LDA) です。

コメント

Oracle7 の **Lda_Def** をバージョン 8 以上のサービス・コンテキスト・ハンドルに変換します。変換されたサービス・コンテキスト・ハンドルを `OCISvcCtxToLda()` 関数に渡すと、このコールの逆の変換ができます。

`OCILdaToSvcCtx()` コールは、`OCISvcCtxToLda()` から取得した **Lda_Def** をサービス・コンテキスト・ハンドルに戻してリセットする場合にかぎり使用します。このコールは、**Lda_def** をサービス・コンテキスト・ハンドルに戻して再開された **Lda_def** を変換する場合には使用できません。

サービス・コンテキストが **Lda_Def** に変換されている場合は、Oracle7 のコールしか使用できません。**Lda_Def** をサービス・コンテキストにリセットせずにバージョン 8 以上の OCI コールを行っても実行されません。

サーバー・ハンドルまたはサービス・コンテキスト・ハンドルの `OCI_ATTR_IN_V8_MODE` 属性によって、アプリケーションでは、そのアプリケーションの現在の実行モードが、Oracle バージョン 7 か、Oracle バージョン 8 以上かを判断できます。

関連項目： [付録 A 「ハンドルおよび記述子の属性」](#)

関連関数

`OCISvcCtxToLda()`

OCIEnvGetEnvironmentVariable()

用途

NLS_LANG からキャラクタ・セット ID を、または NLS_NCHAR から各国語キャラクタ・セット ID を戻します。

構文

```
sword OCIEnvGetEnvironmentVariable ( dvoid      *val,
                                     size_t     size,
                                     ub2        item,
                                     ub2        *charset,
                                     size_t     *rsize );
```

パラメータ

val (IN/OUT)

NLS_LANG キャラクタ・セット ID や NLS_NCHAR キャラクタ・セット ID などの NLS 環境変数の値を戻します。

size (IN)

指定の出力値のサイズを指定します。文字列データのように適用されます。情報の各ピースの最大長は、OCI_NLS_MAXBUFSZ バイトです。数値データの場合、この引数は無視されません。

item (IN)

NLS 環境変数から取得する項目を指定します。次のいずれかの値です。

OCI_NLS_CHARSET_ID — ub2 データ型の NLS_LANG キャラクタ・セット ID。

OCI_NLS_NCHARSET_ID — ub2 データ型の NLS_NCHAR キャラクタ・セット ID。

charset (IN)

取得する文字列データのキャラクタ・セット ID を指定します。0（ゼロ）の場合は、NLS_LANG が使用されます。OCI_UTF16ID はこの引数に対する有効な値です。数値データの場合、この引数は無視されます。

rsize (OUT)

バイトで示した戻り値の長さです。

戻り値

OCI_SUCCESS — 関数は正常に終了しました。

OCI_ERROR — エラーが発生しました。

コメント

NLS_NCHAR が設定されていない場合、各国語キャラクタ・セット ID は、NLS 規則に従ってキャラクタ・セット ID と同じになります。NLS_LANG が設定されていない場合は、デフォルトのキャラクタ・セット ID が戻されます。

この関数の今後の機能拡張（他の値を環境変数から取得する）を考慮に入れ、出力 val のデータ型は、dvoid へのポインタになっています。文字列データはヌル文字で終了しません。

この関数は環境ハンドルを取得しないことに注意してください。したがって、関数が戻すキャラクタ・セット ID および各国語キャラクタ・セット ID は、OCI 環境ハンドルに保存されている値ではなく、NLS_LANG および NLS_NCHAR に指定されている値です。OCI 環境ハンドルが実際に使用するキャラクタ・セット ID を取得するには、それぞれ OCI_ATTR_ENV_CHARSET および OCI_ATTR_ENV_NCHARSET の OCIAttrGet () をコールします。

関連関数

[OCIEnvNlsCreate\(\)](#)

OCIPasswordChange()

用途

このコールは、アカウントのパスワードの変更を許可します。

構文

```
sword OCIPasswordChange ( OCISvcCtx      *svchp,  
                           OCIError      *errhp,  
                           CONST text     *user_name,  
                           ub4           usernm_len,  
                           CONST text     *opasswd,  
                           ub4           opasswd_len,  
                           CONST text     *npasswd,  
                           sb4           npasswd_len,  
                           ub4           mode );
```

パラメータ

svchp (IN/OUT)

サービス・コンテキストへのハンドルです。サービス・コンテキスト・ハンドルは初期化され、サーバー・コンテキスト・ハンドルがそれに対応付けられている必要があります。

errhp (IN)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

user_name (IN)

ユーザー名を指定します。UTF-16 エンコーディングが可能です。サービス・コンテキストが認証ハンドルで初期化されている場合、ユーザー名はヌル文字で終了する必要があります。

usernm_len (IN)

user_name で指定されたユーザー名文字列の長さで、エンコーディングに関係なくバイト数で表されます。usernm_len は 0 (ゼロ) 以外にしてください。

opasswd (IN)

ユーザーの旧パスワードを指定します。UTF-16 エンコーディングが可能です。

opasswd_len (IN)

opasswd で指定された旧パスワード文字列の長さ (バイト単位) です。opasswd_len は 0 (ゼロ) 以外にしてください。

npasswd (IN)

ユーザーの新規パスワードを指定します。UTF-16 エンコーディングが可能です。パスワード複雑度検証ルーチンがユーザーのプロファイルに指定され、新規パスワードの複雑度が検証される場合、新規パスワードは検証関数の複雑度要件に一致する必要があります。

npasswd_len (IN)

npasswd で指定された新規パスワード文字列の長さ（バイト単位）です。パスワード文字列を有効にするには、npasswd_len は 0（ゼロ）以外にしてください。

mode (IN)

- OCI_DEFAULT — 環境ハンドルの設定を使用します。
- OCI_UTF16 — 環境ハンドルの設定に関係なく、UTF-16 エンコーディングを使用します。

user_name、opasswd および npasswd で使用できるエンコーディングは UTF-16 のみで、あとは使用するかしないかの選択のみです。
- OCI_AUTH — ユーザー・セッション・コンテキストが作成されていない場合は、このコールによって作成され、パスワードが変更されます。コールが終了しても、ユーザー・セッション・コンテキストは消去されません。ユーザーはログインされたままです。

ユーザー・セッション・コンテキストがすでに作成されている場合は、パスワードの変更のみが実行され、このフラグによるセッションへの影響はありません。ユーザーはログインされたままです。

コメント

このコールは、アカウントのパスワードの変更を許可します。このコールは OCISessionBegin() に類似していますが、次の点が異なります。

- ユーザー・セッションがすでに確立されている場合は、旧パスワードを使用しているアカウントを認証した後、パスワードを新規パスワードに変更します。
- ユーザー・セッションが確立されていない場合は、ユーザー・セッションを確立して旧パスワードを使用しているアカウントを認証した後、パスワードを新規パスワードに変更します。

このコールは、アカウントのパスワードが期限切れになり、OCISessionBegin() が、エラー（ORA-28001）またはパスワードが期限切れであることを示す警告を戻したときに役に立ちます。

mode または環境ハンドルによって、UTF-16 が使用されているかどうか判断されます。

関連関数

[OCISessionBegin\(\)](#)

OCIReset()

用途

割り込まれた非同期操作およびプロトコルをリセットします。非ブロック操作の処理中に OCIBreak コールが発行された場合に、コールする必要があります。

構文

```
sword OCIReset ( dvoid      *hndlp,  
                 OCIError   *errhp );
```

パラメータ

hndlp(IN)

サービス・コンテキスト・ハンドルまたはサーバー・コンテキスト・ハンドルです。

errhp (IN)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

コメント

このコールは、非ブロック・モードでのみコールされます。割り込まれた非同期操作およびプロトコルをリセットします。非ブロック操作の処理中に OCIBreak () コールが発行された場合に、コールする必要があります。

関連関数

[OCIBreak\(\)](#)

OCIRowidToChar()

用途

ユニバーサル ROWID を拡張文字（BASE64）表現に変換します。

構文

```
sword OCIRowidToChar ( OCIRowid      *rowidDesc,  
                       OraText      *outbfp,  
                       ub2           *outbflp  
                       OCIError     *errhp );
```

パラメータ

rowidDesc (IN)

OCIDescriptorAlloc() で割り当てられ、前に実行された SQL 文で挿入された ROWID 記述子です。

outbfp (OUT)

このコールが正常に実行された後、文字表現が格納されるバッファへのポインタです。

outbflp (IN/OUT)

出力バッファ長へのポインタです。実行前は、バッファ長には *outbfp* のサイズが格納されています。実行後は、変換されたバイト数が格納されます。

変換時の切捨てに備えて、*outbfp* には、変換を正常終了するために必要な長さが格納されています。エラーも戻されます。

errhp (IN)

エラー発生時の診断情報のために OCIErrorGet() に渡すエラー・ハンドルです。

コメント

この変換を行った後は、文字形式の ROWID を OCIBindByPos() コールまたは OCIBindByName() コールにバインドし、指定された ROWID の行の間合せに使用できます。

OCIServerVersion()

用途

Oracle サーバーのバージョン文字列を返します。

構文

```
sword OCIServerVersion ( dvoid          *hndlp,  
                        OCIError      *errhp,  
                        text          *bufp,  
                        ub4           bufksz,  
                        ub1           hndltype );
```

パラメータ

hndlp (IN)

サービス・コンテキスト・ハンドルまたはサーバー・コンテキスト・ハンドルです。

errhp (IN)

エラー発生時の診断情報のために OCIErrorGet () に渡すエラー・ハンドルです。

bufp (IN)

バージョン情報が戻されるバッファです。

bufksz (IN)

バッファの長さです。バイト単位です。

hndltype (IN)

関数に渡されたハンドル型です。

コメント

このコールは、Oracle サーバーのバージョン文字列を返します。Unicode の文字列が可能です（環境ハンドルで Unicode と判断された場合）。

たとえば、アプリケーションが 8.1.5 SunOS サーバー上で実行されている場合は、次のようなバージョン文字列が返ります。

```
Oracle8i Enterprise Edition Release 8.1.5.0.0 - Production  
With the Partitioning and Java options  
PL/SQL Release 8.1.5.0.0 - Production
```

関連関数

[OCIErrorGet \(\)](#)

OCISvcCtxToLda()

用途

バージョン 8 以上のサービス・コンテキスト・ハンドルとバージョン 7 の `Lda_Def` との間の切替えを行います。

構文

```
sword OCISvcCtxToLda ( OCISvcCtx   *srvhp,  
                      OCLError    *errhp,  
                      Lda_Def     *ldap );
```

パラメータ

svchp (IN/OUT)

サービス・コンテキスト・ハンドルです。

errhp (IN/OUT)

エラー発生時の診断情報のために `OCLErrorGet()` に渡すエラー・ハンドルです。

ldap (IN/OUT)

このコールによって初期化される、Oracle7 スタイルの OCI コールのためのログイン・データ領域 (LDA) です。

コメント

OCI バージョン 8 以上のサービス・コンテキスト・ハンドルと Oracle7 の `Lda_Def` との間の切替えを行います。

この関数は、サービス・コンテキストが正常に初期化された後でのみコールできます。

`Lda_Def` に変換した後のサービス・コンテキストは、リリース 7.x の OCI コール（たとえば、`obindps()`、`ofen()`）で使用できます。

注意： 同じサーバー・ハンドルを共有する複数のサービス・コンテキストがある場合、Oracle7 モードは常に 1 つのサーバー・コンテキストのみに可能です。

変換された `Lda_Def` を `OCIldaToSvcCtx()` 関数に渡すと、このコールの逆の変換ができます。

サーバー・ハンドルまたはサービス・コンテキスト・ハンドルの `OCI_ATTR_IN_V8_MODE` 属性によって、アプリケーションでは、そのアプリケーションの現在の実行モードが、Oracle バージョン 7 か、Oracle バージョン 8 以上かを判断できます。

関連項目： [付録 A「ハンドルおよび記述子の属性」](#)

関連関数

[OCIldaToSvcCtx\(\)](#)

OCIUserCallbackGet()

用途

ハンドルに対して登録されたコールバックを判断します。

構文

```
sword OCIUserCallbackGet ( dvoid    *hndlp,
                           ub4      type,
                           dvoid    *ehndlp,
                           ub4      fcode,
                           ub4      when
                           OCIUserCallback (*callbackp)
                           /*_
                               dvoid *ctxp,
                               dvoid *hndlp,
                               ub4 type,
                               ub4 fcode,
                               ub1 when,
                               sword returnCode,
                               ub4 *errnop,
                               va_list arglist
                           _*/),
                           dvoid    **ctxpp
                           OCIUcb   *ucbDesc );
```

パラメータ

hndlp(IN)

type パラメータによって指定された型のハンドルです。

type (IN)

ハンドル・タイプです。次のハンドル・タイプが有効です。

- **OCI_HTYPE_ENV** – コールバックは、環境ハンドル上で生成された *fcode* によって指定される関数の、すべてのコールに対して登録されます。

ehndlp (IN)

OCI エラー・ハンドルまたは環境ハンドルです。エラーがある場合は、*ehndlp* に記録され、**OCI_ERROR** が戻されます。**OCIErrorGet ()** のコールによって診断情報を取得できます。

fcode (IN)

OCI 関数の一意の関数コードです。関数コードは、16-190 ページの表 16-8「OCI 関数コード」にリストされています。

when (IN)

コールバックがいつ呼び出されるかを定義します。次のモードが有効です。

- OCI_CBTTYPE_ENTRY — コールバックは、OCI 関数の開始時に呼び出されます。
- OCI_CBTTYPE_EXIT — コールバックは、OCI 関数の終了前に呼び出されます。
- OCI_UCBTTYPE_REPLACE — このコールバックで OCI_CONTINUE 以外が戻された場合は、次の置換コールバックおよび OCI 関数に対する OCI コードはコールされません。かわりに、最後のコールバックの処理が開始されます。このパラメータの詳細は、16-186 ページの「[OCIUserCallbackRegister\(\)](#)」を参照してください。

callbackp (OUT)

コールバック関数ポインタへのポインタです。*fcode*、*when* および *hndlp* の値に対して現在登録されている関数を戻します。コールバックが登録されていない場合、戻り値は NULL です。

関連項目： *callbackp* のパラメータについては、16-186 ページの「[OCIUserCallbackRegister\(\)](#)」の説明を参照してください。

ctxpp (OUT)

現在登録されているコールバックに対して戻されるコンテキストへのポインタです。

ucbDesc (IN)

OCI 指定の記述子です。この記述子は、環境コールバックで OCI によって渡されます。この記述子には、コールバックを登録する場合の優先順位が格納されます。*ucbDesc* パラメータに NULL を指定すると、このコールバックの優先順位が一番高くなります。

パッケージで動的に登録されたユーザー・コールバックとは対照的に、静的に登録されたユーザー・コールバックは、使用する *ucb* 記述子がないため、NULL 記述子を使用します。

コメント

この関数は、特定のハンドルに対して登録されているコールバックを検索します。

関連項目： コールバック関数の使用制限は、9-38 ページの「[コールバック関数の制限](#)」を参照してください。

関連関数

[OCIUserCallbackRegister\(\)](#)

OCIUserCallbackRegister()

用途

ユーザー作成コールバック関数を登録します。

構文

```
sword OCIUserCallbackRegister ( dvoid    *hndlp,
                                ub4      type,
                                dvoid    *ehndlp,
                                OCIUserCallback (callback)
                                    (/*_
                                        dvoid *ctxp,
                                        dvoid *hndlp,
                                        ub4 type,
                                        ub4 fcode,
                                        ub1 when,
                                        sword returnCode,
                                        ub4 *errnop,
                                        va_list arglist
                                    _*/),
                                dvoid    *ctxp,
                                ub4      fcode,
                                ub4      when
                                OCIUcb   *ucbDesc );
```

パラメータ

hndlp(IN)

type パラメータによって指定された型のハンドルです。

type (IN)

ハンドル・タイプです。次のハンドル・タイプが有効です。

- **OCI_HTYPE_ENV** – コールバックは、環境ハンドル上で生成された *fcode* によって指定される関数の、すべてのコールに対して登録されます。

ehndlp (IN)

OCI エラー・ハンドルまたは環境ハンドルです。エラーがある場合は、*ehndlp* に記録され、**OCI_ERROR** が戻されます。**OCIErrorGet()** のコールによって診断情報を取得できます。**OCIEnvCallback** では、エラー・ハンドルを使用することができないため、環境ハンドルは *ehndlp* として渡されます。

callback (IN)

コールバック関数ポインタです。OCIUserCallback 関数プロトタイプの変数引数のリストは、OCI 関数に渡されるパラメータです。OCIUserCallback の typedef については、後で説明します。

最初のコールバックで OCI_CONTINUE 以外が戻された場合、リターン・コードは後続の最初のコールバックまたは置換コールバック（存在する場合）に渡されます。このコールバックが最新の最初のコールバックで、置換コールバックがない場合、OCI コードは実行されてリターン・コードは無視されます。

置換コールバックが OCI_CONTINUE 以外を戻した場合、後続の置換コールバックおよび OCI コードは迂回されて最後のコールバックの処理が開始されます。

最後のコールバックから OCI_CONTINUE 以外が戻された場合は、OCI 関数から戻り値が戻されます。OCI_CONTINUE が戻された場合は、OCI コードまたは置換コールバック（置換コールバックが OCI_CONTINUE を戻さずに OCI コードを迂回した場合）の戻り値がコールから戻されます。

コールバックに対して NULL 値が渡された場合は、when 値および指定されたハンドルのコールバックが削除されます。これは、NULL の ucbDesc を含む指定の ucbDesc 値に対するコールバックを登録解除する方法です。

ctxp (IN)

コールバックのコンテキスト・ポインタです。

fcode (IN)

OCI 関数の一意の関数コードです。関数コードは、16-190 ページの表 16-8「OCI 関数コード」にリストされています。

when (IN)

コールバックがいつ呼び出されるかを定義します。次のモードが有効です。

- OCI_CBTTYPE_ENTRY — コールバックは、OCI 関数の開始時に呼び出されます。
- OCI_CBTTYPE_EXIT — コールバックは、OCI 関数の終了前に呼び出されます。
- OCI_UCBTTYPE_REPLACE — このコールバックで OCI_CONTINUE 以外が戻された場合、次の置換コールバックおよび OCI 関数に対する OCI コードはコールされません。かわりに、最後のコールバックの処理が開始されます。

ucbDesc (IN)

OCI 指定の記述子です。この記述子は、環境コールバックで OCI によって渡されます。この記述子には、コールバックを登録する場合の優先順位が格納されます。ucbDesc パラメータに NULL を指定すると、このコールバックの優先順位が一番高くなります。

パッケージで動的に登録されたユーザー・コールバックとは対照的に、静的に登録されたユーザー・コールバックは、使用する ucb 記述子がないため、NULL 記述子を使用します。

コメント

この関数は、OCI 環境にユーザー作成のコールバック関数を登録する場合に使用します。

関連項目： 詳細は 9-30 ページの「[ユーザー定義コールバック関数](#)」を参照してください。

このコールバックにより、アプリケーションで次の処理を行うことができます。

1. デバッグおよびパフォーマンス測定のために OCI コールをトレースします。
2. 選択した OCI コールの後に、前処理または後処理を追加実行します。
3. 指定された関数のコードを、外部キー・データ・ソースで実行するコードに置き換えます。

OCI では、最初のコールバック、置換コールバックおよび最後のコールバックの 3 種類のコールバックがサポートされています。

この 3 種類のコールバックは、モード `OCI_UCBTYPE_ENTRY`、`OCI_UCBTYPE_REPLACE` および `OCI_UCBTYPE_EXIT` で識別されます。

次の制御フローがあります。

- 最初のコールバックの実行
- 置換コールバックの実行
- OCI コードの実行
- 最後のコールバックの実行

最初のコールバックは、プログラムが OCI 関数に入るときに実行されます。

置換コールバックは、最初のコールバック実行の後に実行されます。置換コールバックから `OCI_CONTINUE` の値が戻された場合は、後続の置換コールバックまたは通常の OCI 固有のコードが実行されます。このコールバックから `OCI_CONTINUE` 以外が戻された場合、後続の置換コールバックおよび OCI コードは実行されません。

OCI 関数が正常に実行された後または置換コールバックから `OCI_CONTINUE` 以外が戻された後は、プログラムの制御は最後のコールバック（登録されている場合）に移ります。

置換コールバックまたは最後のコールバックから `OCI_CONTINUE` 以外が戻された場合は、コールバックからのリターン・コードが対応付けられた OCI コールから戻されます。

ハンドルに対して登録されているコールバックを検索するには、`OCIUserCallbackGet()` を使用します。

OCIUserCallback typedef のプロトタイプは次のとおりです。

```
typedef sword (*OCIUserCallback)
    (dvoid    *ctxp,
     dvoid    *hndlp,
     ub4      type,
     ub4      fcode,
     ub4      when,
     sword    returnCode,
     ub4      *errnop,
     va_list  arglist
    );
```

OCIUserCallback 関数プロトタイプへのパラメータは次のとおりです。

ctxp (IN)

登録コールバック関数内で *ctxp* として渡されるコンテキストです。

hndlp(IN)

type パラメータで指定された型のハンドルです。コールバックを呼び出すためのハンドルです。OCI_HTYPE_ENV 型以外は使用しないため、環境ハンドル *env* はここに渡されます。

type (IN)

hndlp に対して登録された型です。次のハンドル・タイプが有効です。

- OCI_HTYPE_ENV – コールバックは、環境ハンドル上で生成された *fcode* によって指定される関数の、すべてのコールに対して登録されます。

fcode (IN)

OCI コールの関数コードです。関数コードは、表 16-8「OCI 関数コード」にリストされています。コールバックは、表 16-3「アドバンスド・キューイング関数およびパブリッシュ・サブスクリाइブ関数」に示されている OCI コールに対してのみ登録できます。

when (IN)

コールバックの *when* 値です。

returnCode (IN)

直前のコールバックまたは OCI コードからのリターン・コードです。1 回目の最初のコールバックでは、OCI_SUCCESS が常に渡されます。後続のコールバックでは、OCI コードまたは前のコールバックからのリターン・コードが渡されます。

errnop (IN/OUT)

1 回目の最初のコールバックをコールすると、*errnop の入力値は 0（ゼロ）になります。コールバックにより OCI_CONTINUE 以外の値が戻される場合は、*errnop にエラー番号を設定する必要があります。この値は、OCI コール内に渡されるエラー・ハンドルで設定されます。

後続のすべてのコールバックの場合、*errnop の入力値はエラー・ハンドル内のエラー番号の値です。このため、前のコールバックから OCI_CONTINUE が戻されなかった場合、前のコールバックからの *errnop の出力値はエラー・ハンドル内の値となり、この値が、ここで後続のコールバックに渡されます。反対に、前のコールバックによって OCI_CONTINUE が戻された場合、エラー・ハンドル内の値はすべてここで渡されます。

*errnop に Oracle 以外のエラー番号が戻された場合は、そのエラー番号に対して適切なテキストを戻すために、コールバックを OCIErrorGet () 関数に登録する必要があります。

arglist (IN)

これらは引数の変数値として渡される、OCI コールへのパラメータです。これらのパラメータは、va_arg を使用して間接参照する必要があります。ユーザー・コールバックのデモ・プログラムを参照してください。

関連項目： デモ・プログラムのリストは、[付録 B「OCI デモ・プログラム」](#)を参照してください。

表 16-8 OCI 関数コード

#	OCI ルーチン	#	OCI ルーチン	#	OCI ルーチン
1	OCIInitialize	33	OCITransStart	65	OCIDefineByPos
2	OCIHandleAlloc	34	OCITransDetach	66	OCIBindByPos
3	OCIHandleFree	35	OCITransCommit	67	OCIBindByName
4	OCIDescriptorAlloc	36	(使用されていません)	68	OCILobAssign
5	OCIDescriptorFree	37	OCIErrorGet	69	OCILobIsEqual
6	OCIEnvInit	38	OCILobFileOpen	70	OCILobLocatorIsInit
7	OCIServerAttach	39	OCILobFileClose	71	OCILobEnableBuffering
8	OCIServerDetach	40	(使用されていません)	72	OCILobCharSetID
9	(使用されていません)	41	(使用されていません)	73	OCILobCharSetForm
10	OCISessionBegin	42	OCILobCopy	74	OCILobFileSetName
11	OCISessionEnd	43	OCILobAppend	75	OCILobFileGetName
12	OCIPasswordChange	44	OCILobErase	76	OCILogon

表 16-8 OCI 関数コード (続き)

#	OCI ルーチン	#	OCI ルーチン	#	OCI ルーチン
13	OCIStmtPrepare	45	OCILobGetLength	77	OCILogoff
14	(使用されていません)	46	OCILobTrim	78	OCILobDisableBuffering
15	(使用されていません)	47	OCILobRead	79	OCILobFlushBuffer
16	(使用されていません)	48	OCILobWrite	80	OCILobLoadFromFile
17	OCIBindDynamic	49	(使用されていません)	81	OCILobOpen
18	OCIBindObject	50	OCIBreak	82	OCILobClose
19	(使用されていません)	51	OCIServerError	83	OCILobIsOpen
20	OCIBindArrayOfStruct	52	(使用されていません)	84	OCILobFileIsOpen
21	OCIStmtExecute	53	(使用されていません)	85	OCILobFileExists
22	(使用されていません)	54	OCIAttrGet	86	OCILobFileCloseAll
23	(使用されていません)	55	OCIAttrSet	87	OCILobCreateTemporary
24	(使用されていません)	56	OCIParmSet	88	OCILobFreeTemporary
25	OCIDefineObject	57	OCIParmGet	89	OCILobIsTemporary
26	OCIDefineDynamic	58	OCIStmtGetPieceInfo	90	OCIAQEnq
27	OCIDefineArrayOfStruct	59	OCILdaToSvcCtx	91	OCIAQDeq
28	OCIStmtFetch	60	(使用されていません)	92	OCIReset
29	OCIStmtGetBindInfo	61	OCIStmtSetPieceInfo	93	OCISvcCtxToLda
30	(使用されていません)	62	OCITransForget	94	OCILobLocatorAssign
31	(使用されていません)	63	OCITransPrepare	95	(使用されていません)
32	OCIDescribeAny	64	OCITransRollback	96	OCIAQListen

関連関数

[OCIUserCallbackGet\(\)](#)

OCI のナビゲーション関数と型関数

この章では、Oracle データベース・サーバーから取り出されたオブジェクトをナビゲートする OCI ナビゲーション関数について説明します。さらに、ここには型記述子オブジェクト (TDO) を取得する関数の説明も含まれています。

関連項目： コード例は、Oracle のインストールに含まれているデモ・プログラムを参照してください。追加情報については、[付録 B「OCI デモ・プログラム」](#)を参照してください。

この章は、次の項目で構成されています。

- [ナビゲーション関数および型関数の概要](#)
- [OCI フラッシュまたはリフレッシュ関数](#)
- [OCI オブジェクトおよびキャッシュへのマークまたはマーク解除関数](#)
- [OCI のオブジェクト・ステータス取得関数](#)
- [その他の OCI オブジェクト関数](#)
- [OCI の確保関数、確保解除関数および解放関数](#)
- [OCI の型情報アクセッサ関数](#)

ナビゲーション関数および型関数の概要

オブジェクト・ナビゲーション・パラダイムの中では、データは参照によって接続されたオブジェクトのグラフとして表現されます。このグラフの中のオブジェクトに到達するには、参照をたどります。OCI は、オブジェクトに対するナビゲーション・アクセス用インタフェースを Oracle サーバー内に備えています。この章では、これらのコールについて説明します。

OCI オブジェクト環境は、OCI_OBJECT モードでアプリケーションが OCIInitialize() をコールすると初期化されます。

関連項目： この章で説明されているコールの使用の詳細は、[第 10 章「OCI オブジェクト・リレーショナル・プログラミング」](#) および [第 13 章「オブジェクトのキャッシュおよびナビゲーション」](#) を参照してください。

オブジェクト型と存続期間

オブジェクト・インスタンスとは、Oracle データベースに定義されている型に実際の値が伴ったものです。この項では、OCI でのオブジェクト・インスタンスの記述方法について説明します。17-3 ページの [図 17-1](#) を参照してください。OCI では、オブジェクト・インスタンスは型、存続期間および参照可能性によって次のように分類されます。

- 永続オブジェクトはオブジェクト型のインスタンスの 1 つです。永続オブジェクトは、サーバー内の表の行中に常駐し、セッション（接続）の継続時間より長く存在できます。永続オブジェクトは、オブジェクト識別子を格納するオブジェクト参照によって識別できます。永続オブジェクトは、そのオブジェクト参照を確保することで取得できます。
- 一時オブジェクトはオブジェクト型のインスタンスです。一時オブジェクトは、セッション（接続）の継続時間より長く存在することはできず、一時的な計算結果を格納します。一時オブジェクトは、一時オブジェクト識別子を格納する参照によって識別できます。
- 値はユーザー定義型（オブジェクト型またはコレクション型）のインスタンスの 1 つであるか、組み込み Oracle 型のいずれかです。オブジェクトとは異なり、オブジェクト型の値は、参照ではなくメモリー・ポインタによって識別されます。

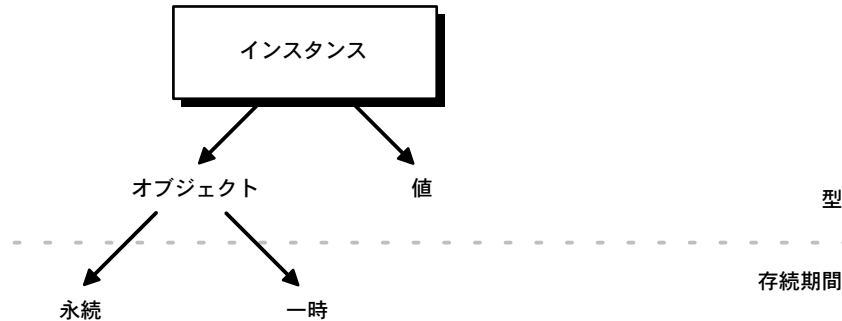
値はスタンドアロン値または埋込み値になります。スタンドアロン値は、通常、SELECT 文を発行することで取得されます。OCI では、クライアント・プログラムが SQL 文を発行することで、オブジェクト表の行を選択して値に代入することもできます。データベース内の参照可能オブジェクトは、参照によって識別できない値として表現できます。また、スタンドアロン値は、VARCHAR や RAW のように、オブジェクト内の表外格納属性にしたり、VARCHAR、RAW、オブジェクトのように、コレクション内の表外格納要素にできます。

埋込み値は、格納インスタンス内に物理的に格納されます。埋込み値は、数値やネストしたオブジェクトなどのオブジェクト内の表内格納属性、またはコレクション内の表内格納要素にできます。

すべての値は、OCI では一時的なものであるとみなされます。つまり、OCI は値のデータベースへの自動フラッシュをサポートしないため、クライアントは、値をデータベースに格納するための SQL 文を明示的に実行する必要があります。埋込み値は、格納インスタンスがフラッシュされるときにフラッシュされます。

図 17-1 に、インスタンスの型と存続期間による分類方法を示します。

図 17-1 インスタンスの型と存続期間による分類



各インスタンスの違いを次の表に示します。

	永続オブジェクト	一時オブジェクト	値
型	オブジェクト型	オブジェクト型	オブジェクト型、組み込み、コレクション
最長存続期間	オブジェクトが削除されるまで	セッション	セッション
参照可能性	あり	あり	なし
埋込み可能性	なし	なし	あり

用語

この章では、次の用語が使用されます。

- オブジェクトとは、一般に、永続オブジェクト、一時オブジェクト、オブジェクト型のスタンドアロン値、またはオブジェクト型の埋込み値のことです。
- 参照可能オブジェクトとは、永続オブジェクトまたは一時オブジェクトのことです。
- スタンドアロン・オブジェクトとは、永続オブジェクト、一時オブジェクト、またはオブジェクト型のスタンドアロン値のことです。

- 埋込みオブジェクトとは、オブジェクト型の埋込み値のことです。
- オブジェクトは、作成（新規作成）された場合、または更新済みや削除済みのマークが設定された場合に、使用済みと表現されます。

関連項目： 異なるオブジェクト型を指す用語の詳細は、10-5 ページの「永続オブジェクト、一時オブジェクトおよび値」を参照してください。

関数の構文

関数ごとに次の情報が記載されています。

用途

この関数によって実行されるアクションを簡単に説明します。

構文

関数の宣言。

コメント

この関数に関する詳細情報。関数の使用上の制約やアプリケーション内でこの関数を使用するときに役に立つ情報が記載されています。

パラメータ

この関数の各パラメータの説明。これにはパラメータのモードが含まれます。パラメータのモードには、次の3つの値があります。

モード	説明
IN	Oracle にデータを渡すパラメータ
OUT	このコールまたは後続のコールで Oracle からデータを受け取るパラメータ
IN/OUT	このコールでデータを渡し、このコールまたは後続のコールからの戻りでデータを受け取るパラメータ

戻り値

表 18-1「関数戻り値」にリストされている標準リターン・コード以外の値が関数によって戻される場合の、戻り値についての説明。

関連関数

役に立つ可能性がある追加情報を提供する関連コールのリスト。

ナビゲーション関数の戻り値

OCI ナビゲーション関数は、通常、次の値を返します。

戻り値	意味
OCI_SUCCESS	操作は成功しました。
OCI_ERROR	操作は失敗しました。関数に渡されたエラー・ハンドルに対して OCIErrorGet () をコールすることで特定のエラーを取り出せます。
OCI_INVALID_HANDLE	関数に渡された OCI ハンドルが無効です。

この章では、各関数に関する記述の後で関数固有の戻り値に関する情報を説明します。各関数から戻される特定のエラー・コードに関する情報は、次の項で記述します。

関連項目： リターン・コードおよびエラー処理の詳細は、2-31 ページの「[エラー処理](#)」を参照してください。

キャッシュ関数およびオブジェクト関数用のサーバー・ラウンドトリップ

個々の OCI キャッシュ関数およびオブジェクト関数に必要なサーバー・ラウンドトリップ回数を示す表は、[付録 C「OCI 関数のサーバー・ラウンドトリップ」](#)を参照してください。

ナビゲーション関数エラー・コード

[表 17-1](#) には、各 OCI ナビゲーション関数によって戻される外部 Oracle エラー・コードがリストされています。表の後に各エラーの内容を説明します。

表 17-1 OCI ナビゲーション関数エラー・コード

関数	発生する可能性のある ORA エラー
OCICacheFlush()	24350、21560、21705
OCICacheFree()	24350、21560、21705
OCICacheRefresh()	24350、21560、21705
OCICacheUnmark()	24350、21560、21705
OCICacheUnpin()	24350、21560、21705
OCIObjectArrayPin()	24350、21560
OCIObjectCopy()	24350、21560、21705、21710

表 17-1 OCI ナビゲーション関数エラー・コード（続き）

関数	発生する可能性のある ORA エラー
OCIObjectExists()	24350、21560、21710
OCIObjectFlush()	24350、21560、21701、21703、21708、21710
OCIObjectFree()	24350、21560、21603、21710
OCIObjectGetAttr()	21560、21600、22305
OCIObjectGetInd()	24350、21560、21710
OCIObjectGetTypeRef()	24350、21560、21710
OCIObjectIsDirty()	24350、21560、21710
OCIObjectIsLocked()	24350、21560、21710
OCIObjectLock()	24350、21560、21701、21708、21710
OCIObjectLockNoWait()	24350、21560、21701、21708、21710
OCIObjectMarkDelete()	24350、21560、21700、21701、21702、21710
OCIObjectMarkDeleteByRef()	24350、21560
OCIObjectMarkUpdate()	24350、21560、21700、21701、21710
OCIObjectNew()	24350、21560、21705、21710
OCIObjectPin()	24350、21560、21700、21702
OCIObjectPinCountReset()	24350、21560、21710
OCIObjectPinTable()	24350、21560、21705
OCIObjectRefresh()	24350、21560、21709、21710
OCIObjectSetAttr()	21560、21600、22305、22279、21601
OCIObjectUnmark()	24350、21560、21710
OCIObjectUnmarkByRef()	24350、21560
OCIObjectUnpin()	24350、21560、21710
OCIObjectGetObjectRef()	24350、21560、21710

表 17-1 の ORA エラーは、次のとおりです (*string* には文字列が入ります)。

- ORA-21560 - 引数 *string* が NULL、無効または範囲外です。
- ORA-21600 - パス式が長すぎます。
- ORA-21601 - 属性がオブジェクトではありません。
- ORA-21603 - プロパティ ID[*string*] は無効です。
- ORA-21700 - オブジェクトが存在しないか、削除マークが設定されています。
- ORA-21701 - 異なるサーバーにオブジェクトをフラッシュしようとしてしました。
- ORA-21702 - オブジェクトがインスタンス化されていないか、またはキャッシュ内でインスタンス化解除されています。
- ORA-21703 - 変更されていないオブジェクトはフラッシュできません。
- ORA-21704 - フラッシュを実行しないとキャッシュまたは接続を終了できません。
- ORA-21705 - サービス・コンテキストが無効です。
- ORA-21708 - 一時オブジェクトで不適切な操作が行われました。
- ORA-21709 - 変更されたオブジェクトをリフレッシュできません。
- ORA-21710 - 引数にはオブジェクトの有効なメモリー・アドレスが必要です。
- ORA-22279 - LOB パッファリングが使用可能な状態では、操作を実行できません。
- ORA-22305 - 属性 / メソッド / パラメータ ¥"*string*¥" が見つかりません。
- ORA-24350 - OCI コールは使用できません。

OCI フラッシュまたはリフレッシュ関数

この項では、OCI フラッシュまたはリフレッシュ関数について説明します。

表 17-2 フラッシュまたはリフレッシュ関数

関数	用途
OCI_CACHE_FLUSH() (17-9 ページ)	キャッシュ内の変更済み永続オブジェクトをサーバーにフラッシュします。
OCI_CACHE_REFRESH() (17-11 ページ)	確保されている永続オブジェクトをリフレッシュします。
OCI_OBJECT_FLUSH() (17-13 ページ)	単一の変更済み永続オブジェクトをサーバーにフラッシュします。
OCI_OBJECT_REFRESH() (17-14 ページ)	永続オブジェクトをリフレッシュします。

OCICacheFlush()

用途

この関数は、変更済み永続オブジェクトをサーバーにフラッシュします。

構文

```
sword OCICacheFlush ( OCIEnv          *env,
                      OCIError        *err,
                      CONST OCISvcCtx *svc,
                      dvoid            *context,
                      OCIRef           *(*get)
                      ( dvoid          *context,
                        ubl            *last ),
                      OCIRef           **ref );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。詳細は、「[OCIEnvCreate\(\)](#)」および「[OCIInitialize\(\)](#)」の説明を参照してください。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet() をコールして診断情報を取得します。

svc (IN)

OCI サービス・コンテキストです。

context (IN) [オプション]

クライアント・コールバック関数 `get` への引数であるユーザー・コンテキストを指定します。ユーザー・コンテキストがない場合、このパラメータは `NULL` に設定します。

get (IN) [オプション]

フラッシュを必要とする使用済みオブジェクトのバッチを取り出すイテレータとして機能するクライアント定義関数です。関数が `NULL` でない場合は、使用済みオブジェクトの参照を取得するために、この関数がコールされます。これは、クライアント関数によって `NULL` 参照が戻されるか、`last` パラメータが `TRUE` に設定されるまで繰り返されます。クライアント関数がコールされるたびに `get()` に `context` パラメータが渡されます。ユーザー・コールバックを指定しない場合、このパラメータは `NULL` にする必要があります。クライアント関数によって戻されたオブジェクトが使用済み永続オブジェクトでない場合、そのオブジェクトは無視されます。

クライアント関数から戻されるすべてのオブジェクトは、同一のサービス・コンテキストを使用する新規または確保済みのオブジェクトにしてください。そうでない場合はエラーが通知されます。戻されたオブジェクトは、使用済みとマークされた順序でフラッシュされることに注意してください。

このパラメータに NULL を渡すと（たとえば、クライアント定義関数が指定されていない場合）、指定されたサービス・コンテキストのすべての使用済み永続オブジェクトが、使用済みとマークされた順序でフラッシュされます。

ref (OUT) [オプション]

オブジェクトのフラッシュでエラーが発生すると、(**ref*) はエラーの原因となったオブジェクトのポインタになります。*ref* が NULL の場合、オブジェクトは戻りません。**ref* が NULL の場合は、参照が割り当てられ、そのオブジェクトのポインタに設定されます。

ref* が NULL でない場合は、オブジェクトの参照が指定の領域にコピーされます。エラーの原因が使用済みオブジェクトでない場合は、指定の REF が NULL 参照になるように初期化されます (OCIRefIsNull(ref*) が TRUE になります)。

REF はセッション継続時間 (OCI_DURATION_SESSION) に対して割り当てられます。アプリケーションは OCIObjectFree() 関数を使用して、割り当てられた REF を解放できます。

コメント

この関数は、オブジェクト・キャッシュの変更済み永続オブジェクトをサーバーにフラッシュします。オブジェクトは、新たに作成された順序、または更新済みや削除済みのマークが設定された順序でフラッシュされます。

関連項目： 17-13 ページ「OCIObjectFlush()」

この関数によるネットワーク・ラウンドトリップは大抵 1 回です。

関連関数

[OCIObjectFlush\(\)](#)

OCICacheRefresh()

用途

キャッシュ内のすべての確保済み永続オブジェクトをリフレッシュします。

構文

```
sword OCICacheRefresh ( OCIEnv          *env,
                        OCIError        *err,
                        CONST OCISvcCtx *svc,
                        OCIRefreshOpt   option,
                        dvoid           *context,
                        OCIRef          *( *get) (dvoid *context),
                        OCIRef          **ref );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。詳細は、「[OCIEnvCreate\(\)](#)」および「[OCIInitialize\(\)](#)」の説明を参照してください。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、err に記録され、OCI_ERROR が戻されます。OCIErrorGet () をコールして診断情報を取得します。

svc (IN)

OCI サービス・コンテキストです。

option (IN) [オプション]

OCI_REFRESH_LOADED が指定されると、そのトランザクションの間にロードされたすべてのオブジェクトをリフレッシュします。オプションが OCI_REFRESH_LOADED で、get パラメータが NULL でない場合、このパラメータは無視されます。

context (IN) [オプション]

クライアント・コールバック関数 get への引数であるユーザー・コンテキストを指定します。ユーザー・コンテキストがない場合、このパラメータは NULL に設定します。

get (IN) [オプション]

リフレッシュされる必要があるオブジェクトのバッチを取り出すイテレータとして機能するクライアント定義関数です。関数が NULL でない場合は、オブジェクトの参照を取得するために、この関数がコールされます。参照が NULL でない場合、オブジェクトはリフレッシュされます。このステップは、この関数から NULL 参照が戻されるまで繰り返されます。クライアント関数がコールされるたびに get () に context パラメータが渡されます。ユーザー・コールバックを指定しない場合、このパラメータは NULL にする必要があります。

ref (OUT) [オプション]

オブジェクトのリフレッシュでエラーが発生すると、(**ref*) はエラーの原因となったオブジェクトのポインタになります。*ref* が NULL の場合、オブジェクトは戻りません。**ref* が NULL の場合は、参照が割り当てられ、そのオブジェクトのポインタに設定されます。**ref* が NULL でない場合は、オブジェクトの参照が指定の領域にコピーされます。エラーの原因がオブジェクトでない場合は、指定の REF が NULL 参照になるように初期化されます (OCIRefIsNull(**ref*) が TRUE になります)。

コメント

この関数は、すべての確保済み永続オブジェクトをリフレッシュします。確保解除されたすべての永続オブジェクトは、オブジェクト・キャッシュから解放されます。

関連項目： リフレッシュの詳細は、「[OCIObjectRefresh\(\)](#)」の説明、および 13-12 ページの「[オブジェクト・コピーのリフレッシュ](#)」を参照してください。

注意： オブジェクトがリフレッシュされると、それらのオブジェクトの 2 次メモリーがメモリー内の別の場所に移動することがあります。このため、このコール前に保存された属性へのポインタが無効になる可能性があります。2 次メモリーを使用する属性には、**OCIString ***、**OCIColl ***、**OCIRaw *** などがあります。

関連関数

[OCIObjectRefresh\(\)](#)

OCIObjectFlush()

用途

変更済み永続オブジェクトをサーバーにフラッシュします。

構文

```
sword OCIObjectFlush ( OCIEnv      *env,
                      OCIError    *err,
                      dvoid       *object );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。詳細は、[「OCIEnvCreate\(\)」](#) および [「OCIInitialize\(\)」](#) の説明を参照してください。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、err に記録され、OCI_ERROR が戻されます。OCIErrorGet () をコールして診断情報を取得します。

object (IN)

永続オブジェクトへのポインタです。このコールの前に、オブジェクトが確保されることが必要です。

コメント

この関数は、変更済み永続オブジェクトをサーバーにフラッシュします。フラッシュ時には、オブジェクトの排他ロックが暗黙的に取得されます。オブジェクトがサーバーに書き込まれるときにトリガーを起動できます。この関数は、一時オブジェクトと値、および未修正の永続オブジェクトに対して、エラーを戻します。

オブジェクトは、サーバーでトリガーによって変更できます。キャッシュにあるオブジェクトとデータベースとの間の整合性を保つために、アプリケーションではキャッシュにあるオブジェクトの解放やリフレッシュを行うことができます。

フラッシュするオブジェクトに内部 LOB 属性があり、その LOB 属性が OCIObjectCopy ()、OCILobAssign () または OCILobLocatorAssign () によって変更されたり、別の LOB ロケータの割当てによって変更された場合は、フラッシュによって、割当て時にソース LOB 内に存在していた LOB 値のコピーか、内部 LOB のロケータまたはオブジェクトのコピーが生成されます。

関連項目： LOB 関数の詳細は、16-22 ページの [「LOB 関数」](#) を参照してください。

関連関数

[OCIObjectPin\(\)](#)、[OCICacheFlush\(\)](#)

OCIObjectRefresh()

用途

カレント・データベースの最新のスナップショットから永続オブジェクトをリフレッシュします。

構文

```
sword OCIObjectRefresh ( OCIEnv      *env,
                        OCIError    *err,
                        dvoid        *object );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。詳細は、「[OCIEnvCreate\(\)](#)」および「[OCIInitialize\(\)](#)」の説明を参照してください。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet() をコールして診断情報を取得します。

object (IN)

永続オブジェクトへのポインタです。すでに確保されていることが必要です。

コメント

この関数は、サーバー内の最新のスナップショットから取り出されたデータを使用して、オブジェクトをリフレッシュします。オブジェクトは、オブジェクト・キャッシュ内のオブジェクトとサーバー側のオブジェクトとの間に整合性がなくなったときにリフレッシュされる必要があります。

注意： オブジェクトをサーバーにフラッシュすると、トリガーを起動してサーバー内のさらに多くのオブジェクトを変更できます。その場合、オブジェクト・キャッシュ内にある（トリガーによって変更したオブジェクトと）同一のオブジェクトは最新の状態ではなくなり、リフレッシュしないとロックまたはフラッシュできません。

これは、ユーザーが SQL 文または PL/SQL プロシージャを発行してサーバー内のオブジェクトを変更したときに発生します。

注意： 最後のフラッシュ以降にオブジェクト（使用済みオブジェクト）に対して行われた変更は、マークされていないオブジェクトがこの関数でリフレッシュされると失われます。

リフレッシュの後、オブジェクトの様々なメタ属性フラグと継続時間が、次のように変更されます。

オブジェクト属性	リフレッシュ後の状態
既存	適切な値に設定
確保済み	変更なし
割当て時間	変更なし
確保継続時間	変更なし

リフレッシュされたオブジェクトは、その場で置き換えられます。オブジェクトがその場で置き換えられると、オブジェクトのトップレベル・メモリーが再利用されるため、新しいデータが同じメモリー・アドレスにロードされるようになります。NULL インジケータ構造体のトップレベル・メモリーも再利用されます。トップレベル・メモリーとは異なり、2 次メモリーは解放され、再割当てが行われます。

ポインタはオブジェクトのリフレッシュ後に無効になる場合があるため、2 次メモリーのアドレスをローカル変数に割り当てるときなど、2 次メモリー・チャンクへのポインタを保持する機能の記述時には注意が必要です。

この関数は、一時オブジェクトまたは値に対して何も影響を与えません。

関連関数

[OCICacheRefresh\(\)](#)

OCI オブジェクトおよびキャッシュへのマークまたはマーク解除関数

この項では、OCI オブジェクトおよびキャッシュへのマークまたはマーク解除関数について説明します。

表 17-3 オブジェクトおよびキャッシュへのマークまたはマーク解除関数

関数	用途
OCICacheUnmark() (17-17 ページ)	キャッシュ内のオブジェクトのマークを解除します。
OCIObjectMarkDelete() (17-18 ページ)	オブジェクトに削除済みのマークを付けます。または、値インスタンスを削除します。
OCIObjectMarkDeleteByRef() (17-19 ページ)	参照によって指定されたオブジェクトに削除マークを設定します。
OCIObjectMarkUpdate() (17-20 ページ)	オブジェクトに更新済み / 使用済みのマークを付けます。
OCIObjectUnmark() (17-22 ページ)	オブジェクトのマークを解除します。
OCIObjectUnmarkByRef() (17-23 ページ)	参照によって指定されたオブジェクトのマークを解除します。

OCICacheUnmark()

用途

オブジェクト・キャッシュ内のすべての使用済みオブジェクトのマークを解除します。

構文

```
sword OCICacheUnmark ( OCIEnv          *env,  
                       OCIError        *err,  
                       CONST OCISvcCtx  *svc );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。詳細は、[「OCIEnvCreate\(\)」](#) および [「OCIInitialize\(\)」](#) の説明を参照してください。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet() をコールして診断情報を取得します。

svc (IN)

OCI サービス・コンテキストです。

コメント

接続が指定された場合、この関数は、その接続内のすべての使用済みオブジェクトのマークを解除します。それ以外の場合は、キャッシュ内のすべての使用済みオブジェクトのマークを解除します。

関連項目： オブジェクトのマーク解除の詳細は、17-22 ページの [「OCIObjectUnmark\(\)」](#) を参照してください。

関連関数

[OCIObjectUnmark\(\)](#)

OCIObjectMarkDelete()

用途

スタンドアロン・インスタンスへのポインタを指定して、そのインスタンスに削除マークを設定します。

構文

```
sword OCIObjectMarkDelete ( OCIEnv      *env,
                             OCIError    *err,
                             dvoid       *instance );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。詳細は、「[OCIEnvCreate\(\)](#)」および「[OCIInitialize\(\)](#)」の説明を参照してください。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet() をコールして診断情報を取得します。

instance (IN)

インスタンスへのポインタです。これはスタンドアロン型にする必要があり、オブジェクトの場合は確保されることも必要です。

コメント

この関数は、スタンドアロン・インスタンスへのポインタを受け取り、そのオブジェクトに削除マークを設定します。オブジェクトは、次の規則に従って解放されます。

永続オブジェクト

オブジェクトに削除マークが設定されます。オブジェクトのメモリーは解放されません。オブジェクトは、そのオブジェクトがフラッシュされたときにサーバー内で削除されます。

一時オブジェクト

オブジェクトに削除マークが設定されます。オブジェクトのメモリーは解放されません。

値

この関数は、ただちに値を解放します。

関連関数

[OCIObjectMarkDeleteByRef\(\)](#)、[OCIObjectGetProperty\(\)](#)

OCIObjectMarkDeleteByRef()

用途

オブジェクトへの参照を指定して、そのオブジェクトに削除マークを設定します。

構文

```
sword OCIObjectMarkDeleteByRef ( OCIEnv          *env,  
                                OCIError        *err,  
                                OCIRef          *object_ref );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。詳細は、[「OCIEnvCreate\(\)」](#) および [「OCIInitialize\(\)」](#) の説明を参照してください。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、err に記録され、OCI_ERROR が戻されます。OCIErrorGet() をコールして診断情報を取得します。

object_ref (IN)

削除されるオブジェクトへの参照です。

コメント

この関数は、オブジェクトへの参照を受け取り、*object_ref* によって指定されたオブジェクトに削除マークを設定します。オブジェクトはマーク付けされて、次のように解放されます。

永続オブジェクト

オブジェクトがロードされていない場合は、一時オブジェクトが作成され、それに対して削除マークが設定されます。それ以外の場合は、オブジェクトに削除マークが設定されます。

オブジェクトは、そのオブジェクトがフラッシュされたときにサーバー内で削除されます。

一時オブジェクト

オブジェクトに削除マークが設定されます。オブジェクトは、確保が解除されるまで解放されません。

関連関数

[OCIObjectMarkDelete\(\)](#)、[OCIObjectGetProperty\(\)](#)

OCIObjectMarkUpdate()

用途

永続オブジェクトに更新済みまたは使用済みマークを設定します。

構文

```
sword OCIObjectMarkUpdate ( OCIEnv          *env,  
                             OCIError       *err,  
                             dvoid          *object );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。詳細は、[「OCIEnvCreate\(\)」](#) および [「OCIInitialize\(\)」](#) の説明を参照してください。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet() をコールして診断情報を取得します。

object (IN)

永続オブジェクトへのポインタです。すでに確保されていることが必要です。

コメント

この関数は、永続オブジェクトに更新済みまたは使用済みマークを設定します。オブジェクトの型に応じて、次の規則が適用されます。オブジェクトの使用済み状態は、[OCIObjectIsLocked\(\)](#) をコールすることによりチェックできます。

永続オブジェクト

この関数は、指定された永続オブジェクトに更新済みマークを設定します。

永続オブジェクトは、オブジェクト・キャッシュがフラッシュされたときにサーバーに書き込まれます。この関数は、オブジェクトのロックもフラッシュも行いません。削除済みオブジェクトを更新しようとすると、エラーが発生します。

オブジェクトの更新済みマークの設定とフラッシュが実行された後、オブジェクトがフラッシュ後も使用済みになっている場合は、この関数を再度コールしてオブジェクトに更新済みマークを設定する必要があります。

一時オブジェクト

この関数は、指定された一時オブジェクトに更新済みマークを設定します。一時オブジェクトは、サーバーには書き込まれません。削除済みオブジェクトを更新しようとすると、エラーが発生します。

値

値に対しては何も行われません。

関連項目： この関数の使用方法の詳細は、10-14 ページの「[オブジェクトのマークおよび変更のフラッシュ](#)」を参照してください。

関連関数

`OCIObjectPin()`、`OCIObjectGetProperty()`、`OCIObjectIsDirty()`、`OCIObjectUnmark()`

OCIObjectUnmark()

用途

オブジェクトの使用済みマークを解除します。

構文

```
sword OCIObjectUnmark ( OCIEnv      *env,  
                        OCIError    *err,  
                        dvoid        *object );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。詳細は、「[OCIEnvCreate\(\)](#)」および「[OCIInitialize\(\)](#)」の説明を参照してください。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。`OCIErrorGet()` をコールして診断情報を取得します。

object (IN)

永続オブジェクトへのポインタです。オブジェクトは確保されることが必要です。

コメント

永続オブジェクトと一時オブジェクト

この関数は、指定された永続オブジェクトの使用済みマークを解除します。オブジェクトに対して行われた変更は、サーバーに書き込まれません。オブジェクトにロック・マークが設定されている場合は、ロックされたままになります。オブジェクトに対してこれまで実行された変更は、暗黙的に元に戻されます。

値

値に対しては何も行われません。これは、値に対して関数をコールしても何の影響がないことを意味します。

関連関数

[OCIObjectUnmarkByRef\(\)](#)

OCIObjectUnmarkByRef()

用途

オブジェクトへの REF を指定して、オブジェクトの使用済みマークを解除します。

構文

```
sword OCIObjectUnmarkByRef ( OCIEnv      *env,  
                             OCIError    *err,  
                             OCIRef      *ref );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。詳細は、「[OCIEnvCreate\(\)](#)」および「[OCIInitialize\(\)](#)」の説明を参照してください。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、err に記録され、OCI_ERROR が戻されます。OCIErrorGet () をコールして診断情報を取得します。

ref (IN)

オブジェクトの参照です。オブジェクトは確保されることが必要です。

コメント

この関数は、オブジェクトの使用済みマークを解除します。オブジェクトへの REF を引数として取ることを除けば、この関数は [OCIObjectUnmark\(\)](#) と同じです。

永続オブジェクトと一時オブジェクト

この関数は、指定された永続オブジェクトの使用済みマークを解除します。オブジェクトに対して行われた変更は、サーバーに書き込まれません。オブジェクトにロック・マークが設定されている場合は、ロックされたままになります。オブジェクトに対してこれまで実行された変更は、暗黙的に元に戻されます。

値

値に対しては何も行われません。

関連関数

[OCIObjectUnmark\(\)](#)

OCI のオブジェクト・ステータス取得関数

この項では、OCI のオブジェクト・ステータス取得関数について説明します。

表 17-4 オブジェクト・ステータス取得関数

関数	用途
OCIObjectExists() (17-25 ページ)	インスタンスの存在ステータスを取得します。
OCIObjectGetProperty() (17-26 ページ)	特定のオブジェクト・プロパティのステータスを取得します。
OCIObjectIsDirty() (17-31 ページ)	インスタンスの使用済みステータスを取得します。
OCIObjectIsLocked() (17-31 ページ)	インスタンスのロック状態を取得します。

OCIObjectExists()

用途

スタンドアロン・インスタンスの存在メタ属性を戻します。

構文

```
sword OCIObjectExists ( OCIEnv          *env,  
                        OCIError        *err,  
                        dvoid            *ins,  
                        boolean          *exist );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。詳細は、[「OCIEnvCreate\(\)」](#) および [「OCIInitialize\(\)」](#) の説明を参照してください。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet() をコールして診断情報を取得します。

ins (IN)

インスタンスへのポインタです。オブジェクトの場合は、確保されることが必要です。

exist (OUT)

存在状況の戻り値です。

コメント

この関数は、インスタンスの存在を戻します。インスタンスが値の場合、この関数は常に TRUE を戻します。インスタンスは、スタンドアロン型の永続または一時オブジェクトにしてください。

関連項目： オブジェクトのメタ属性の詳細は、10-16 ページの [「オブジェクトのメタ属性」](#) を参照してください。

関連関数

[OCIObjectPin\(\)](#)

OCIObjectGetProperty()

用途

オブジェクトの指定されたプロパティを取り出します。

構文

```
sword OCIObjectGetProperty ( OCIEnv          *envh,  
                             OCIError        *errh,  
                             CONST dvoid     *obj,  
                             OCIObjectPropId propertyId,  
                             dvoid          *property,  
                             ub4            *size );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。詳細は、[「OCIEnvCreate\(\)」](#) および [「OCIInitialize\(\)」](#) の説明を参照してください。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet() をコールして診断情報を取得します。

obj (IN)

そのプロパティが戻されるオブジェクトです。

propertyId (IN)

目的のプロパティを識別する識別子です。

property (OUT)

目的のプロパティがコピーされるバッファです。

size (IN/OUT)

入力の際は、コール元から渡されるプロパティ・バッファのサイズがこのパラメータによって指定されます。

出力の際は、戻されるプロパティのサイズがバイト単位でここに格納されます。このパラメータは、`OCI_OBJECTPROP_SCHEMA`、`OCI_OBJECTPROP_TABLE` などの文字列型のプロパティのみに必要です。非文字列型のプロパティの場合は、サイズが固定しているため、このパラメータは無視されます。

コメント

この関数によりオブジェクトの指定されたプロパティが戻されます。目的のプロパティは `propertyId` で識別されます。プロパティの値は `property` にコピーされ、文字列型プロパティの場合は、文字列のサイズが `size` によって戻されます。

オブジェクトは、その存続期間と参照可能性に基づいて永続、一時および値に分類されます。プロパティのいくつかは永続オブジェクトのみに適用され、また他のプロパティは永続オブジェクトと一時オブジェクト（の両方）に適用されます。指定したオブジェクトに適用されないプロパティを取得しようとする、エラーが戻されます。このようなエラーを回避するには、最初にそのオブジェクト（の `OCI_OBJECTPROP_LIFETIME` プロパティ）が永続、一時、値のどれであるかをチェックしてから、他のプロパティを適切に問い合わせるようにします。

様々なプロパティ ID とそれに対応する `property` 引数の型を次に示します。

OCI_OBJECTPROP_LIFETIME

このプロパティによって、指定されたオブジェクトが永続オブジェクト、一時オブジェクト、または値のインスタンスかのいずれであるが識別されます。`property` 引数は `OCIObjectLifetime` 型の変数へのポインタにしてください。可能な値は、次のとおりです。

- `OCI_OBJECT_PERSISTENT`
- `OCI_OBJECT_TRANSIENT`
- `OCI_OBJECT_VALUE`

OCI_OBJECTPROP_SCHEMA

このプロパティによって、オブジェクトが存在する表のスキーマ名が戻されます。指定されたオブジェクトが一時インスタンスまたは値を指し示している場合は、エラーが戻されます。スキーマ名の保持に入力バッファのサイズが足りない場合は、エラーが戻され、エラー・メッセージで必要なサイズが表示されます。成功した場合は、`size` によって、戻されたスキーマ名のサイズがバイト単位で戻されます。`property` 引数は、`text` 型の配列にしてください。コール元は、`size` をバイト単位で配列のサイズに設定する必要があります。

OCI_OBJECTPROP_TABLE

このプロパティによってそのオブジェクトが存在する表名が戻されます。指定されたオブジェクトが一時インスタンスまたは値を指し示している場合は、エラーが戻されます。表名の保持に入力バッファが足りない場合は、エラーが戻され、エラー・メッセージで必要なサイズが表示されます。成功した場合は、`size` によって、戻された表名のサイズがバイト単位で戻されます。`property` 引数は、`text` 型の配列にしてください。コール元では、`size` を配列のサイズにバイト単位で設定する必要があります。

OCI_OBJECTPROP_PIN_DURATION

このプロパティによってオブジェクトの確保継続時間が戻されます。指定されたオブジェクトが値のインスタンスを指し示している場合は、エラーが戻されます。`property` 引数は `OCIDuration` 型の変数へのポインタにしてください。有効な値は、次のとおりです。

- OCI_DURATION_SESSION
- OCI_DURATION_TRANS

関連項目： 継続時間については、13-15 ページの「[オブジェクト継続時間](#)」を参照してください。

OCI_OBJECTPROP_ALLOC_DURATION

このプロパティによって、オブジェクトの割当て時間が戻されます。`property` 引数は `OCIDuration` 型の変数へのポインタにしてください。次の値が有効です。

- OCI_DURATION_SESSION
- OCI_DURATION_TRANS

継続時間については、13-15 ページの「[オブジェクト継続時間](#)」を参照してください。

OCI_OBJECTPROP_LOCK

このプロパティによってオブジェクトのロック状況が戻されます。可能なロック状況は `OCILockOpt` によって列挙されます。指定されたオブジェクトが一時インスタンスまたは値インスタンスを指し示している場合は、エラーが戻されます。`property` 引数は `OCILockOpt` 型の変数へのポインタにしてください。オブジェクトのロック状況は、`OCIObjectIsLocked()` 関数をコールしても取り出せます。次の値が有効です。

- OCI_LOCK_NONE — ロックなし
- OCI_LOCK_X — 排他ロック
- OCI_LOCK_X_NOWAIT — NOWAIT オプションを使用した排他ロック

関連項目： NOWAIT オプションの詳細は、13-14 ページの「[NOWAIT オプションを使用したロック](#)」を参照してください。

OCI_OBJECTPROP_MARKSTATUS

このプロパティによって使用済みステータスが戻され、オブジェクトが新規のオブジェクトであるか、更新されたオブジェクトであるか、削除されたオブジェクトであるかが示されます。指定されたオブジェクトが一時インスタンスまたは値インスタンスを指し示している場合は、エラーが戻されます。*property* 引数は **OCIObjectMarkStatus** 型にしてください。次の値が有効です。

- OCI_OBJECT_NEW
- OCI_OBJECT_DELETED
- OCI_OBJECT_UPDATED

次のマクロを使用してマーク状態をテストすることが可能です。

- OCI_OBJECT_IS_UPDATED (flag)
- OCI_OBJECT_IS_DELETED (flag)
- OCI_OBJECT_IS_NEW (flag)
- OCI_OBJECT_IS_DIRTY (flag)

OCI_OBJECTPROP_VIEW

このプロパティによって指定されたオブジェクトがビュー・オブジェクトであるかどうか識別されます。戻されたプロパティ値が **TRUE** の場合、そのオブジェクトはビューであり、そうでない場合はビューではありません。指定されたオブジェクトが一時インスタンスまたは値インスタンスを指し示している場合は、エラーが戻されます。この *property* 引数はブール型にする必要があります。

関連関数

[OCIObjectLock\(\)](#)、[OCIObjectMarkDelete\(\)](#)、[OCIObjectMarkUpdate\(\)](#)、[OCIObjectPin\(\)](#)、[OCIObjectPin\(\)](#)

OCIObjectIsDirty()

用途

オブジェクトに使用済みマークが設定されているかどうかをチェックします。

構文

```
sword OCIObjectIsDirty ( OCIEnv      *env,  
                          OCIError    *err,  
                          dvoid        *ins,  
                          boolean      *dirty );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。詳細は、「[OCIEnvCreate\(\)](#)」および「[OCIInitialize\(\)](#)」の説明を参照してください。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、err に記録され、OCI_ERROR が戻されます。OCIErrorGet() をコールして診断情報を取得します。

ins (IN)

インスタンスへのポインタです。

dirty (OUT)

使用済みステータスに対する値を戻します。

コメント

この関数に渡すインスタンスはスタンドアロンにしてください。インスタンスがオブジェクトの場合、そのインスタンスは確保されることが必要です。

この関数は、インスタンスの使用済みステータスを戻します。インスタンスが値の場合、この関数は使用済みステータスに対して常に FALSE を戻します。

関連関数

[OCIObjectMarkUpdate\(\)](#)、[OCIObjectGetProperty\(\)](#)

OCIObjectIsLocked()

用途

オブジェクトのロック状態を取得します。

構文

```
sword OCIObjectIsLocked ( OCIEnv      *env,  
                          OCIError    *err,  
                          dvoid       *ins,  
                          boolean     *lock );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。詳細は、[「OCIEnvCreate\(\)」](#) および [「OCIInitialize\(\)」](#) の説明を参照してください。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet() をコールして診断情報を取得します。

ins (IN)

インスタンスへのポインタです。インスタンスはスタンドアロンであることが必要で、インスタンスがオブジェクトの場合は確保されることも必要です。

lock (OUT)

ロック状態に対する値を戻します。

コメント

この関数は、インスタンスのロック状態を戻します。インスタンスが値の場合、この関数は常に `FALSE` を戻します。

関連関数

[OCIObjectLock\(\)](#)、[OCIObjectGetProperty\(\)](#)

その他の OCI オブジェクト関数

この項では、その他のオブジェクト関数について説明します。

表 17-5 その他のオブジェクト関数

関数	用途
OCIObjectCopy() (17-33 ページ)	1 つのインスタンスを別のインスタンスにコピーします。
OCIObjectGetAttr() (17-35 ページ)	オブジェクト属性を取得します。
OCIObjectGetInd() (17-37 ページ)	インスタンスの NULL インジケータ構造体を取得します。
OCIObjectGetObjectRef() (17-38 ページ)	指定のオブジェクトへの参照を戻します。
OCIObjectGetTypeRef() (17-39 ページ)	インスタンスの TDO への参照を取得します。
OCIObjectLock() (17-40 ページ)	永続オブジェクトをロックします。
OCIObjectLockNoWait() (17-41 ページ)	永続オブジェクトをロックしますが、ロック待機は行いません。
OCIObjectPin() (17-56 ページ)	新規インスタンスを作成します。
OCIObjectSetAttr() (17-47 ページ)	オブジェクト属性を設定します。

OCIObjectCopy()

用途

ソース・インスタンスを宛先インスタンスにコピーします。

構文

```
sword OCIObjectCopy ( OCIEnv          *env,
                      OCIError        *err,
                      CONST OCISvcCtx *svc,
                      dvoid            *source,
                      dvoid            *null_source,
                      dvoid            *target,
                      dvoid            *null_target,
                      OCIType          *tdo,
                      OCIDuration      duration,
                      ub1               option );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。詳細は、第 15 章の「[OCIEnvCreate\(\)](#)」および「[OCIInitialize\(\)](#)」の説明を参照してください。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、err に記録され、OCI_ERROR が戻されます。OCIErrorGet () をコールして診断情報を取得します。

svc (IN)

OCI サービス・コンテキスト・ハンドルで、コピー処理が発生するサービス・コンテキストを指定します。

source (IN)

ソース・インスタンスへのポインタです。ポインタがオブジェクトの場合は、確保する必要があります。

関連項目： 17-56 ページ「[OCIObjectPin\(\)](#)」

null_source (IN)

ソース・オブジェクトの NULL インジケータ構造体へのポインタです。

target (IN)

ターゲット・インスタンスへのポインタです。ポインタがオブジェクトの場合は、確保する必要があります。

null_target (IN)

ターゲット・オブジェクトの NULL インジケータ構造体へのポインタです。

tdo (IN)

ソースとターゲットの両方に対する TDO です。OCIDescribeAny() で取り出せます。

duration (IN)

ターゲット・メモリーの割当て時間です。

option (IN)

このパラメータは使用されていません。0（ゼロ）または OCI_DEFAULT を渡します。

コメント

この関数は、*source* インスタンスの内容を *target* インスタンスにコピーします。この関数は、次のすべてがコピーされるディープ・コピーを実行します。

- すべての最上位属性（後述の例外参照）
- 最上位属性から到達可能な（ソースの）すべての 2 次メモリー
- インスタンスの NULL インジケータ構造体

メモリーは、*duration* パラメータに指定された継続時間の間、割り当てられます。

次のデータ項目はコピーされません。

- *option* パラメータにオプション OCI_OBJECTCOPY_NOREF が指定されている場合、ソース内のすべての参照はコピーされません。ターゲット内の参照は NULL に設定されます。
- 属性が内部 LOB の場合は、ソース・オブジェクトの LOB ロケータのみがコピーされません。LOB データのコピーは OCIObjectFlush() がコールされるまで作成されません。ターゲット・オブジェクトがフラッシュされるまで、ソースとターゲットのロケータはともに同じ LOB 値を参照します。

ターゲットまたはターゲットの格納インスタンスは、事前に作成されることが必要です。これは、ターゲット・オブジェクトの存在の有無に応じて、OCIObjectNew() または OCIObjectPin() を使用して実行できます。

source インスタンスと *target* インスタンスは、同じ型にしてください。ソースとターゲットが別々のデータベースに置かれている場合は、両方のデータベースに同じ型が存在することが必要です。

関連関数

[OCIObjectPin\(\)](#)

OCIObjectGetAttr()

用途

オブジェクト属性を取り出します。

構文

```

sword OCIObjectGetAttr ( OCIEnv          *env,
                        OCIError        *err,
                        dvoid           *instance,
                        dvoid           *null_struct,
                        struct OCIType  *tdo,
                        CONST OraText  **names,
                        CONST ub4      *lengths,
                        CONST ub4      name_count,
                        CONST ub4      *indexes,
                        CONST ub4      index_count,
                        OCIInd          *attr_null_status,
                        dvoid           **attr_null_struct,
                        dvoid           **attr_value,
                        struct OCIType  **attr_tdo );

```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。詳細は、「[OCIEnvCreate\(\)](#)」および「[OCIInitialize\(\)](#)」の説明を参照してください。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet () をコールして診断情報を取得します。

instance (IN)

オブジェクトへのポインタです。

null_struct (IN)

オブジェクトまたは配列の NULL インジケータ構造体です。

tdo (IN)

TDO へのポインタです。

names (IN)

属性名の配列です。パス式に属性名を指定します。

lengths (IN)

属性名長 (バイト単位) の配列です。

name_count (IN)

配列 *names* の要素数です。

indexes (IN) [オプション]

現行ではサポートされていません。(ub4 *)0 で渡します。

index_count (IN) [オプション]

現行ではサポートされていません。(ub4)0 で渡します。

attr_null_status (OUT)

属性タイプが基本形の場合、属性は NULL ステータスです。

attr_null_struct (OUT)

オブジェクトまたはコレクション属性の NULL インジケータ構造体です。

attr_value (OUT)

属性値へのポインタです。

attr_tdo (OUT)

属性の TDO へのポインタです。

コメント

この関数は、オブジェクトまたは配列から値を取得します。パラメータ *instance* がオブジェクトを指し示している場合は、パス式によってそのオブジェクトの属性の位置が指定されます。オブジェクトが確保され、戻された値はオブジェクトの確保が解除されるまで有効であるとみなされます。

関連関数

[OCIObjectSetAttr\(\)](#)

OCIObjectGetInd()

用途

スタンドアロン・インスタンスの NULL インジケータ構造体を取得します。

構文

```
sword OCIObjectGetInd ( OCIEnv      *env,  
                        OCIError    *err,  
                        dvoid        *instance,  
                        dvoid        **null_struct );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。詳細は、「[OCIEnvCreate\(\)](#)」および「[OCIInitialize\(\)](#)」の説明を参照してください。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet() をコールして診断情報を取得します。

instance (IN)

取得する NULL インジケータに対応するインスタンスへのポインタです。インスタンスは、スタンドアロン型にしてください。`instance` がオブジェクトの場合は、事前に確保されることが必要です。

null_struct (OUT)

インスタンスの NULL インジケータ構造体です。

コメント

なし

関連関数

[OCIObjectPin\(\)](#)

OCIObjectGetObjectRef()

用途

指定された永続オブジェクトへの参照を戻します。

構文

```
sword OCIObjectGetObjectRef ( OCIEnv      *env,
                              OCIError    *err,
                              dvoid       *object,
                              OCIRef      *object_ref );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。詳細は、[「OCIEnvCreate\(\)」](#) および [「OCIInitialize\(\)」](#) の説明を参照してください。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、err に記録され、OCI_ERROR が戻されます。OCIErrorGet() をコールして診断情報を取得します。

object (IN)

永続オブジェクトへのポインタです。事前に確保されることが必要です。

object_ref (OUT)

object に指定されているオブジェクトへの参照です。参照は、事前に割り当てられることが必要です。これは、OCIObjectNew() によって実行できます。

コメント

この関数は、オブジェクトへのポインタによって指定された特定の永続オブジェクトへの参照を戻します。(オブジェクトではなく) 値をこの関数に渡すと、エラーになります。

関連項目： オブジェクトのメタ属性の詳細は、10-16 ページの [「オブジェクトのメタ属性」](#) を参照してください。

関連関数

[OCIObjectPin\(\)](#)、[OCIObjectPin\(\)](#)

OCIObjectGetTypeRef()

用途

スタンドアロン・インスタンスの型記述子オブジェクト（TDO）への参照を戻します。

構文

```
sword OCIObjectGetTypeRef ( OCIEnv      *env,  
                           OCIError    *err,  
                           dvoid        *instance,  
                           OCIRef       *type_ref );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。詳細は、[「OCIEnvCreate\(\)」](#) および [「OCIInitialize\(\)」](#) の説明を参照してください。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、*err* に記録され、OCI_ERROR が戻されます。OCIErrorGet() をコールして診断情報を取得します。

instance (IN)

スタンドアロン・インスタンスへのポインタです。これはスタンドアロン型であることが必要で、インスタンスがオブジェクトの場合は、事前に確保されていることも必要です。

type_ref (OUT)

オブジェクトの型に対する参照です。参照は事前に割り当てられる必要があります。これは、OCIObjectNew() によって実行できます。

コメント

なし

関連関数

[OCIObjectPin\(\)](#)、[OCIObjectPin\(\)](#)

OCIObjectLock()

用途

永続オブジェクトをサーバー側でロックします。

構文

```
sword OCIObjectLock ( OCIEnv      *env,  
                      OCIError    *err,  
                      dvoid        *object );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。詳細は、[「OCIEnvCreate\(\)」](#) および [「OCIInitialize\(\)」](#) の説明を参照してください。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet() をコールして診断情報を取得します。

object (IN)

ロックされている永続オブジェクトへのポインタです。事前に確保されることが必要です。

コメント

この関数は、一時オブジェクトと値ではエラーを戻します。また、オブジェクトが存在しない場合にもエラーを戻します。

関連項目： オブジェクトのロックの詳細は、13-13 ページの [「更新するためのオブジェクトのロック」](#) を参照してください。

関連関数

[OCIObjectPin\(\)](#)、[OCIObjectIsLocked\(\)](#)、[OCIObjectGetProperty\(\)](#)、[OCIObjectLockNoWait\(\)](#)

OCIObjectLockNoWait()

用途

永続オブジェクトをサーバー側でロックしますが、ロック待機は行いません。ロックが使用不可能である場合は、エラーを戻します。

構文

```
sword OCIObjectLockNoWait ( OCIEnv      *env,  
                             OCIError    *err,  
                             dvoid        *object );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。詳細は、[「OCIEnvCreate\(\)」](#) および [「OCIInitialize\(\)」](#) の説明を参照してください。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、err に記録され、OCI_ERROR が戻されます。OCIErrorGet() をコールして診断情報を取得します。

object (IN)

ロックされている永続オブジェクトへのポインタです。事前に確保されることが必要です。

コメント

この関数は、永続オブジェクトをサーバー側でロックします。ただし、OCIObjectLock() とは異なり、この関数では、目的のオブジェクトに対するロックを別のユーザーが保持している場合に待機が行われません。オブジェクトが現在別のユーザーによってロックされている場合は、エラーが戻されます。また、この関数は、一時オブジェクトおよび値、または存在していないオブジェクトに対してエラーを戻します。

オブジェクトのロックは、トランザクションの終了時に解除されます。

関連項目： オブジェクトのロックの詳細は、13-13 ページの [「更新するためのオブジェクトのロック」](#) を参照してください。

OCIObjectLockNoWait() は、次の値を戻します。

- OCI_INVALID_HANDLE (環境ハンドルまたはエラー・ハンドルが NULL である場合)
- OCI_SUCCESS (操作が成功した場合)
- OCI_ERROR (操作が失敗した場合)

関連関数

[OCIObjectPin\(\)](#)、[OCIObjectIsLocked\(\)](#)、[OCIObjectGetProperty\(\)](#)、[OCIObjectLock\(\)](#)

OCIObjectNew()

用途

スタンドアロン・インスタンスを作成します。

構文

```
sword OCIObjectNew ( OCIEnv          *env,
                    OCLError         *err,
                    CONST OCISvcCtx  *svc,
                    OCITypeCode      typecode,
                    OCIType          *tdo,
                    dvoid             *table,
                    OCIDuration      duration,
                    boolean           value,
                    dvoid             **instance );
```

パラメータ

env (IN/OUT)

Unicode 設定を使用してオブジェクト・モードで初期化された OCI 環境ハンドルです。詳細は、「[OCIEnvCreate\(\)](#)」および「[OCIInitialize\(\)](#)」の説明を参照してください。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、err に記録され、OCI_ERROR が戻されます。OCIErrorGet() をコールして診断情報を取得します。

svc (IN) [オプション]

OCI サービス・ハンドルです。プログラムでインスタンスの継続時間と OCI サービスを関連付ける（たとえば、トランザクションのコミット時に文字列を解放する）場合は、これを必ず指定します。このパラメータは、TDO が指定された場合は無視されます。

typecode (IN)

インスタンスの型のコードです。

関連項目： 3-29 ページ [「型コード」](#)

tdo (IN) [オプション]

記述子オブジェクト型へのポインタです。TDO は、作成されるインスタンスの型を記述します。TDO の取得の詳細は、「[OCITypeByName\(\)](#)」を参照してください。TDO は、オブジェクトやコレクションなどの名前付き型の作成に必要です。

table (IN) [オプション]

表をサーバーに指定している表オブジェクトへのポインタです。表が指定されていない場合、このパラメータは NULL に設定できます。作成するインスタンスの種類（永続、一時、値）を判断するための表オブジェクトと TDO の使用方法は、次の説明を参照してください。表オブジェクトの取出しについては、「OCIObjectPinTable ()」も参照してください。

duration (IN)

これはオーバーロードされたパラメータです。このパラメータの使用方法は、作成されるインスタンスの種類によって決まります。

- 永続オブジェクト。このパラメータは、確保継続時間を指定します。
- 一時オブジェクト。このパラメータは、割当て時間および確保継続時間を指定します。
- 値。このパラメータは、割当て時間を指定します。

value (IN)

作成されたオブジェクトが値かどうかを指定します。TRUE の場合は、値が作成されます。それ以外の場合は、参照可能オブジェクトが作成されます。インスタンスがオブジェクトではない場合、このパラメータは無視されます。

instance (OUT)

新規に作成されたインスタンスのアドレスです。環境ハンドルが適切に設定され、オブジェクトが OCIStrng の場合、インスタンスは Unicode の文字列であることが可能です。この場合、インスタンスには、Unicode 設定であることを示すフラグが付きます。

コメント

この関数は、型コードまたは TDO によって指定された型のインスタンスを新たに作成します。型コードによって、作成するオブジェクトが OCIStrng であることが示されている場合は、Unicode バッファを使用して OCIStrng オブジェクトを作成できます。

関連項目： 3-29 ページ [「型コード」](#)

パラメータ typecode（または tdo）、value および table に基づいて、次のような異なる種類のインスタンスが作成されます。

作成される型	NULL 以外	NULL
オブジェクト型（value = TRUE）	値	値
オブジェクト型（value = FALSE）	永続オブジェクト	一時オブジェクト
組込み型	値	値
コレクション型	値	値

この関数は、インスタンスのトップレベル・メモリー・チャンクを割り当てます。トップレベル・メモリー内の属性が初期化されます。つまり、**VARCHAR2** の属性が長さ 0（ゼロ）の **OCIString** に初期化されます。インスタンスがオブジェクトの場合、そのオブジェクトに存在マークが設定されますがオブジェクトはアトミック NULL になります。

関連項目： オブジェクト・ビューまたはユーザー作成 OID に基づいた新規オブジェクト作成の詳細は、10-35 ページの「[オブジェクト・ビューまたはユーザー定義 OID に基づいたオブジェクトの作成](#)」を参照してください。

永続オブジェクト

オブジェクトに使用済みおよび存在マークが設定されます。オブジェクトの割当て時間はセッションです。オブジェクトは、*duration* パラメータに指定された確保継続時間の間、確保されます。永続オブジェクトが作成されても、そのオブジェクトがサーバーにフラッシュされるまで、データベース表へのエントリは行われません。

一時オブジェクト

オブジェクトが確保されます。割当て時間および確保継続時間は、*duration* パラメータによって指定されます。

値

割当て時間は、*duration* パラメータによって指定されます。

新規オブジェクトの属性値

デフォルトでは、新規作成されたオブジェクトのすべての属性に NULL 値があります。属性データを初期化した後は、対応する各属性の NULL ステータスを NULL 以外に変更する必要があります。

オブジェクトの作成時に、属性を NULL 以外の値に設定できます。これは、`OCIAttrSet()` を使用して環境ハンドルの `OCI_ATTR_OBJECT_NEWNOTNULL` 属性を TRUE に設定することによって実行できます。その後、この属性を FALSE に設定すると、このモードを無効にできます。`OCI_ATTR_OBJECT_NEWNOTNULL` が TRUE に設定されている場合は、`OCIObjectNew()` によって NULL 以外のオブジェクトが作成されます。

関連項目： 10-33 ページ「[新しいオブジェクトの属性値](#)」

LOB 属性を持つオブジェクト

オブジェクトに内部 LOB 属性がある場合、LOB は Empty 値に設定されます。LOB へのデータ書込みを開始する前には、そのオブジェクトに使用済みマークを設定して（オブジェクトを表に挿入するために）フラッシュし、再確保する必要があります。作成後にオブジェクトを確保するときは、新しく更新された LOB ロケータをサーバーから取り出すために、`OCI_PIN_LATEST PIN` オプションを使用する必要があります。

オブジェクトに外部 LOB 属性（FILE）がある場合は、FILE ロケータが割り当てられますが、初期化はされません。オブジェクトをデータベースにフラッシュする前に、`OCILobFileSetName()` をコールして FILE 属性を初期化する必要があります。最初にディ

レクトリ別名およびファイル名を指定せずに **FILE** の **INSERT** または **UPDATE** を実行すると、エラーが発生します。ファイル名を設定すると、**FILE** の読み込みを開始できます。

注意： Oracle では、現在、バイナリ・ファイル（**BFILE**）のみをサポートしています。

関連関数

`OCIObjectPinTable()`、`OCIObjectFree()`

OCIObjectSetAttr()

用途

オブジェクト属性を設定します。

構文

```
sword OCIObjectSetAttr ( OCIEnv          *env,
                        OCIError        *err,
                        dvoid           *instance,
                        dvoid           *null_struct,
                        struct OCIType   *tdo,
                        CONST OraText   **names,
                        CONST ub4       *lengths,
                        CONST ub4       name_count,
                        CONST ub4       *indexes,
                        CONST ub4       index_count,
                        CONST OCIIInd    null_status,
                        CONST dvoid      *attr_null_struct,
                        CONST dvoid      *attr_value );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。詳細は、「[OCIEnvCreate\(\)](#)」および「[OCIInitialize\(\)](#)」の説明を参照してください。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet() をコールして診断情報を取得します。

instance (IN)

オブジェクト・インスタンスへのポインタです。

null_struct (IN)

オブジェクト・インスタンスまたは配列の NULL インジケータ構造体です。

tdo (IN)

TDO へのポインタです。

names (IN)

属性名の配列です。パス式に属性名を指定します。

lengths (IN)

属性名長（バイト単位）の配列です。

name_count (IN)

配列 *names* の要素数です。

indexes (IN) [オプション]

現行ではサポートされていません。(ub4 *)0 で渡します。

index_count (IN) [オプション]

現行ではサポートされていません。(ub4)0 で渡します。

attr_null_status (IN)

属性タイプが基本形の場合、属性は NULL ステータスです。

attr_null_struct (IN)

オブジェクトまたはコレクション属性の NULL インジケータ構造体です。

attr_value (IN)

属性値へのポインタです。

コメント

この関数は、所定のオブジェクトの属性に所定の値を設定します。属性の位置は名前配列および索引配列であるパス式で指定されます。

例

パス式 `stanford.cs.stu[5].addr` の場合、配列は次のようになります。

```
names = {"stanford", "cs", "stu", "addr"}
```

```
lengths = {8, 2, 3, 4}
```

```
indexes = {5}
```

関連関数

[OCIObjectGetAttr\(\)](#)

OCI の確保関数、確保解除関数および解放関数

この項では、OCI の確保関数、確保解除関数および解放関数について説明します。

表 17-6 確保関数、確保解除関数および解放関数

関数	用途
OCICacheFree() (17-50 ページ)	キャッシュ内のオブジェクトを解放します。
OCICacheUnpin() (17-51 ページ)	キャッシュまたは接続で永続オブジェクトの確保を解除します。
OCIObjectArrayPin() (17-52 ページ)	参照の配列を確保します。
OCIObjectFree() (17-54 ページ)	割当て済みのオブジェクトを解放します。
OCIObjectPin() (17-56 ページ)	オブジェクトを確保します。
OCIObjectPinCountReset() (17-59 ページ)	オブジェクトの確保を解除して確保カウントを 0 (ゼロ) にします。
OCIObjectPinTable() (17-60 ページ)	継続時間を指定して表オブジェクトを確保します。
OCIObjectUnpin() (17-62 ページ)	オブジェクトの確保を解除します。

OCICacheFree()

用途

指定された接続に対してキャッシュ内のオブジェクトと値をすべて解放します。

構文

```
sword OCICacheFree ( OCIEnv          *env,  
                     OCIError       *err,  
                     CONST OCISvcCtx *svc );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。詳細は、[「OCIEnvCreate\(\)」](#) および [「OCIInitialize\(\)」](#) の説明を参照してください。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。`OCIErrorGet()` をコールして診断情報を取得します。

svc (IN)

OCI サービス・コンテキストです。

コメント

接続が指定された場合、この関数は、その接続に対して割り当てられている永続オブジェクト、一時オブジェクトおよび値を解放します。それ以外の場合は、オブジェクト・キャッシュ内の永続オブジェクト、一時オブジェクトおよび値をすべて解放します。オブジェクトは、確保カウントに関係なく解放されます。

関連項目： インスタンスの解放の詳細は、17-54 ページの [「OCIObjectFree\(\)」](#) を参照してください。

関連関数

[OCIObjectFree\(\)](#)

OCICacheUnpin()

用途

永続オブジェクトの確保を解除します。

構文

```
sword OCICacheUnpin ( OCIEnv          *env,  
                      OCIError        *err,  
                      CONST OCISvcCtx *svc );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。詳細は、[「OCIEnvCreate\(\)」](#) および [「OCIInitialize\(\)」](#) の説明を参照してください。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、err に記録され、OCI_ERROR が戻されます。OCIErrorGet() をコールして診断情報を取得します。

svc (IN)

OCI サービス・コンテキスト・ハンドルです。指定された接続のオブジェクトの確保が解除されます。

コメント

この関数は、所定の接続に対するすべての永続オブジェクトの確保を完全に解除します。オブジェクトの確保カウントは 0 (ゼロ) にリセットされます。

関連項目： 確保および確保解除の詳細は、10-11 ページの「[オブジェクトの確保](#)」および 10-29 ページの「[確保カウントおよび確保解除](#)」を参照してください。

関連関数

[OCIObjectUnpin\(\)](#)

OCIObjectArrayPin()

用途

参照配列を確保します。

構文

```
sword OCIObjectArrayPin ( OCIEnv          *env,
                          OCIError        *err,
                          OCIRef          **ref_array,
                          ub4             array_size,
                          OCIComplexObject **cor_array,
                          ub4             cor_array_size,
                          OCIPinOpt       pin_option,
                          OCIDuration     pin_duration,
                          OCILockOpt      lock,
                          dvoid           **obj_array,
                          ub4             *pos );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。詳細は、[「OCIEnvCreate\(\)」](#) および [「OCIInitialize\(\)」](#) の説明を参照してください。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet() をコールして診断情報を取得します。

ref_array (IN)

確保される参照の配列です。

array_size (IN)

参照の配列にある要素の数です。

cor_array

確保されるオブジェクトに対応する COR ハンドルの配列です。

cor_array_size

`cor_array` にある要素の数です。

pin_option (IN)

PIN オプションです。

関連項目： 17-56 ページ [「OCIObjectPin\(\)」](#)

pin_duration (IN)

確保継続時間です。「OCIObjectPin()」を参照してください。

lock (IN)

ロック・オプションです。「OCIObjectPin()」を参照してください。

obj_array (OUT)

この引数が NULL でない場合は、確保されたオブジェクトをこの配列に戻します。**dvoid ***型の要素をこの配列に割り当てる必要があります。この配列のサイズは *array_size* と同じです。

pos (OUT)

エラーが発生すると、この引数によってエラーの原因となった要素が示されます。この引数には、*ref_array* の第 1 要素に対して、1 が設定されます。

コメント

確保されたすべてのオブジェクトが 1 回のネットワーク・ラウンドトリップでデータベースから取り出されます。ユーザーが出力配列 (*obj_array*) を指定した場合は、確保されたオブジェクトのアドレスが配列内の各要素に割り当てられます。

関連関数

[OCIObjectPin\(\)](#)

OCIObjectFree()

用途

オブジェクト・インスタンスの解放と確保解除を行います。

構文

```
sword OCIObjectFree ( OCIEnv      *env,
                      OCIError    *err,
                      dvoid       *instance,
                      ub2         flags );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。詳細は、「[OCIEnvCreate\(\)](#)」および「[OCIInitialize\(\)](#)」の説明を参照してください。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。`OCIErrorGet()` をコールして診断情報を取得します。

instance (IN)

スタンドアロン・インスタンスへのポインタです。オブジェクトの場合は、確保されることが必要です。

flags (IN)

`OCI_OBJECTFREE_FORCE` が渡された場合は、オブジェクトが確保済みまたは使用済みである場合も解放します。`OCI_OBJECTFREE_NONULL` が渡されると、`NULL` インジケータ構造体は解放されません。

コメント

この関数は、オブジェクト・インスタンスのために割り当てられたすべてのメモリの割当てを解除します (`NULL` インジケータ構造体も含まれます)。インスタンスの型に応じて、次の規則が適用されます。

永続オブジェクト

この関数は、フラッシュされていない使用済み永続オブジェクトをクライアントが解放しようとした場合、エラーを戻します。クライアントは、永続オブジェクトをフラッシュするか、マークを解除するか、または `flags` パラメータに `OCI_OBJECTFREE_FORCE` を設定する必要があります。

この関数は、オブジェクトの確保を完全に解除できるかどうかを調べるために `OCIObjectUnpin()` を 1 回コールします。成功した場合は、関数の残りを続行してオブジェクトを解放します。失敗した場合は、`flags` パラメータに `OCI_OBJECTFREE_FORCE` が設定されていないかぎり、エラーが戻ります。

メモリー内の永続オブジェクトを解放しても、そのオブジェクトのサーバー側での永続状態が変化することはありません。たとえば、オブジェクトが解放された後でも、そのオブジェクトはロックされたままになります。

一時オブジェクト

この関数は、オブジェクトの確保を完全に解除できるかどうかを調べるために `OCIObjectUnpin()` を 1 回コールします。成功した場合は、関数の残りを続行しオブジェクトを解放します。失敗した場合は、`flags` パラメータに `OCI_OBJECTFREE_FORCE` が設定されていないかぎり、エラーが戻ります。

値

オブジェクトのメモリーが即時に解放されます。

関連関数

`OCICacheFree()`

OCIObjectPin()

用途

参照可能オブジェクトを確保します。

構文

```
sword OCIObjectPin ( OCIEnv          *env,
                    OCIError         *err,
                    OCIRef            *object_ref,
                    OCIComplexObject *corhdl,
                    OCIPinOpt        pin_option,
                    OCIDuration       pin_duration,
                    OCILockOpt        lock_option,
                    dvoid              **object );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。詳細は、「[OCIEnvCreate\(\)](#)」および「[OCIInitialize\(\)](#)」の説明を参照してください。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。`OCIErrorGet()` をコールして診断情報を取得します。

object_ref (IN)

オブジェクトに対する参照です。

corhdl (IN)

複合オブジェクト検索に対するハンドルです。

pin_option (IN)

取り出されるオブジェクトのコピーの指定に使用します。

pin_duration (IN)

オブジェクトがクライアントによってアクセスされる継続時間です。オブジェクトの確保は、確保継続時間の終了時点で暗黙的に解除されます。`OCI_DURATION_NULL` が渡された場合、オブジェクトがすでにキャッシュ内にロードされているときは、確保の促進はありません。オブジェクトが未ロードで `OCI_DURATION_NULL` の場合、確保継続時間には `OCI_DURATION_DEFAULT` が設定されます。

lock_option (IN)

ロック・オプション（たとえば、排他的ロック）です。ロック・オプションが指定された場合、オブジェクトはサーバー内でロックされます。オブジェクトのロック状況は、OCIObjectIsLocked() 関数をコールしても取り出せます。次の値が有効です。

- OCI_LOCK_NONE – ロックなし
- OCI_LOCK_X – 排他ロック
- OCI_LOCK_X_NOWAIT – NOWAIT オプションを使用した排他ロック

関連項目： NOWAIT オプションの詳細は、13-14 ページの「[NOWAIT オプションを使用したロック](#)」を参照してください。

object (OUT)

確保されたオブジェクトへのポインタです。

コメント

この関数は、オブジェクト参照によって指示された参照可能オブジェクト・インスタンスを確保します。この確保のプロセスで、次の 2 つの目的を達成します。

- 参照によって指示されたオブジェクトの場所を特定します。これは、オブジェクト・キャッシュ内のオブジェクトを追跡し記録するオブジェクト・キャッシュによって実行されます。
- 永続オブジェクトは使用中のため、エージ・アウトが不可であることをオブジェクト・キャッシュに通知します。永続オブジェクトは必要に応じていつでもサーバーからロードできるため、割当て時間の経過前であっても、確保が完全に解除された永続オブジェクトを解放する（エージ・アウトさせる）ことができる場合は、利用できるメモリーを増やすことができます。オブジェクトは複数回確保できます。確保されたオブジェクトは、確保が完全に解除されるまでメモリー内に残ります。

関連項目： 17-62 ページの「[OCIObjectUnpin\(\)](#)」を参照してください。

永続オブジェクト

永続オブジェクトを確保したときにそのオブジェクトがキャッシュ内に存在しない場合は、永続ストアからフェッチされます。オブジェクトの割当て時間はセッションです。オブジェクトがキャッシュ内に存在する場合は、そのオブジェクトがクライアントに戻されます。ロック・オプションが指定されている場合、オブジェクトは、サーバー内でロックされます。

この関数は、存在しないオブジェクトに対してはエラーを戻します。

確保オプションを使用して、取り出されるオブジェクトのコピーを指定します。

- *pin_option* が OCI_PIN_ANY（任意のオブジェクトを確保）であり、オブジェクトがすでにオブジェクト・キャッシュ内にある場合は、そのオブジェクトが戻されます。それ以外の場合、オブジェクトはデータベースから取り出されます。これは、

pin_option が `OCI_PIN_LATEST` である場合と同じです。このオプションは、クライアントがセッションのデータに排他的アクセス権を持っていることを認識しているときに役立ちます。

- *pin_option* が `OCI_PIN_LATEST`（最新のオブジェクトを確保）の場合は、オブジェクトがロックされていなければデータベースから取り出されます。オブジェクトがキャッシュ処理されている場合は、最新バージョンによってリフレッシュされます。リフレッシュの詳細は、「`OCIObjectRefresh()`」を参照してください。オブジェクトがすでにキャッシュ内に確保され、使用済みマークが設定されている場合は、そのオブジェクトへのポインタが戻ります。オブジェクトは、データベースからリフレッシュされません。
- *pin_option* が `OCI_PIN_RECENT`（最近のオブジェクトを確保）であり、オプションがカレント・トランザクションでキャッシュにロードされた場合は、そのオブジェクトが戻されます。オブジェクトがカレント・トランザクション中にロードされていない場合、オブジェクトはサーバーからリフレッシュされます。

一時オブジェクト

一時オブジェクトがすでに解放されている場合、この関数はエラーを戻します。ロック・オプションに排他ロックが指定されている場合は、エラーを戻しません。

関連関数

`OCIObjectUnpin()`、`OCIObjectPinCountReset()`

OCIObjectPinCountReset()

用途

オブジェクトの確保を完全に解除し、その確保カウントを 0（ゼロ）に設定します。

構文

```
sword OCIObjectPinCountReset ( OCIEnv      *env,  
                               OCIError    *err,  
                               dvoid        *object );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。詳細は、「[OCIEnvCreate\(\)](#)」および「[OCIInitialize\(\)](#)」の説明を参照してください。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、err に記録され、OCI_ERROR が戻されます。OCIErrorGet () をコールして診断情報を取得します。

object (IN)

オブジェクトへのポインタで、すでに確保されていることが必要です。

コメント

この関数は、オブジェクトの確保を完全に解除し、その確保カウントに 0（ゼロ）を設定します。オブジェクトの確保が完全に解除されると、OCI は、そのオブジェクトをエラーなしにいつでも暗黙的に解放します。オブジェクトの型に応じて、次の規則が適用されます。

永続オブジェクト

永続オブジェクトの確保が完全に解除された場合、そのオブジェクトはエージ・アウトの対象になります。オブジェクトのメモリーは、オブジェクトのエージ・アウトが終了したときに解放されます。エージ・アウトは、メモリーを最大限に利用するために行われます。使用済みオブジェクトは、フラッシュされるまでエージ・アウトされません。

一時オブジェクト

オブジェクトの確保カウントが減分されます。一時オブジェクトは、その割当て時間の終了時、または OCIObjectFree () をコールすることで明示的に解放された場合にかぎり解放できます。

値

この関数は、値に対してはエラーを戻します。

関連項目： 10-29 ページ「[確保カウントおよび確保解除](#)」

関連関数

[OCIObjectPin\(\)](#)、[OCIObjectUnpin\(\)](#)

OCIObjectPinTable()

用途

表オブジェクトを指定された継続時間の間、確保します。

構文

```
sword OCIObjectPinTable ( OCIEnv          *env,
                          OCIError        *err,
                          CONST OCISvcCtx *svc,
                          CONST OraText   *schema_name,
                          ub4             s_n_length,
                          CONST OraText   *object_name,
                          ub4             o_n_length,
                          dvoid           *not_used,
                          OCIDuration     pin_duration,
                          dvoid           **object );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。詳細は、「[OCIEnvCreate\(\)](#)」および「[OCIInitialize\(\)](#)」の説明を参照してください。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet () をコールして診断情報を取得します。

svc (IN)

OCI サービス・コンテキスト・ハンドルです。

schema_name (IN) [オプション]

表のスキーマ名です。

s_n_length (IN) [オプション]

`schema_name` に示されているスキーマ名の長さ（バイト単位）です。

object_name (IN)

表の名前です。

o_n_length (IN)

`object_name` に示されている表名の長さ（バイト単位）です。

not_used (IN/OUT)

このパラメータは使用されていません。NULL を渡します。

pin_duration (IN)

確保継続時間です。

関連項目： 17-56 ページの「[OCIObjectPin\(\)](#)」の説明を参照してください。

object (OUT)

確保された表オブジェクトです。

コメント

この関数は、表オブジェクトを指定の確保継続時間の間、確保します。クライアントは、[OCIObjectUnpin\(\)](#) をコールしてこのオブジェクトの確保を解除できます。

このコールによって確保される表オブジェクトは、スタンドアロン型の永続オブジェクトを作成するために、パラメータとして [OCIObjectNew\(\)](#) に渡すことができます。

注意： サービス・コンテキストは、TDO または表定義が使用する論理パーティションに対応している必要があります。TDO または表定義が複数の論理パーティションで使用されている場合、リリースによって動作が異なる可能性があります。

関連関数

[OCIObjectPin\(\)](#)、[OCIObjectUnpin\(\)](#)

OCIObjectUnpin()

用途

オブジェクトの確保を解除します。

構文

```
sword OCIObjectUnpin ( OCIEnv          *env,  
                      OCIError        *err,  
                      dvoid           *object );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。詳細は、「[OCIEnvCreate\(\)](#)」および「[OCIInitialize\(\)](#)」の説明を参照してください。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet() をコールして診断情報を取得します。

object (IN)

オブジェクトへのポインタで、すでに確保されている必要があります。

コメント

各オブジェクトには、そのオブジェクトが確保されるたびに増分される確保カウントが対応付けられています。オブジェクトの確保カウントが 0（ゼロ）になったとき、そのオブジェクトの確保は完全に解除されたことになります。確保が解除されたオブジェクトは、OCI によって、エラーなしにいつでも暗黙的に解放されます。

この関数は、オブジェクトの確保を解除します。次のいずれかが TRUE であるとき、オブジェクトの確保は完全に解除されています。

1. オブジェクトの確保カウントが 0（ゼロ）に達しています（つまり、全部で N 回確保された後に全部で N 回確保解除されています）。
2. オブジェクトの確保継続時間が終了しています。
3. オブジェクトに対して OCIObjectPinCountReset() 関数がコールされました。

オブジェクトの確保が完全に解除されると、OCI は、そのオブジェクトをエラーなしにいつでも暗黙的に解放します。

オブジェクトの型に応じて、次の規則が適用されます。

永続オブジェクト

永続オブジェクトの確保が完全に解除された場合、そのオブジェクトはエージ・アウトの対象になります。オブジェクトのメモリーは、オブジェクトのエージ・アウトが終了したときに解放されます。エージ・アウトは、メモリーを最大限に利用するために行われます。使用済みオブジェクトは、フラッシュされるまでエージ・アウトされません。

一時オブジェクト

オブジェクトの確保カウントが減分されます。一時オブジェクトは、その割当て時間の終了時、または `OCIObjectFree()` をコールすることで明示的に削除したときにかぎり解放されます。

値

この関数は値に対してはエラーを戻します。

関連関数

`OCIObjectPin()`、`OCIObjectPinCountReset()`

OCI の型情報アクセッサ関数

この項では、OCI の型情報アクセッサ関数について説明します。

表 17-7 型情報アクセッサ関数

関数	用途
OCITypeArrayByName () (17-65 ページ)	オブジェクト名配列を指定して TDO 配列を取得します。
OCITypeArrayByRef () (17-68 ページ)	オブジェクト参照配列を指定して TDO 配列を取得します。
OCITypeByName () (17-70 ページ)	オブジェクト名を指定して TDO を取得します。
OCITypeByRef () (17-73 ページ)	オブジェクト参照を指定して TDO を取得します。

OCITypeArrayByName()

用途

名前配列を指定して型配列を取得します。

構文

```
sword OCITypeArrayByName ( OCIEnv           *envhp,
                           OCIError         *errhp,
                           CONST OCISvcCtx  *svc,
                           ub4              array_len,
                           CONST text       *schema_name[],
                           ub4              s_length[],
                           CONST text       *type_name[],
                           ub4              t_length[],
                           CONST text       *version_name[],
                           ub4              v_length[],
                           OCIDuration      pin_duration,
                           OCITypeGetOpt    get_option,
                           OCIType         *tdo[] );
```

パラメータ

envhp (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。詳細は、「[OCIEnvCreate\(\)](#)」および「[OCIInitialize\(\)](#)」の説明を参照してください。

errhp (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet() をコールして診断情報を取得します。

svc (IN)

OCI サービス・ハンドルです。

array_len (IN)

取り出される `schema_name/type_name/version_name` エントリの数です。

schema_name (IN、オプション)

取り出される型に対応付けられているスキーマ名の配列です。この配列を指定する場合は、`array_len` 個の要素が必要です。0 (ゼロ) を指定した場合は、デフォルト・スキーマであるとみなされます。それ以外の場合は、`array_len` 個の要素が必要です。デフォルト・スキーマが必要であることを指示するために、1 つ以上のエントリに 0 (ゼロ) を指定できます。

s_length (IN)

schema_name 長さの配列で、それぞれのエントリが、*schema_name* 配列の対応する *schema_name* エントリのバイト単位の長さに対応しています。この配列には、*array_len* 個の要素が必要です。*schema_name* が指定されていない場合は、0（ゼロ）にしてください。

type_name (IN)

取り出される型の名前の配列です。*array_len* 個の要素が必要です。

t_length (IN)

type_name 配列にある型名のバイト単位の長さの配列です。

version_name (IN)

バージョン名は無視され、要求された型の最新バージョンが戻されます。リリース 9.0.1 以上は型の変更が可能であるため、それより前のリリースのアプリケーションで変更された型にアクセスすると、エラーが発生します。この場合は、新しい型定義を使用して、アプリケーションの変更、再コンパイルおよび再リンクを行う必要があります。

取り出される型に対応するバージョン名の配列です。この配列は、最新バージョンの取出しを示す 0（ゼロ）にするか、または *array_len* 個の要素が必要です。

0（ゼロ）が指定された場合は、最新バージョンとみなされます。それ以外の場合は、*array_len* 個の要素が必要です。現行バージョンが必要であることを示すために、複数のエントリに 0（ゼロ）を指定できます。

v_length (IN)

version_name 配列にあるバージョン名のバイト単位の長さの配列です。

pin_duration (IN)

取り出された型用の確保継続時間（たとえば、カレント・トランザクションの終わりまで）です。各オプションの説明は、*oro.h* を参照してください。

get_option (IN)

型をロードするためのオプションは、次のどちらかの値です。

- OCI_TYPEGET_HEADER — ロードされるヘッダー用のみ
- OCI_TYPEGET_ALL — TDO およびロードされるすべての ADO と MDO 用

tdo (OUT)

オブジェクト・キャッシュに確保された各型へのポインタ用出力配列です。これには、*array_len* ポインタ用の領域が必要です。OCIObjectGetObjectRef() を使用して、確保された各型記述子に対する CREF を取得します。

コメント

スキーマ / 型名配列に対応付けられている既存の型のポインタを取得します。

`get_option` パラメータを使用すると、各ラウンドトリップごとにロードされる TDO の部分を制御できます。

この関数は、必須パラメータに `NULL` がある場合、あるいはスキーマ名または型名に対応付けられたオブジェクト型が存在しない場合はエラーを戻します。

配列ではなく単一の型を取り出すには、`OCITypeByName()` を使用します。

注意： サービス・コンテキストは、TDO または表定義が使用する論理パーティションに対応している必要があります。TDO または表定義が複数の論理パーティションで使用されている場合、リリースによって動作が異なる可能性があります。

関連関数

[OCITypeArrayByRef\(\)](#)、[OCITypeByName\(\)](#)、[OCITypeByRef\(\)](#)

OCITypeArrayByRef()

用途

参照配列を指定して型配列を取得します。

構文

```
sword OCITypeArrayByRef ( OCIEnv          *envhp,
                          OCIError        *errhp,
                          ub4              array_len,
                          CONST OCISRef   *type_ref[],
                          OCIDuration      pin_duration,
                          OCITypeGetOpt    get_option,
                          OCISRef          *tdo[] );
```

パラメータ

envhp (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。詳細は、「[OCIEnvCreate\(\)](#)」および「[OCIInitialize\(\)](#)」の説明を参照してください。

errhp (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet() をコールして診断情報を取得します。

array_len (IN)

取り出される `schema_name/type_name/version_name` エントリの数です。

type_ref (IN)

取り出される型記述子オブジェクトの特定のバージョンを指し示す **OCISRef*** の配列です。この配列を指定する場合は、`array_len` 個の要素が必要です。

pin_duration (IN)

取り出された型用の確保継続時間（たとえば、カレント・トランザクションの終わりまで）です。各オプションの説明は、`oro.h` を参照してください。

get_option (IN)

型をロードするためのオプションは、次のどちらかの値です。

- `OCI_TYPEGET_HEADER` — ロードされるヘッダーのみ
- `OCI_TYPEGET_ALL` — TDO およびロードされるすべての ADO と MDO 用

tdo (OUT)

オブジェクト・キャッシュに確保された各型へのポインタ用出力配列です。これには、`array_len` ポインタ用の領域が必要です。OCIObjectGetObjectRef() を使用して、確保された各型記述子に対する CREF を取得します。

コメント

スキーマ / 型名配列に対応付けられている既存の型のポインタを取得します。

この関数は、次の場合にエラーを戻します。

- 必須パラメータに NULL がある場合
- スキーマ / 型名エントリに対応付けられている 1 つ以上のオブジェクト型が存在しない場合

型の配列ではなく、単一の型を取り出すには、`OCITypeByRef()` を使用します。

関連関数

`OCITypeArrayByName()`、`OCITypeByRef()`、`OCITypeByName()`

OCITypeByName()

用途

既存の型の最新バージョンを名前を指定して取得します。

構文

```
sword OCITypeByName ( OCIEnv          *env,
                      OCIError        *err,
                      CONST OCISvcCtx *svc,
                      CONST text      *schema_name,
                      ub4             s_length,
                      CONST text      *type_name,
                      ub4             t_length,
                      CONST text      *version_name,
                      ub4             v_length,
                      OCIDuration     pin_duration,
                      OCITypeGetOpt   get_option,
                      OCIType         **tdo );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。詳細は、「[OCIEnvCreate\(\)](#)」および「[OCIInitialize\(\)](#)」の説明を参照してください。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。`OCIErrorGet()` をコールして診断情報を取得します。

svc (IN)

OCI サービス・ハンドルです。

schema_name (IN、オプション)

型に対応付けられているスキーマ名です。デフォルトでは、ユーザーのスキーマ名が使用されます。この文字列はすべて大文字であることが必要です。それ以外の場合は、内部エラーが発生してプログラムが停止します。

s_length (IN)

`schema_name` パラメータの長さ（バイト単位）です。

type_name (IN)

取得する型の名前です。この文字列はすべて大文字であることが必要です。それ以外の場合は、内部エラーが発生してプログラムが停止します。

t_length (IN)

type_name パラメータの長さ（バイト単位）です。

version_name (IN)

バージョン名は無視され、要求された型の最新バージョンが戻されます。リリース 9.0.1 以上は型の変更が可能であるため、それより前のリリースのアプリケーションで変更された型にアクセスすると、エラーが発生します。この場合は、新しい型定義を使用して、アプリケーションの変更、再コンパイルおよび再リンクを行う必要があります。

ユーザーが読取り可能な、型のバージョンです。最新のバージョンを取り出すには、(text *) 0 を渡します。

v_length (IN)

version_name のバイト単位の長さです。

pin_duration (IN)

確保継続時間です。

関連項目： 13-15 ページ「[オブジェクト継続時間](#)」

get_option (IN)

型をロードするためのオプションは、次のどちらかの値です。

- OCI_TYPEGET_HEADER – ロードされるヘッダー用のみ
- OCI_TYPEGET_ALL – TDO およびロードされるすべての ADO と MDO 用

tdo (OUT)

オブジェクト・キャッシュに確保された型へのポインタです。

コメント

この関数は、スキーマ / 型名に対応付けられている既存の型へのポインタを取得します。必須パラメータに NULL がある場合、スキーマ / 型名に対応付けられたオブジェクト型が存在しない場合、または *version_name* が存在しない場合はエラーを戻します。

注意： スキーマおよび型名は大 / 小文字が区別されます。SQL を使用して作成されている場合は、すべて大文字の文字列を使用する必要があります。すべて大文字でない場合、プログラムは停止します。

この関数では、常にサーバーへのラウンドトリップが発生するため、この関数を繰り返しコールして型を取得すると、パフォーマンスが大幅に低下する可能性があります。ラウンドトリップを最小限にするため、アプリケーションでは、型ごとにこの関数をコールし、型オブジェクトをキャッシュできます。

この関数によって取得した型を解放するには、`OCIObjectUnpin()` または `OCIObjectPinCountReset()` をコールします。

`OCITypeArrayByName()` または `OCITypeArrayByRef()` をコールすることにより、アプリケーションで TDO の配列を取り出せます。

注意： サービス・コンテキストは、TDO または表定義が使用する論理パーティションに対応している必要があります。TDO または表定義が複数の論理パーティションで使用されている場合、リリースによって動作が異なる可能性があります。

関連関数

`OCITypeByRef()`、`OCITypeArrayByName()`、`OCITypeArrayByRef()`

OCITypeByRef()

用途

参照を指定して型を取得します。

構文

```
sword OCITypeByRef ( OCIEnv          *env,  
                    OCIError        *err,  
                    CONST OCISRef    *type_ref,  
                    OCIDuration      pin_duration,  
                    OCITypeGetOpt    get_option,  
                    OCISRef          *tdo );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。詳細は、[「OCISCreate\(\)」](#) および [「OCIInitialize\(\)」](#) の説明を参照してください。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、err に記録され、OCI_ERROR が戻されます。OCIErrorGet() をコールして診断情報を取得します。

type_ref (IN)

取得する型記述子オブジェクトのバージョンを指し示す **OCISRef*** です。

pin_duration (IN)

取り出す型に対する、カレント・トランザクションの終了までの確保継続時間です。各オプションの説明は、oro.h を参照してください。

get_option (IN)

型をロードするためのオプションは、次のどちらかの値です。

- OCI_TYPEGET_HEADER – ロードされるヘッダー用のみ
- OCI_TYPEGET_ALL – TDO およびロードされるすべての ADO と MDO 用

tdo (OUT)

オブジェクト・キャッシュに確保された型へのポインタです。

コメント

OCITypeByRef() 関数は、必須パラメータに NULL がある場合、エラーを戻します。

関連関数

[OCISByName\(\)](#)、[OCISArrayByName\(\)](#)、[OCISArrayByRef\(\)](#)

OCI のデータ型マッピング関数および 操作関数

この章では、OCI のデータ型マッピング関数および操作関数について説明します。これらは Oracle 事前定義型に対する、Oracle の外部 C 言語インタフェースです。

関連項目： コード例は、Oracle のインストールに含まれているデモ・プログラムを参照してください。追加情報については、[付録 B「OCI デモ・プログラム」](#)を参照してください。

この章は、次の項目で構成されています。

- [データ型マッピング関数および操作関数の概要](#)
- [OCI コレクションおよびイテレータ関数](#)
- [OCI の日付関数、日時関数および時間隔関数](#)
- [OCI 数値関数](#)
- [OCI ロー関数](#)
- [OCI REF 関数](#)
- [OCI 文字列関数](#)
- [OCI 表関数](#)

データ型マッピング関数および操作関数の概要

この章では、OCI データ型マッピング関数および操作関数の詳細を説明します。

関連項目： この章にリストされている関数の詳細は、[第 11 章「オブジェクト・リレーショナル・データ型」](#)を参照してください。

関数の構文

関数ごとに次の情報が記載されています。

用途

関数の用途を簡単に説明します。

構文

関数の宣言。

コメント

この関数に関する詳細情報。関数の使用上の制約やアプリケーション内でこの関数を使用するときに役に立つ情報が記載されています。

パラメータ

この関数の各パラメータの説明。これにはパラメータのモードが含まれます。パラメータのモードには、次の 3 つの値があります。

モード	説明
IN	Oracle にデータを渡すパラメータ
OUT	このコールまたは後続のコールで Oracle からデータを受け取るパラメータ
IN/OUT	このコールでデータを渡し、このコールまたは後続のコールからの戻りでデータを受け取るパラメータ

戻り値

18-3 ページの[表 18-1「関数戻り値」](#)にリストされている標準リターン・コード以外の値が関数によって戻される場合の、戻り値についての説明。

関連関数

関連する関数のリスト

データ型マッピング関数および操作関数の戻り値

OCI のデータ型マッピング関数および操作関数は、通常、次の値の 1 つを返します。

表 18-1 関数戻り値

戻り値	意味
OCI_SUCCESS	操作は成功しました。
OCI_ERROR	操作は失敗しました。関数に渡されたエラー・ハンドルに対して <code>OCIErrorGet()</code> をコールすることで特定のエラーを取り出せます。
OCI_INVALID_HANDLE	関数に渡された OCI ハンドルが無効です。

この章では、各関数に関する記述の後で関数固有の戻り値に関する情報を説明します。

関連項目： リターン・コードおよびエラー処理の詳細は、2-31 ページの「[エラー処理](#)」を参照してください。

その他の値を返す関数

一部の関数は、表 18-1 にリストされていない値を返します。これらの関数を使用する場合は必ず、OUT パラメータからではなく、関数コールから直接値が返ることを考慮してください。

- `OCICollMax()`
- `OCIRawPtr()`
- `OCIRawSize()`
- `OCIRefHexSize()`
- `OCIRefIsEqual()`
- `OCIRefIsNull()`
- `OCIStringPtr()`
- `OCIStringSize()`

データ型マッピング関数および操作関数のサーバー・ラウンドトリップ回数

個々の OCI のデータ型マッピング関数および操作関数に必要なサーバー・ラウンドトリップ回数を示す表は、[付録 C「OCI 関数のサーバー・ラウンドトリップ」](#)を参照してください。

例

コード例など、これらの関数の詳細は、[第 11 章「オブジェクト・リレーショナル・データ型」](#)を参照してください。

OCI コレクションおよびイテレータ関数

この項では、コレクションおよびイテレータ関数について説明します。

表 18-2 コレクションおよびイテレータ関数

関数	用途
OCICollAppend() (18-6 ページ)	コレクションに要素を追加します。
OCICollAssign() (18-7 ページ)	コレクションを割り当てます。
OCICollAssignElem() (18-9 ページ)	コレクションに要素を割り当てます。
OCICollGetElem() (18-11 ページ)	要素へのポインタを取得します。
OCICollIsLocator() (18-14 ページ)	コレクションがロケータベースであるかどうかを示します。
OCICollMax() (18-15 ページ)	コレクションの最大要素数を戻します。
OCICollSize() (18-16 ページ)	コレクションのカレント・サイズ（要素数単位）を取得します。
OCICollTrim() (18-18 ページ)	コレクションから要素を切り捨てます。
OCIIterCreate() (18-19 ページ)	VARRAY 要素をスキャンするためのイテレータを作成します。
OCIIterDelete() (18-20 ページ)	イテレータを削除します。
OCIIterGetCurrent() (18-21 ページ)	カレント・コレクション要素を取得します。
OCIIterInit() (18-22 ページ)	イテレータを初期化して指定のコレクションをスキャンします。
OCIIterNext() (18-23 ページ)	次のコレクション要素を取得します。
OCIIterPrev() (18-25 ページ)	前のコレクション要素を取得します。

OCICollAppend()

用途

コレクションの最後に要素を追加します。

構文

```
sword OCICollAppend ( OCIEnv          *env,  
                      OCIError        *err,  
                      CONST dvoid      *elem,  
                      CONST dvoid      *elemind,  
                      OCIColl          *coll );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet() をコールして診断情報を取得します。

elem (IN)

与えられたコレクションの最後に追加される要素へのポインタです。

elemind (IN) [オプション]

要素の NULL インジケータ情報へのポインタです。`elemind == NULL` の場合、追加された要素の NULL インジケータ情報は、NULL 以外に設定されます。

coll (IN/OUT)

更新されたコレクションです。

コメント

要素の追加は、コレクションのサイズを 1 要素分増やし、最終要素のデータを指定の要素のデータで更新（ディープ・コピー）することと等価です。指定の要素 `elem` へのポインタはこの関数によって保存されないことに注意してください。つまり、`elem` は厳密に入力パラメータです。

この関数は、コレクションのカレント・サイズが、要素を追加する前のコレクションの最大サイズ（上限）と等しい場合、エラーを戻します。また、この関数は、入力パラメータに NULL がある場合はエラーを戻します。

関連関数

[OCIErrorGet\(\)](#)

OCICollAssign()

用途

あるコレクションを別のコレクションに割り当てます（ディープ・コピーします）。

構文

```
sword OCICollAssign ( OCIEnv          *env,  
                      OCIError        *err,  
                      CONST OCIColl    *rhs,  
                      OCIColl         *lhs );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet() をコールして診断情報を取得します。

rhs (IN)

割当て元となる右側（ソース）コレクションです。

lhs (OUT)

割当て先となる左側（ターゲット）コレクションです。

コメント

`rhs`（ソース）を `lhs`（ターゲット）に割り当てます。`lhs` コレクションは、`rhs` のサイズに応じて変更されます。`lhs` に要素が格納されている場合は、割当てに先立ってその要素が削除されます。この関数はディープ・コピーを実行します。要素用のメモリーはオブジェクト・キャッシュから取られます。

`lhs` コレクションと `rhs` コレクションの要素の型が一致しない場合はエラーが戻ります。`lhs` の上限が `rhs` コレクション内の現行の要素数より小さい場合もエラーが戻ります。次の場合にもエラーが戻されます。

- 入力パラメータに NULL がある場合
- `lhs` コレクションと `rhs` コレクションの型が一致しない場合
- `lhs` コレクションの上限が `rhs` コレクション内の現行の要素数より小さい場合

関連関数

[OCIErrorGet\(\)](#)、[OCICollAssignElem\(\)](#)

OCICollAssignElem()

用途

指定の要素値 *elem* を *coll[index]* の要素に割り当てます。

構文

```
sword OCICollAssignElem ( OCIEnv          *env,  
                          OCIError        *err,  
                          sb4              index,  
                          CONST dvoid     *elem,  
                          CONST dvoid     *elemind,  
                          OCIColl         *coll );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、*err* に記録され、OCI_ERROR が戻されます。OCIErrorGet() をコールして診断情報を取得します。

index (IN)

割当て先となる要素の索引です。

elem (IN)

割当て元となる要素（ソース要素）です。

elemind (IN) [オプション]

要素の NULL インジケータ情報へのポインタです。*elemind* == NULL の場合、割り当てられた要素の NULL インジケータ情報は、NULL 以外に設定されます。

coll (IN/OUT)

更新されるコレクションです。

コメント

コレクションの型が **NESTED TABLE** である場合は、要素が削除されているときと同様に、指定の索引位置に要素が存在しない場合があります。このような場合は、*index* の位置に指定の要素が挿入されます。それ以外の場合は、*index* の位置にある要素が *elem* の値で更新されます。

指定の要素はディープ・コピーされること、および *elem* は厳密に入力パラメータであることに注意してください。

この関数は、入力パラメータに **NULL** があるか、または指定の索引が指定のコレクションの上限を超えている場合にエラーを戻します。

関連関数

[OCIErrorGet\(\)](#)、[OCICollAssign\(\)](#)

OCICollGetElem()

用途

指定の索引位置にある要素へのポインタを取得します。

構文

```
sword OCICollGetElem ( OCIEnv          *env,
                      OCIError        *err,
                      CONST OCIColl   *coll,
                      sb4             index,
                      boolean         *exists,
                      dvoid           **elem,
                      dvoid           **elemind );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet() をコールして診断情報を取得します。

coll (IN)

このコレクション内の要素へのポインタが戻されます。

index (IN)

ポインタが戻される要素の索引です。

exists (OUT)

指定した索引の要素が存在しない場合は `FALSE` に設定されます。そうでない場合は `TRUE` に設定されます。

elem (OUT)

要求する要素のアドレスが戻されます。

elemind (OUT) [オプション]

NULL インジケータ情報のアドレスが戻されます。`elemind == NULL` の場合、NULL インジケータ情報は戻されません。

コメント

指定の位置にある要素のアドレスを取得します。また、この関数は、オプションで要素の NULL インジケータ情報のアドレスを戻します。

コレクションの各要素型と対応する要素ポインタの型を次の表に示します。要素ポインタは OCICollGetElem() の elem パラメータによって戻されます。

要素型	*elem の設定
Oracle 数値 (OCINumber)	OCINumber*
日付 (OCIDate)	OCIDate*
日時 (OCIDateTime)	OCIDateTime*
時間隔 (OCIInterval)	OCIInterval*
可変長文字列 (OCIStrng*)	OCIStrng**
可変長 RAW (OCIRaw*)	OCIRaw**
オブジェクト参照 (OCISRef*)	OCISRef**
LOB ロケータ (OCILobLocator*)	OCILobLocator**
オブジェクト型 (person など)	person*

OCICollGetElem() で戻される要素ポインタは、要素データにアクセスするために使用できる書式のみでなく、代入文のターゲットとして使用できる書式の場合もあります。

たとえば、要素型がオブジェクト参照 (**OCISRef***) であるコレクションの要素を反復するとします。OCICollGetElem() のコールにより、参照ハンドル (**OCISRef****) へのポインタが戻ります。コレクション要素へのポインタを取得した後、新しい参照を割り当てることでそれを修正できます。

この修正は、次のような REF 割当て関数によって実行できます。

```
sword OCISRefAssign( OCISEnv      *env,
                    OCISError    *err,
                    CONST OCISRef *source,
                    OCISRef      **target );
```

OCISRefAssign() の target パラメータの型は **OCISRef**** であることに注意してください。したがって OCICollGetElem() は、**OCISRef**** を戻します。*target が NULL の場合は、OCISRefAssign() によって新しい REF が割り当てられ、target パラメータに戻されます。

同様に、コレクション要素がタイプ文字列 (**OCIStrng***) のものである場合、OCICollGetElem() は文字列ハンドル (つまり、**OCIStrng****) へのポインタを戻します。OCIStrngAssign() または OCIStrngAssignText() を介して新しい文字列を割り当てる場合、ターゲットの型は **OCIStrng**** にしてください。

コレクションの要素型が Oracle 数値の場合、OCICollGetElem() は **OCINumber*** を返します。OCINumberAssign() のプロトタイプを次に示します。

```
sword OCINumberAssign(OCIError      *err,  
                      CONST OCINumber *from,  
                      OCINumber      *to);
```

この関数は、入力パラメータに NULL がある場合はエラーを返します。

関連関数

[OCIErrorGet\(\)](#)、[OCICollAssignElem\(\)](#)

OCICollIsLocator()

用途

コレクションがロケータベースであるかどうかを示します。

構文

```
sword OCICollIsLocator ( OCIEnv *env,  
                        OCLError *err,  
                        CONST OCIColl *coll,  
                        boolean *result );
```

パラメータ

env (IN)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。`OCLErrorGet()` をコールして診断情報を取得します。

coll (IN)

コレクション項目です。

result (OUT)

コレクション項目がロケータベースである場合は `TRUE` を戻します。それ以外の場合は `FALSE` を戻します。

コメント

この関数は、コレクションがロケータベースであるかどうかを確認するためのテストを行います。コレクション項目がロケータベースである場合は `result` パラメータに `TRUE` を戻します。それ以外の場合は `FALSE` を戻します。

関連関数

[OCLErrorGet\(\)](#)

OCICollMax()

用途

指定のコレクションの最大サイズを要素数単位で取得します。

構文

```
sb4 OCICollMax ( OCIEnv          *env,  
                  CONST OCIColl   *coll );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

coll (IN)

要素数が戻されるコレクションです。coll は有効なコレクション記述子を指し示している必要があります。

コメント

指定されたコレクションが保持できる要素の最大数を返します。値 0（ゼロ）は、そのコレクションには上限がないことを示します。

戻り値

指定のコレクションの上限。

関連関数

[OCIErrorGet\(\)](#)、[OCICollSize\(\)](#)

OCICollSize()

用途

指定のコレクションのカレント・サイズを要素数単位で取得します。

構文

```
sword OCICollSize ( OCIEnv          *env,
                    OCIErr         *err,
                    CONST OCIColl   *coll
                    sb4             *size );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEncCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。`OCIErrGet()` をコールして診断情報を取得します。

coll (IN)

要素の数が戻されるコレクションです。有効なコレクション識別子を指し示している必要があります。

size (OUT)

コレクション内の現行の要素数です。

コメント

指定されたコレクションの現行の要素数を戻します。NESTED TABLE の場合、要素を削除してもこのカウントは減分されません。したがって、このカウントには要素の削除によって発生した空が含まれています。切捨て操作 ([OCICollTrim\(\)](#)) は、切り捨てられた要素の数のみこのカウントを減分します。削除済みの要素を差し引いたカウントを取得するには、[OCITableSize\(\)](#) を使用します。

次の疑似コードで例をいくつか示します。

```
OCICollSize(...);
// assume 'size' returned is equal to 5
OCITableDelete(...); // delete one element
OCICollSize(...);
// 'size' returned is still 5
```

削除済みの要素を差し引いたカウントを取得するには、`OCITableSize()` を使用します。
前述の例を続けます。

```
OCITableSize(...)  
// 'size' returned is equal to 4
```

切捨て操作 (`OCICollTrim()`) は、切り捨てられた要素の数のみこのカウントを減分します。
前述の例を続けます。

```
OCICollTrim(...,1..); // trim one element  
OCICollSize(...);  
// 'size' returned is equal to 4
```

この関数は、コレクションのオブジェクト・キャッシュへのロード中にエラーが発生した場合、または入力パラメータに `NULL` がある場合にエラーを戻します。

関連関数

[OCIErrorGet\(\)](#)、[OCICollMax\(\)](#)

OCICollTrim()

用途

指定数の要素をコレクションの最後から切り捨てます。

構文

```
sword OCICollTrim ( OCIEnv      *env,  
                   OCIError    *err,  
                   sb4         trim_num,  
                   OCIColl     *coll );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。`OCIErrorGet()` をコールして診断情報を取得します。

trim_num (IN)

切り捨てる要素数です。

coll (IN/OUT)

この関数は、`trim_num` 要素を `coll` の最後から削除（解放）します。

コメント

要素は、コレクションの末尾から削除されます。`trim_num` がコレクションのカレント・サイズよりも大きい場合はエラーを戻します。

関連関数

[OCIErrorGet\(\)](#)、[OCICollSize\(\)](#)

OCIIterCreate()

用途

要素またはコレクションをスキャンするためのイテレータを作成します。

構文

```
sword OCIIterCreate ( OCIEnv          *env,  
                     OCIError       *err,  
                     CONST OCIColl   *coll,  
                     OCIIter        **itr );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、err に記録され、OCI_ERROR が戻されます。OCIErrorGet() をコールして診断情報を取得します。

coll (IN)

スキャンするコレクションです。このリリースでは、有効なコレクション型として VARRAY と NESTED TABLE が含まれています。

itr (OUT)

割り当てられたコレクション・イテレータのアドレスは、この関数で戻されます。

コメント

イテレータは、オブジェクト・キャッシュの中に作成されます。イテレータは、コレクションの先頭を指し示すように初期化されます。

イテレータの作成直後に OCIIterNext() をコールした場合は、コレクションの最初の要素が戻ります。イテレータの作成直後に OCIIterPrev() をコールした場合は、「コレクションの先頭にいる」ことを示すエラーが戻ります。

この関数は、入力パラメータに NULL がある場合はエラーを戻します。

関連関数

[OCIErrorGet\(\)](#)、[OCIIterDelete\(\)](#)

OCIIterDelete()

用途

コレクション・イテレータを削除します。

構文

```
sword OCIIterDelete ( OCIEnv          *env,  
                      OCIError       *err,  
                      OCIIter        **itr );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。`OCIErrorGet()` をコールして診断情報を取得します。

itr (IN/OUT)

コレクション・イテレータです。これは戻す前に破棄され `NULL` に設定されます。

コメント

[OCIIterCreate\(\)](#) のコールによって以前に作成されたイテレータを削除します。

この関数は、入力パラメータに `NULL` がある場合はエラーを戻します。

関連関数

[OCIErrorGet\(\)](#)、[OCIIterCreate\(\)](#)

OCIIterGetCurrent()

用途

カレント・イテレータ・コレクション要素へのポインタを取得します。

構文

```
sword OCIIterGetCurrent ( OCIEnv          *env,  
                           OCIError       *err,  
                           CONST OCIIter   *itr,  
                           dvoid          **elem,  
                           dvoid          **elemind );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、err に記録され、OCI_ERROR が戻されます。OCIErrorGet () をコールして診断情報を取得します。

itr (IN)

現在の要素を指し示すイテレータです。

elem (OUT)

イテレータが参照した要素のアドレスが戻されます。

elemind (OUT) [オプション]

要素の NULL インジケータ情報のアドレスが戻されます。elemind == NULL の場合、NULL インジケータ情報は戻されません。

コメント

カレント・イテレータ・コレクション要素へのポインタとそれに対応する NULL 情報を戻します。この関数は、入力パラメータに NULL がある場合はエラーを戻します。

関連関数

[OCIErrorGet\(\)](#)、[OCIIterNext\(\)](#)、[OCIIterPrev\(\)](#)

OCIIterInit()

用途

コレクションをスキャンするためのイテレータを初期化します。

構文

```
sword OCIIterInit ( OCIEnv          *env,  
                   OCIError        *err,  
                   CONST OCIColl    *coll,  
                   OCIIter          *itr );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。`OCIErrorGet()` をコールして診断情報を取得します。

coll (IN)

スキャンするコレクションです。`Oracle8i` 以上では、有効なコレクション型として `VARRAY` と `NESTED TABLE` が含まれています。

itr (IN/OUT)

割り当てられたコレクション・イテレータへのポインタです。

コメント

イテレータが、指定したコレクションの先頭を指し示すように初期化します。入力パラメータに `NULL` がある場合はエラーを戻します。この関数を使用すると次のことができます。

- コレクションの先頭を指し示すようにイテレータをリセットします。
- 割り当て済みのイテレータを再利用して別のコレクションをスキャンします。

関連関数

[OCIErrorGet\(\)](#)

OCIIterNext()

用途

次のイテレータ・コレクション要素へのポインタを取得します。

構文

```
sword OCIIterNext ( OCIEnv          *env,  
                   OCIError        *err,  
                   OCIIter         *itr,  
                   dvoid           **elem,  
                   dvoid           **elemind,  
                   boolean          *eoc);
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet() をコールして診断情報を取得します。

itr (IN/OUT)

次の要素を指し示すようにイテレータを更新します。

elem (OUT)

イテレータが次要素を指し示すように更新された後、要素のアドレスを戻します。

elemind (OUT) [オプション]

要素の NULL インジケータ情報のアドレスが戻されます。`elemind == NULL` の場合、NULL インジケータ情報は戻されません。

eoc (OUT)

イテレータがコレクションの末尾にある（つまり、次の要素が存在しない）場合は TRUE、それ以外の場合は FALSE となります。

コメント

この関数は、次のイテレータ・コレクション要素へのポインタおよびそれに対応する NULL 情報を戻します。また、次の要素を指し示すようにイテレータを更新します。

この関数の実行前にイテレータがコレクションの最後の要素を指し示している場合は、この関数をコールすると *eoc* フラグに TRUE が設定されます。この場合、イテレータは更新されません。

この関数は、入力パラメータに NULL がある場合はエラーを戻します。

関連関数

[OCIErrorGet\(\)](#)、[OCIIterGetCurrent\(\)](#)、[OCIIterPrev\(\)](#)

OCIIterPrev()

用途

直前のイテレータ・コレクション要素へのポインタを取得します。

構文

```
sword OCIIterPrev ( OCIEnv          *env,  
                    OCIError       *err,  
                    OCIIter        *itr,  
                    dvoid          **elem,  
                    dvoid          **elemind,  
                    boolean        *boc );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet() をコールして診断情報を取得します。

itr (IN/OUT)

前の要素を指し示すように更新されるイテレータです。

elem (OUT)

イテレータが前の要素を指し示すように更新された後に戻される、前の要素のアドレスです。

elemind (OUT) [オプション]

要素の NULL インジケータのアドレスが戻されます。`elemind == NULL` の場合、NULL インジケータは戻されません。

boc (OUT)

イテレータがコレクションの先頭にある（つまり、前の要素が存在しない）場合は TRUE、それ以外の場合は FALSE となります。

コメント

この関数は、直前のイテレータ・コレクション要素へのポインタおよびそれに対応する NULL 情報を戻します。イテレータは、前の要素を指し示すように更新されます。

この関数の実行前にイテレータがコレクションの最初の要素を指し示している場合は、この関数をコールすると *boc* フラグに TRUE が設定されます。この場合、イテレータは更新されません。

この関数は、入力パラメータに NULL がある場合はエラーを戻します。

関連関数

[OCIErrorGet\(\)](#)、[OCIIterGetCurrent\(\)](#)、[OCIIterNext\(\)](#)

OCI の日付関数、日時関数および時間隔関数

この項では、OCI の日付関数および時間隔関数について説明します。

表 18-3 日付関数

関数	用途
OCIDateAddDays() (18-30 ページ)	日数の加算または減算を行います。
OCIDateAddMonths() (18-31 ページ)	月の加算または減算を行います。
OCIDateAssign() (18-32 ページ)	日付を割り当てます。
OCIDateCheck() (18-33 ページ)	指定の日付が有効かどうかをチェックします。
OCIDateCompare() (18-35 ページ)	日付を比較します。
OCIDateDaysBetween() (18-36 ページ)	2 つの日付の間にある日数を取得します。
OCIDateFromText() (18-37 ページ)	文字列を日付に変換します。
OCIDateGetDate() (18-39 ページ)	日付の日付部分を取得します。
OCIDateGetTime() (18-40 ページ)	日付の時刻部分を取得します。
OCIDateLastDay() (18-41 ページ)	月末の日付を取得します。
OCIDateNextDay() (18-42 ページ)	指定された次の曜日の日付を取得します。
OCIDateSetDate() (18-43 ページ)	日付の日付部分を設定します。
OCIDateSetTime() (18-44 ページ)	日付の時刻部分を設定します。
OCIDateSysDate() (18-45 ページ)	現行のシステム日付および時刻を取得します。
OCIDateToText() (18-46 ページ)	日付を文字列に変換します。
OCIDateTimeAssign() (18-48 ページ)	日時の割当てを実行します。
OCIDateTimeCheck() (18-49 ページ)	指定の日付が有効かどうかをチェックします。
OCIDateTimeCompare() (18-51 ページ)	2 つの日時値を比較します。
OCIDateTimeConstruct() (18-52 ページ)	日時記述子を作成します。
OCIDateTimeConvert() (18-54 ページ)	ある日時型を別の日時型に変換します。
OCIDateTimeFromArray() (18-55 ページ)	サイズ <code>OCI_DT_ARRAYLEN</code> の配列を OCIDateTime 記述子に変換します。
OCIDateTimeFromText() (18-57 ページ)	指定された書式に従って、指定の文字列を OCIDateTime 記述子の Oracle 日時型に変換します。

表 18-3 日付関数（続き）

関数	用途
OCIDateTimeGetDate() (18-59 ページ)	日時値の日付部分（年、月、日）を取得します。
OCIDateTimeGetTime() (18-60 ページ)	日時値の時刻部分（時間、分、秒、小数秒）を取得します。
OCIDateTimeGetTimeZoneName() (18-62 ページ)	日時値のタイム・ゾーン名部分を取得します。
OCIDateTimeGetTimeZoneOffset() (18-63 ページ)	日時値のタイム・ゾーン（時間、分）部分を取得します。
OCIDateTimeIntervalAdd() (18-64 ページ)	日時に時間隔を加算して、結果の日時を生成します。
OCIDateTimeIntervalSub() (18-65 ページ)	日時から時間隔を減算して、その結果を日時に格納します。
OCIDateTimeSubtract() (18-66 ページ)	2 つの日時を入力値にして、その差異を時間隔に格納します。
OCIDateTimeSysTimeStamp() (18-67 ページ)	現行のシステム日付および時刻を、タイム・ゾーン付きタイムスタンプとして取得します。
OCIDateTimeToArray() (18-68 ページ)	OCIDateTime 記述子を配列に変換します。
OCIDateTimeToText() (18-70 ページ)	指定された日付を指定の書式の文字列に変換します。
OCIDateZoneToZone() (18-72 ページ)	あるタイム・ゾーンの日付を別のゾーンの日付に変換します。
OCIIntervalAdd() (18-74 ページ)	2 つの時間隔を加算して、その結果の時間隔を生成します。
OCIIntervalAssign() (18-75 ページ)	ある時間隔を別の時間隔にコピーします。
OCIIntervalCheck() (18-76 ページ)	時間隔の妥当性をチェックします。
OCIIntervalCompare() (18-78 ページ)	2 つの時間隔を比較します。
OCIIntervalDivide() (18-79 ページ)	時間隔を Oracle 数値で除算して、その結果の時間隔を生成します。
OCIIntervalFromNumber() (18-80 ページ)	Oracle 数値を時間隔に変換します。
OCIIntervalFromText() (18-81 ページ)	時間隔文字列が指定されている場合、その文字列で表現される時間隔を戻します。
OCIIntervalFromTZ() (18-83 ページ)	OCI_DTYPE_INTERVAL_DS を戻します。
OCIIntervalGetDaySecond() (18-84 ページ)	時間隔から日、時、分、秒の値を取得します。
OCIIntervalGetYearMonth() (18-86 ページ)	時間隔から年と月を取得します。
OCIIntervalMultiply() (18-87 ページ)	時間隔を Oracle 数値で乗算して、その結果の時間隔を生成します。

表 18-3 日付関数（続き）

関数	用途
<code>OCIIntervalSetDaySecond()</code> (18-88 ページ)	時間隔に日、時、分、秒を設定します。
<code>OCIIntervalSetYearMonth()</code> (18-90 ページ)	時間隔に年と月を設定します。
<code>OCIIntervalSubtract()</code> (18-91 ページ)	2 つの時間隔を減算し、その結果を時間隔に格納します。
<code>OCIIntervalToNumber()</code> (18-92 ページ)	時間隔を Oracle 数値に変換します。
<code>OCIIntervalToText()</code> (18-93 ページ)	時間隔が指定されている場合、その時間隔を表現する文字列を生成します。

OCIDateAddDays()

用途

指定の日付に日数を加算または減算します。

構文

```
sword OCIDateAddDays ( OCIError          *err,  
                        CONST OCIDate      *date,  
                        sb4                 num_days,  
                        OCIDate             *result );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。 `OCIErrorGet()` をコールして診断情報を取得します。

date (IN)

加算または減算の対象となる指定の日付です。

num_days (IN)

加算または減算される日数です。マイナス値の場合は減算となります。

result (IN/OUT)

`date` に対して日数を加算または減算した結果です。

コメント

この関数は、無効な日付が渡された場合はエラーを戻します。

関連関数

[OCIErrorGet\(\)](#)、[OCIDateAddMonths\(\)](#)

OCIDateAddMonths()

用途

指定の日付に月数を加算または減算します。

構文

```
sword OCIDateAddMonths ( OCIError          *err,  
                          CONST OCIDate     *date,  
                          sb4               num_months,  
                          OCIDate           *result );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet () をコールして診断情報を取得します。

date (IN)

加算または減算の対象となる指定の日付です。

num_months (IN)

加算または減算される月数です。マイナス値の場合は減算となります。

result (IN/OUT)

`date` に対して日数を加算または減算した結果です。

コメント

入力された `date` が月末の場合は、出力される日付も月末になるように適切な調整が行われます。たとえば、2 月 28 日 + 1 か月 = 3 月 31 日、11 月 30 日 - 3 か月 = 8 月 31 日となります。それ以外の場合、`result` 日付の日は `date` の日と同じになります。

この関数は、無効な日付が渡された場合はエラーを戻します。

関連関数

[OCIErrorGet \(\)](#)、[OCIDateAddDays \(\)](#)

OCIDateAssign()

用途

日付の割当てを実行します。

構文

```
sword OCIDateAssign ( OCLError      *err,  
                      CONST OCIDate  *from,  
                      OCIDate        *to );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。 `OCLErrorGet()` をコールして診断情報を取得します。

from (IN)

割り当てる日付です。

to (OUT)

割当てのターゲットです。

コメント

この関数は、ある **OCIDate** 変数の値を別の変数に割り当てます。

関連関数

[OCLErrorGet\(\)](#)、[OCIDateCheck\(\)](#)

OCIDateCheck()

用途

指定の日付が有効かどうかをチェックします。

構文

```
sword OCIDateCheck ( OCIError      *err,
                     CONST OCIDate  *date,
                     uword          *valid );
```

パラメータ

- err (IN/OUT)**
OCI エラー・ハンドルです。エラーがある場合は、`err`に記録され、`OCI_ERROR` が戻されます。 `OCIErrorGet()` をコールして診断情報を取得します。
- date (IN)**
チェックする日付です。
- valid (OUT)**
有効な日付に対して 0（ゼロ）を戻します。それ以外の場合は、次のように指定されたすべてのエラー・ビットの OR による組合せを戻します。

マクロ名	ビット数	エラー
OCI_DATE_INVALID_DAY	0x1	無効な日。
OCI_DATE_DAY_BELOW_VALID	0x2	無効な日の、有効な日に対する大 / 小を示すビット（1= 小）。
OCI_DATE_INVALID_MONTH	0x4	無効な月。
OCI_DATE_MONTH_BELOW_VALID	0x8	無効な月の、有効な月に対する大 / 小を示すビット（1= 小）。
OCI_DATE_INVALID_YEAR	0x10	無効な年。
OCI_DATE_YEAR_BELOW_VALID	0x20	無効な年の、有効な年に対する大 / 小を示すビット（1= 小）。
OCI_DATE_INVALID_HOUR	0x40	無効な時刻（時）。
OCI_DATE_HOUR_BELOW_VALID	0x80	無効な時刻（時）の、有効な時刻（時）に対する大 / 小を示すビット（1= 小）。
OCI_DATE_INVALID_MINUTE	0x100	無効な時刻（分）。
OCI_DATE_MINUTE_BELOW_VALID	0x200	無効な時刻（分）の、有効な時刻（分）に対する大 / 小を示すビット（1= 小）。

マクロ名	ビット数	エラー
OCI_DATE_INVALID_SECOND	0x400	無効な時刻（秒）。
OCI_DATE_SECOND_BELOW_VALID	0x800	無効な時刻（秒）の、有効な時刻（秒）に対する大 / 小を示すビット（1= 小）。
OCI_DATE_DAY_MISSING_FROM_1582	0x1000	日が 1582 から欠落しています。
OCI_DATE_YEAR_ZERO	0x2000	年が 0（ゼロ）に等しくありません。
OCI_DATE_INVALID_FORMAT	0x8000	無効な日付書式入力。

たとえば、渡された日付が 2/0/1990 25:61:10（月 / 日 / 年 時 : 分 : 秒書式）の場合は、次のエラーが戻されます。

OCI_DATE_INVALID_DAY | OCI_DATE_DAY_BELOW_VALID | OCI_DATE_INVALID_HOUR |
OCI_DATE_INVALID_MINUTE.

コメント

この関数は、*date* または *valid* ポインタが NULL の場合はエラーを戻します。

関連関数

[OCIErrorGet\(\)](#)、[OCIDateCompare\(\)](#)

OCIDateCompare()

用途

2 つの日付を比較します。

構文

```
sword OCIDateCompare ( OCIError          *err,  
                        CONST OCIDate      *date1,  
                        CONST OCIDate      *date2,  
                        sword              *result );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet () をコールして診断情報を取得します。

date1、date2 (IN)

比較する日付です。

result (OUT)

比較結果は次のとおりです。

比較結果	<i>result</i> パラメータの出力
date1 < date2	-1
date1 = date2	0
date1 > date2	1

コメント

この関数は、無効な日付が渡された場合はエラーを戻します。

関連関数

[OCIErrorGet \(\)](#)、[OCIDateCheck \(\)](#)

OCIDateDaysBetween()

用途

2 つの日付間の日数を取得します。

構文

```
sword OCIDateDaysBetween ( OCIError          *err,  
                           CONST OCIDate      *date1,  
                           CONST OCIDate      *date2,  
                           sb4                *num_days );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。 `OCIErrorGet()` をコールして診断情報を取得します。

date1 (IN)

入力日付です。

date2 (IN)

入力日付です。

num_days (OUT)

`date1` と `date2` の間の日数です。

コメント

`date1` と `date2` の間の日数の計算時には、時刻は無視されます。

この関数は、無効な日付が渡された場合はエラーを戻します。

関連関数

[OCIErrorGet\(\)](#)、[OCIDateCheck\(\)](#)

OCIDateFromText()

用途

指定された書式に従って、文字列を日付型に変換します。

構文

```
sword OCIDateFromText ( OCLError          *err,
                        CONST text        *date_str,
                        ub4               d_str_length,
                        CONST text        *fmt,
                        ub1               fmt_length,
                        CONST text        *lang_name,
                        ub4               lang_length,
                        OCIDate           *date );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、*err* に記録され、OCI_ERROR が戻されます。OCIErrorGet () をコールして診断情報を取得します。

date_str (IN)

Oracle 日付に変換される入力文字列です。

d_str_length (IN)

入力文字列のサイズです。長さが -1 の場合、*date_str* はヌル文字で終了する文字列として処理されます。

fmt (IN)

変換書式です。*fmt* が NULL ポインタの場合、文字列は 'DD-MON-YY' 書式になります。

fmt_length (IN)

fmt パラメータの長さです。

lang_name (IN)

日および月の名前と省略が指定される言語です。*lang_name* がヌル文字列 (text *) 0 の場合は、そのセッションのデフォルト言語が使用されます。

lang_length (IN)

lang_name パラメータの長さです。

date (OUT)

日付に変換された指定文字列です。

コメント

書式および多言語パラメータの詳細は、『Oracle9i SQL リファレンス』の TO_DATE 変換関数の説明を参照してください。

この関数は、無効な書式、言語または入力文字列を受け取った場合はエラーを戻します。

関連関数

[OCIErrorGet\(\)](#)、[OCIDateToText\(\)](#)

OCIDateGetDate()

用途

Oracle 日付に格納されている年月日を取得します。

構文

```
void OCIDateGetDate ( CONST OCIDate      *date,  
                      sb2                *year,  
                      ub1                *month,  
                      ub1                *day );
```

パラメータ

date (IN)

年、月、日データの取出し元となる Oracle 日付です。

year (OUT)

戻される年の値です。

month (OUT)

戻される月の値です。

day (OUT)

戻される日の値です。

コメント

なし

関連関数

[OCIDateSetDate\(\)](#)、[OCIDateGetTime\(\)](#)

OCIDateGetTime()

用途

Oracle 日付に格納されている時刻を取得します。

構文

```
void OCIDateGetTime ( CONST OCIDate      *date,
                      ub1                  *hour,
                      ub1                  *min,
                      ub1                  *sec );
```

パラメータ

date (IN)

時間データの取出し元となる Oracle 日付です。

hour (OUT)

戻される時間の値です。

min (OUT)

戻される分の値です。

sec (OUT)

戻される秒の値です。

コメント

時、分、秒の時刻情報を戻します。

関連関数

[OCIDateSetTime\(\)](#)、[OCIDateGetDate\(\)](#)

OCIDateLastDay()

用途

指定された日付の月の最後の日の日付を取得します。

構文

```
sword OCIDateLastDay ( OCIError          *err,  
                       CONST OCIDate      *date,  
                       OCIDate            *last_day );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、OCI_ERROR が戻されます。OCIErrorGet () をコールして診断情報を取得します。

date (IN)

入力日付です。

last_day (OUT)

date の月の最後の日付です。

コメント

この関数は、無効な日付が渡された場合はエラーを戻します。

関連関数

[OCIErrorGet \(\)](#)、[OCIDateGetDate \(\)](#)

OCIDateNextDay()

用途

指定した日付以降で、指定した曜日の最初の日付を取得します。

構文

```
sword OCIDateNextDay ( OCIError          *err,  
                        CONST OCIDate      *date,  
                        CONST OraText      *day,  
                        ub4                 day_length,  
                        OCIDate            *next_day );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、*err* に記録され、OCI_ERROR が戻されます。OCIErrorGet() をコールして診断情報を取得します。

date (IN)

戻される日付は、この日付よりも後になります。

day (IN)

これによって指定された週の最初の日が戻されます。

day_length (IN)

文字列 *day* のバイト長です。

next_day (OUT)

date 以降で、*day* によって指定された週の最初の日付です。

コメント

date 以降で、*day* によって指定された週の最初の曜日の日付を戻します。

例

1996 年 4 月 18 日（木曜日）の次の月曜日の日付を取得します。

```
OCIDateNextDay(&err, '18-APR-96', 'MONDAY', strlen('MONDAY'), &next_day)
```

OCIDateNextDay() は '22-APR-96' を戻します。

この関数は、無効な日付または曜日が渡された場合はエラーを戻します。

関連関数

[OCIErrorGet\(\)](#)、[OCIDateGetDate\(\)](#)

OCIDateSetDate()

用途

Oracle 日付に値を設定します。

構文

```
void OCIDateSetDate ( OCIDate      *date,
                      sb2          year,
                      ub1          month,
                      ub1          day );
```

パラメータ

date (OUT)

時間データ設定の対象となる Oracle 日付です。

year (IN)

設定する年の値です。

month (IN)

設定する月の値です。

day (IN)

設定する日の値です。

コメント

なし

関連関数

[OCIDateGetDate\(\)](#)

OCIDateSetTime()

用途

Oracle 日付に時刻情報を設定します。

構文

```
void OCIDateSetTime ( OCIDate      *date,  
                      ub1           hour,  
                      ub1           min,  
                      ub1           sec );
```

パラメータ

date (OUT)

時間データ設定の対象となる Oracle 日付です。

hour (IN)

設定する時間の値です。

min (IN)

設定する分の値です。

sec (IN)

設定する秒の値です。

コメント

なし

関連関数

[OCIDateGetTime\(\)](#)

OCIDateSysDate()

用途

クライアントの現行のシステム日付およびシステム時刻を取得します。

構文

```
sword OCIDateSysDate ( OCIError      *err,  
                       OCIDate       *sys_date );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet () をコールして診断情報を取得します。

sys_date (OUT)

クライアントの現行日付およびシステム時刻です。

コメント

なし

関連関数

[OCIErrorGet \(\)](#)

OCIDateToText()

用途

日付型を文字列に変換します。

構文

```
sword OCIDateToText ( OCIError          *err,
                     CONST OCIDate      *date,
                     CONSTOraText OraText *fmt,
                     ub1                fmt_length,
                     CONST OraText      *lang_name,
                     ub4                lang_length,
                     ub4                *buf_size,
                     OraText            *buf );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、*err* に記録され、OCI_ERROR が戻されます。OCIErrorGet () をコールして診断情報を取得します。

date (IN)

変換する Oracle 日付です。

fmt (IN)

変換書式です。NULL、つまり (text *) 0 の場合、日付はデフォルトの日付書式 DD-MON-YY の文字列に変換されます。

fmt_length (IN)

fmt パラメータの長さです。

lang_name (IN)

月および日の名前と略称が戻される際の言語を指定します。*lang_name* が NULL ((text *) 0) の場合は、セッションのデフォルト言語が使用されます。

lang_length (IN)

lang_name パラメータの長さです。

buf_size (IN/OUT)

- バッファのサイズです (IN)。
- このパラメータ (OUT) に、結果の文字列のサイズが戻されます。

buf (OUT)

変換された文字列が配置されるバッファです。

コメント

指定された日付を指定の書式の文字列に変換します。変換されてヌル文字で終了する日付文字列は、*buf* に格納されます。

書式および多言語パラメータの詳細は、『Oracle9i SQL リファレンス』の TO_DATE 変換関数の説明を参照してください。

この関数は、バッファが小さすぎる場合または無効な書式や認識できない言語が渡された場合はエラーを戻します。オーバーフローもエラーの原因となります。たとえば、値 10 を書式 9 に変換すると、エラーが発生します。

関連関数

[OCIErrorGet\(\)](#)、[OCIDateFromText\(\)](#)

OCIDateTimeAssign()

用途

日時の割当てを実行します。

構文

```
sword OCIDateTimeAssign ( dvoid          *hndl,
                          OCIError       *err,
                          CONST OCIDateTime *from,
                          OCIDateTime    *to );
```

パラメータ

hndl (IN)

OCI ユーザー・セッション・ハンドルまたは環境ハンドルです。ユーザー・セッション・ハンドルが渡されると、セッションの NLS_LANGUAGE およびセッションの NLS_CALENDAR で変換が行われます。渡されない場合は、デフォルトが使用されます。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、*err* に記録され、OCI_ERROR が戻されます。OCIErrorGet () をコールして診断情報を取得します。

from (IN)

割り当てるソース (*rhs*) 日時です。

to (OUT)

割当てのターゲット (*lhs*) です。

コメント

この関数は、*type* パラメータの説明にリストされている日時型に関して、*from* 日時から *to* 日時への割当てを実行します。

出力の *type* は入力と同じです。

戻り値

OCI_SUCCESS

OCI_ERROR

関連関数

[OCIDateTimeCheck\(\)](#)、[OCIDateTimeConstruct\(\)](#)

OCIDateTimeCheck()

用途

指定の日付が有効かどうかをチェックします。

構文

```
sword OCIDateTimeCheck ( dvoid          *hndl,
                        OCIError        *err,
                        CONST OCIDateTime *date,
                        ub4              *valid );
```

パラメータ

hndl (IN)
OCI ユーザー・セッション・ハンドルまたは環境ハンドルです。ユーザー・セッション・ハンドルが渡されると、セッションの NLS_LANGUAGE およびセッションの NLS_CALENDAR で変換が行われます。渡されない場合は、デフォルトが使用されます。

err (IN/OUT)
OCI エラー・ハンドルです。エラーがある場合は、err に記録され、OCI_ERROR が戻されます。OCIErrorGet () をコールして診断情報を取得します。

date (IN)
チェックする日付です。

valid (OUT)
有効な日付の場合は 0（ゼロ）を戻します。それ以外の場合は、次のように指定されたすべてのエラー・ビットの OR による組合せを戻します。

マクロ名	ビット数	エラー
OCI_DT_INVALID_DAY	0x1	無効な日。
OCI_DT_DAY_BELOW_VALID	0x2	無効な日の、有効な日に対する大 / 小を示すビット（1= 小）。
OCI_DT_INVALID_MONTH	0x4	無効な月。
OCI_DT_MONTH_BELOW_VALID	0x8	無効な月の、有効な月に対する大 / 小を示すビット（1= 小）。
OCI_DT_INVALID_YEAR	0x10	無効な年。
OCI_DT_YEAR_BELOW_VALID	0x20	無効な年の、有効な年に対する大 / 小を示すビット（1= 小）。
OCI_DT_INVALID_HOUR	0x40	無効な時刻（時）。

マクロ名	ビット数	エラー
OCI_DT_HOUR_BELOW_VALID	0x80	無効な時刻（時）の、有効な時刻（時）に対する大 / 小を示すビット（1= 小）。
OCI_DT_INVALID_MINUTE	0x100	無効な時刻（分）。
OCI_DT_MINUTE_BELOW_VALID	0x200	無効な時刻（分）の、有効な時刻（分）に対する大 / 小を示すビット（1= 小）。
OCI_DT_INVALID_SECOND	0x400	無効な時刻（秒）。
OCI_DT_SECOND_BELOW_VALID	0x800	無効な時刻（秒）の、有効な時刻（秒）に対する大 / 小を示すビット（1= 小）。
OCI_DT_DAY_MISSING_FROM_1582	0x1000	日が 1582 から欠落しています。
OCI_DT_YEAR_ZERO	0x2000	年が 0（ゼロ）に等しくありません。
OCI_DT_INVALID_TIMEZONE	0x4000	無効なタイム・ゾーン。
OCI_DT_INVALID_FORMAT	0x8000	無効な日付書式入力。

たとえば、渡された日付が 2/0/1990 25:61:10（月 / 日 / 年 時 : 分 : 秒書式）の場合は、次のエラーが戻されます。

```
OCI_DT_INVALID_DAY | OCI_DT_DAY_BELOW_VALID |
OCI_DT_INVALID_HOUR | OCI_DT_INVALID_MINUTE.
```

戻り値

- OCI_SUCCESS
- OCI_INVALID_HANDLE（err が NULL ポインタの場合）
- OCI_ERROR（date または valid が NULL ポインタの場合）

関連関数

```
OCIDateTimeAssign()
```

OCIDateTimeCompare()

用途

2 つの日時値を比較します。

構文

```
sword OCIDateTimeCompare ( dvoid          *hndl,
                           OCIError       *err,
                           CONST OCIDateTime *date1,
                           CONST OCIDateTime *date2,
                           sword          *result );
```

パラメータ

hndl (IN/OUT)
OCI ユーザー・セッション・ハンドルまたは環境ハンドルです。

err (IN/OUT)
OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。 `OCIErrorGet()` をコールして診断情報を取得します。

date1、date2 (IN)
比較する日付です。

result (OUT)
比較結果は次のとおりです。

比較結果	<i>result</i> パラメータの出力
date1 < date2	-1
date1 = date2	0
date1 > date2	1

戻り値

- OCI_SUCCESS
- OCI_INVALID_HANDLE (`err` が NULL ポインタの場合)
- OCI_ERROR (無効な日付の場合、または入力日付引数が相互に比較可能な型でない場合)

関連関数

```
OCIDateTimeConstruct()
```

OCIDateTimeConstruct()

用途

日時記述子を作成します。

構文

```
sword OCIDateTimeConstruct ( dvoid          *hdl,
                             OCIError       *err,
                             OCIDateTime    *datetime,
                             sb2            year,
                             ub1            month,
                             ub1            day,
                             ub1            hour,
                             ub1            min,
                             ub1            sec,
                             ub4            fsec,
                             OraText        *timezone,
                             size_t         timezone_length );
```

パラメータ

hdl (IN)

OCI ユーザー・セッション・ハンドルまたは環境ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、OCI_ERROR が戻されます。OCIErrorGet () をコールして診断情報を取得します。

datetime (IN)

OCIDateTime 記述子へのポインタです。

year (IN)

年の値です。

month (IN)

月の値です。

day (IN)

日の値です。

hour (IN)

時間の値です。

min (IN)

分の値です。

sec (IN)

秒の値です。

fsec (IN)

小数秒の値です。

timezone (IN)

タイム・ゾーン文字列です。

timezone_length (IN)

タイム・ゾーン文字列の長さです。

コメント

日時の型は、**OCIDateTime** 記述子の型です。型に基づいた関連フィールドのみを使用します。タイム・ゾーンを持つ型の場合、日付フィールドと時刻フィールドは、指定されたタイム・ゾーンのローカル・タイムとみなされます。

タイム・ゾーンが指定されていない場合は、セッションのデフォルト・タイム・ゾーンが使用されます。

戻り値

OCI_SUCCESS

OCI_ERROR (日時が無効な場合)

関連関数

[OCIDateTimeAssign\(\)](#)、[OCIDateTimeConvert\(\)](#)

OCIDateTimeConvert()

用途

ある日時型を別の日時型に変換します。

構文

```
sword OCIDateTimeConvert ( dvoid          *hndl,  
                           OCIError       *err,  
                           OCIDateTime    *indate,  
                           OCIDateTime    *outdate );
```

パラメータ

hndl (IN)

OCI ユーザー・セッション・ハンドルまたは環境ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。`OCIErrorGet()` をコールして診断情報を取得します。

indate (IN)

入力日付へのポインタです。

outdate (OUT)

出力日時へのポインタです。

コメント

この関数は、ある日時型を別の日時型に変換します。結果タイプは、`outdate` 記述子の型になります。タイム・ゾーンを持たない日時をタイム・ゾーンを持つ日時に変換する場合は、セッションのデフォルト・タイム・ゾーン (`ORA_SDTZ`) が使用されます。

戻り値

`OCI_SUCCESS`

`OCI_INVALID_HANDLE` (`err` が `NULL` の場合)

`OCI_ERROR` (指定の入力値では変換が不可能な場合)

関連関数

[OCIDateTimeCheck\(\)](#)

OCIDateTimeFromArray()

用途

日付が含まれている配列を **OCIDateTime** 記述子に変換します。

構文

```
sword OCIDateTimeFromArray ( dvoid          *hndl,
                             OCIError      *err,
                             CONST ub1     *inarray,
                             ub4           *len,
                             ub1          type,
                             OCIDateTime  *datetime,
                             CONST OCIInterval *reftz,
                             ub1          fsprec );
```

パラメータ

hndl (IN)

OCI ユーザー・セッション・ハンドルまたは環境ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、*err* に記録され、OCI_ERROR が戻されます。OCIErrorGet () をコールして診断情報を取得します。

inarray(IN)

日付が含まれる *ub1* の配列です。

len (IN)

inarray の長さです。

type (IN)

結果の日時の型です。配列は、特定の SQLT 型に変換されます。

datetime (OUT)

OCIDateTime 記述子へのポインタです。

reftz (IN)

SQLT_TIMESTAMP_LTZ 型の変換時に参照として使用する **OCIInterval** の記述子です。

fsprec (IN)

結果の日時の小数秒の精度です。

戻り値

OCI_SUCCESS

OCI_ERROR (*type* が無効な場合)

関連関数

[OCIDateTimeFromText\(\)](#)、[OCIDateTimeToArray\(\)](#)

OCIDateTimeFromText()

用途

指定された書式に従って、指定の文字列を **OCIDateTime** 記述子の Oracle 日時型に変換します。

構文

```
sword OCIDateTimeFromText ( dvoid          *hndl,
                           OCIError       *err,
                           CONST OraText  *date_str,
                           size_t         dstr_length,
                           CONST OraText  *fmt,
                           ubl            fmt_length,
                           CONST OraText  *lang_name,
                           size_t         lang_length,
                           OCIDateTime    *datetime );
```

パラメータ

hndl (IN)

OCI ユーザー・セッション・ハンドルまたは環境ハンドルです。ユーザー・セッション・ハンドルが渡されると、セッションの NLS_LANGUAGE およびセッションの NLS_CALENDAR で変換が行われます。渡されない場合は、デフォルトが使用されます。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、*err* に記録され、OCI_ERROR が戻されます。OCIErrorGet () をコールして診断情報を取得します。

date_str (IN)

Oracle 日時に変換される入力文字列です。

dstr_length (IN)

入力文字列のサイズです。長さが -1 の場合、*date_str* はヌル文字で終了する文字列として処理されます。

fmt (IN)

変換書式です。*fmt* が NULL ポインタの場合、文字列は日時型のデフォルトの書式になります。

fmt_length (IN)

fmt パラメータの長さです。

lang_name (IN)

日および月の名前と略称が指定される際の言語を指定します。*lang_name* が NULL (*lang_name* = (text *) 0) の場合は、セッションのデフォルト言語が使用されます。

lang_length (IN)

lang_name パラメータの長さです。

datetime (OUT)

日付に変換された指定文字列です。

コメント

書式の引数についての説明は、Oracle8i 以上の SQL リファレンスで TO_DATE 変換関数の説明を参照してください。

戻り値

OCI_SUCCESS

OCI_INVALID_HANDLE (*err* が NULL の場合)

OCI_ERROR (次のいずれかの場合)

- 無効な書式が使用されている場合
- 不明な言語が使用されている場合
- 無効な入力文字列が使用されている場合

関連関数

[OCIDateTimeToText\(\)](#)、[OCIDateTimeFromArray\(\)](#)

OCIDateTimeGetDate()

用途

日時値の日付部分（年、月、日）を取得します。

構文

```
void OCIDateTimeGetDate ( dvoid          *hndl,  
                          OCIError      *err,  
                          CONST OCIDateTime *datetime,  
                          sb2           *year,  
                          ub1           *month,  
                          ub1           *day );
```

パラメータ

hndl (IN)

OCI ユーザー・セッション・ハンドルまたは環境ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。 `OCIErrorGet()` をコールして診断情報を取得します。

datetime (IN)

日付情報が取り出される **OCIDateTime** 記述子へのポインタです。

year (OUT)

month (OUT)

day (OUT)

取り出された年、月および日の値です。

コメント

この関数は、日時値の日付部分（年、月、日）を取得します。

戻り値

`OCI_SUCCESS`

`OCI_ERROR`（入力型が `SQLT_TIME` または `OCI_TIME_TZ` の場合）

関連関数

[OCIDateTimeGetTime\(\)](#)

OCIDateTimeGetTime()

用途

日時値の時刻部分（時間、分、秒、小数秒）を取得します。

構文

```
void OCIDateTimeGetTime ( dvoid          *hndl,
                          OCIError      *err,
                          OCIDateTime   *datetime,
                          ub1           *hour,
                          ub1           *min,
                          ub1           *sec,
                          ub4           *fsec );
```

パラメータ

hndl (IN)

OCI ユーザー・セッション・ハンドルまたは環境ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet() をコールして診断情報を取得します。

datetime (IN)

時刻情報が取り出される **OCIDateTime** 記述子へのポインタです。

hour (OUT)

取り出される時間の値です。

min (OUT)

取り出される分の値です。

sec (OUT)

取り出される秒の値です。

fsec (OUT)

取り出される小数秒の値です。

コメント

この関数は、日時値の時刻部分（時間、分、秒、小数秒）を取得します。

この関数は、指定の日時に時間情報が含まれていない場合はエラーを戻します。

戻り値

OCI_SUCCESS

OCI_ERROR（日時に時間（SQLT_DATE）が含まれていない場合）

関連関数

`OCIDateTimeGetDate()`

OCIDateTimeGetTimeZoneName()

用途

日時のタイム・ゾーン名部分を取得します。

構文

```
void OCIDateTimeGetTimeZoneName ( dvoid          *hndl,
                                OCIError       *err,
                                CONST OCIDateTime *datetime,
                                ub1            *buf,
                                ub4            *buflen, );
```

パラメータ

hndl (IN)

OCI ユーザー・セッション・ハンドルまたは環境ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet () をコールして診断情報を取得します。

datetime (IN)

`OCIDateTime` 記述子へのポインタです。

buf (OUT)

取り出されたタイム・ゾーン名を格納するバッファです。

buflen (IN/OUT)

(IN) はバッファのサイズです。(OUT) は名前フィールドのサイズです。

コメント

この関数は、日時値の時刻部分（時間、分、秒、小数秒）を取得します。

この関数は、指定の日時に時間情報が含まれていない場合はエラーを戻します。

戻り値

`OCI_SUCCESS`

`OCI_ERROR`（日時にタイム・ゾーン（`SQLT_DATE`、`SQLT_TIMESTAMP`）が含まれていない場合）

関連関数

[OCIDateTimeGetDate\(\)](#)、[OCIDateTimeGetTimeZoneOffset\(\)](#)

OCIDateTimeGetTimeZoneOffset()

用途

日時値のタイム・ゾーン（時間、分）部分を取得します。

構文

```
void OCIDateTimeGetTimeZoneOffset ( dvoid          *hndl,
                                     OCIError       *err,
                                     CONST OCIDateTime *datetime,
                                     sb1             *hour,
                                     sb1             *min, );
```

パラメータ

hndl (IN)

OCI ユーザー・セッション・ハンドルまたは環境ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、OCI_ERROR が戻されます。OCIErrorGet () をコールして診断情報を取得します。

datetime (IN)

OCIDateTime 記述子へのポインタです。

hour (OUT)

取り出されるタイム・ゾーンの時間の値です。

min (OUT)

取り出されるタイム・ゾーンの分の値です。

コメント

この関数は、指定の日時値からタイム・ゾーンの時間と分の部分を取得します。

この関数は、指定の日時に時間情報が含まれていない場合はエラーを戻します。

戻り値

OCI_SUCCESS

OCI_ERROR（日時にタイム・ゾーン（SQLT_DATE、SQLT_TIMESTAMP）が含まれていない場合）

関連関数

[OCIDateTimeGetDate\(\)](#)、[OCIDateTimeGetTimeZoneName\(\)](#)

OCIDateTimeIntervalAdd()

用途

時間隔を日時に加算して、結果の日時を生成します。

構文

```
sword OCIDateTimeIntervalAdd ( dvoid          *hndl,
                              OCIError       *err,
                              OCIDateTime    *datetime,
                              OCIInterval    *inter,
                              OCIDateTime    *outdatetime );
```

パラメータ

hndl (IN)

ユーザー・セッション・ハンドルまたは環境ハンドルです。セッション・ハンドルが渡されると、セッションのデフォルト・カレンダーで加算が行われます。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、OCI_ERROR が戻されます。OCIErrorGet () をコールして診断情報を取得します。

datetime (IN)

入力日時へのポインタです。

inter (IN)

入力時間隔へのポインタです。

outdatetime (OUT)

出力日時へのポインタです。出力日時と入力日時の型は同じになります。

戻り値

OCI_SUCCESS (関数が正常に完了した場合)

OCI_INVALID_HANDLE (err が NULL ポインタの場合)

OCI_ERROR (結果の日付が紀元前 4713 年 1 月 1 日より前、または 9999 年 12 月 31 日より後の場合)

関連関数

[OCIDateTimeIntervalSub\(\)](#)

OCIDateTimeIntervalSub()

用途

日時から時間隔を減算して、その結果を日時に格納します。

構文

```
sword OCIDateTimeIntervalSub ( dvoid          *hndl,
                               OCIError       *err,
                               OCIDateTime    *datetime,
                               OCIInterval    *inter,
                               OCIDateTime    *outdatetime );
```

パラメータ

hndl (IN)

ユーザー・セッション・ハンドルまたは環境ハンドルです。セッション・ハンドルが渡されると、セッションのデフォルト・カレンダーで減算が行われます。時間隔は、セッション・カレンダー内であるとみなされます。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、OCI_ERROR が戻されます。OCIErrorGet() をコールして診断情報を取得します。

datetime (IN)

入力日時値へのポインタです。

inter (IN)

入力時間隔へのポインタです。

outdatetime (OUT)

出力日時へのポインタです。出力日時と入力日時の型は同じになります。

戻り値

OCI_SUCCESS (関数が正常に完了した場合)

OCI_INVALID_HANDLE (**err** が NULL ポインタの場合)

OCI_ERROR (結果の日付が紀元前 4713 年 1 月 1 日より前、または 9999 年 12 月 31 日より後の場合)

関連関数

[OCIDateTimeIntervalAdd\(\)](#)

OCIDateTimeSubtract()

用途

2つの日時を入力値にして、その差異を時間隔に格納します。

構文

```
sword OCIDateTimeSubtract ( dvoid          *hdl,  
                             OCIError      *err,  
                             OCIDateTime   *indate1,  
                             OCIDateTime   *indate2,  
                             OCIInterval   *inter );
```

パラメータ

hdl (IN)

OCI ユーザー・セッション・ハンドルまたは環境ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet () をコールして診断情報を取得します。

indate1(IN)

減数へのポインタです。

indate2(IN)

被減数へのポインタです。

inter (OUT)

出力時間隔へのポインタです。

戻り値

`OCI_SUCCESS` (関数が正常に完了した場合)

`OCI_INVALID_HANDLE` (`err` が `NULL` ポインタの場合)

`OCI_ERROR` (入力日時が比較可能な型でない場合)

関連関数

[OCIDateTimeCompare\(\)](#)

OCIDateTimeSysTimeStamp()

用途

現行のシステム日付および時刻を、タイム・ゾーン付きタイムスタンプとして取得します。

構文

```
sword OCIDateTimeSysTimeStamp ( dvoid          *hndl,  
                                OCIError       *err,  
                                OCIDateTime    *sys_date );
```

パラメータ

hndl (IN)

OCI ユーザー・セッション・ハンドルまたは環境ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、*err* に記録され、OCI_ERROR が戻されます。OCIErrorGet() をコールして診断情報を取得します。

sys_date (OUT)

出力タイムスタンプへのポインタです。

戻り値

OCI_SUCCESS

OCI_INVALID_HANDLE (*err* が NULL ポインタの場合)

関連関数

[OCIDateSysDate\(\)](#)

OCIDateTimeToArray()

用途

OCIDateTime 記述子を配列に変換します。

構文

```
sword OCIDateTimeToArray ( dvoid          *hndl,
                           OCIError      *err,
                           CONST OCIDateTime *datetime,
                           CONST OCIInterval *reftz,
                           ub1           *outarray,
                           ub4           *len,
                           ub1           fsprec );
```

パラメータ

hndl (IN)

OCI ユーザー・セッション・ハンドルまたは環境ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、OCI_ERROR が戻されます。OCIErrorGet () をコールして診断情報を取得します。

datetime (IN)

OCIDateTime 記述子へのポインタです。

reftz (IN)

SQL_TIMESTAMP_LTZ 型の変換時に参照として使用する **OCIInterval** の記述子です。

outarray(OUT)

日付が含まれるバイトの配列です。

len (OUT)

outarray の長さです。

fsprec (IN)

配列の小数秒の精度です。

コメント

OCI によって配列が割り当てられ、その長さが戻されます。

戻り値

OCI_SUCCESS

OCI_ERROR（日時が無効な場合）

関連関数

[OCIDateTimeToText\(\)](#)、[OCIDateTimeFromArray\(\)](#)

OCIDateTimeToText()

用途

指定された日付を指定の書式の文字列に変換します。

構文

```
sword OCIDateTimeToText ( dvoid          *hdl,  
                          OCIError      *err,  
                          CONST OCIDateTime *date,  
                          CONST OraText  *fmt,  
                          ub1            fmt_length,  
                          ub1            fsprec,  
                          CONST OraText  *lang_name,  
                          size_t         lang_length,  
                          ub4            *buf_size,  
                          OraText        *buf );
```

パラメータ

hdl (IN)

OCI ユーザー・セッション・ハンドルまたは環境ハンドルです。ユーザー・セッション・ハンドルが渡されると、セッションの NLS_LANGUAGE およびセッションの NLS_CALENDAR で変換が行われます。渡されない場合は、デフォルトが使用されます。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet() をコールして診断情報を取得します。

date (IN)

変換する Oracle 日時値です。

fmt (IN)

変換書式です。ヌル文字列ポインタ (`text*`) 0 の場合、日付は、その型に対するデフォルトの書式で文字列に変換されます。

fmt_length (IN)

fmt パラメータの長さです。

fsprec (IN)

戻される小数秒の精度を指定します。

lang_name (IN)

日および月の名前と略称が戻される際の言語を指定します。*lang_name* が NULL (*lang_name* = (`OraText *`) 0) の場合は、セッションのデフォルト言語が使用されます。

lang_length (IN)

lang_name パラメータの長さです。

buf_size (IN/OUT)

(IN) は *buf* バッファのサイズです。

(OUT) は変換結果の文字列のサイズです。

buf (OUT)

変換された文字列が配置されるバッファです。

コメント

書式および多言語の引数についての説明は、Oracle9i 以上の SQL リファレンスで TO_DATE 変換関数の説明を参照してください。変換されてヌル文字で終了する日付文字列は、*buf* バッファに格納されます。

戻り値

OCI_SUCCESS

OCI_INVALID_HANDLE (*err* が NULL の場合)

OCI_ERROR (次のいずれかの場合)

- バッファが小さすぎる場合
- 無効な書式が使用されている場合
- 不明な言語が使用されている場合
- オーバーフロー・エラーがある場合

関連関数

[OCIDateTimeFromText\(\)](#)

OCIDateZoneToZone()

用途

あるタイム・ゾーンの日付を別のタイム・ゾーンの日付に変換します。

構文

```
sword OCIDateZoneToZone ( OCIError          *err,  
                          CONST OCIDate      *date1,  
                          CONST OraText      *zon1,  
                          ub4                zon1_length,  
                          CONST OraText      *zon2,  
                          ub4                zon2_length,  
                          OCIDate            *date2 );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、OCI_ERROR が戻されます。OCIErrorGet () をコールして診断情報を取得します。

date1 (IN)

変換する日付です。

zon1 (IN)

入力日付のゾーンです。

zon1_length (IN)

zon1 のバイト長です。

zon2 (IN)

変換先のゾーンです。

zon2_length (IN)

zon2 のバイト長です。

date2 (OUT)

zon2 での変換日付です。

コメント

タイム・ゾーン *zon1* の指定の日付 *date1* を、タイム・ゾーン *zon2* の日付 *date2* に変換します。北米のタイム・ゾーンのみが対象です。

有効なゾーン文字列のリストは、『Oracle9i SQL リファレンス』の NEW_TIME 関数の説明を参照してください。有効なゾーン文字列の例を次に示します。

- AST (大西洋標準時)
- ADT (大西洋夏時間)
- BST (ベーリング標準時)
- BDT (ベーリング夏時間)

この関数は、無効な日付またはタイム・ゾーンが渡された場合はエラーを戻します。

関連関数

[OCIErrorGet\(\)](#)、[OCIDateCheck\(\)](#)

OCIIntervalAdd()

用途

2つの時間隔を加算して、結果の時間隔を生成します。

構文

```
sword OCIIntervalAdd ( dvoid          *hndl,  
                      OCIError      *err,  
                      OCIInterval   *addend1,  
                      OCIInterval   *addend2,  
                      OCIInterval   *result );
```

パラメータ

hndl (IN)

OCI ユーザー・セッション・ハンドルまたは環境ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet () をコールして診断情報を取得します。

addend1 (IN)

加算する時間隔です。

addend2 (IN)

加算する時間隔です。

result (OUT)

加算結果の時間隔 (`addend1 + addend2`) です。

戻り値

`OCI_SUCCESS`

`OCI_ERROR` (次のいずれかの場合)

- 入力された 2 つの時間隔が相互に比較可能でない場合
- 結果の年が `SB4MAXVAL` を超える場合
- 結果の年が `SB4MINVAL` を下回る場合

`OCI_INVALID_HANDLE` (`err` が `NULL` ポインタの場合)

関連関数

[OCIIntervalSubtract\(\)](#)

OCIIntervalAssign()

用途

ある時間隔を別の時間隔にコピーします。

構文

```
void OCIIntervalAssign ( dvoid          *hdl,  
                        OCIError       *err,  
                        CONST OCIInterval *inpinter,  
                        OCIInterval    *outinter );
```

パラメータ

hdl (IN)

OCI ユーザー・セッション・ハンドルまたは環境ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。 `OCIErrorGet()` をコールして診断情報を取得します。

inpinter (IN)

入力時間隔です。

outinter (OUT)

出力時間隔です。

戻り値

`OCI_SUCCESS`

`OCI_INVALID_HANDLE` (`err` が `NULL` ポインタの場合)

関連関数

[OCIIntervalCompare\(\)](#)

OCIIntervalCheck()

用途

時間隔の妥当性をチェックします。

構文

```
sword OCIIntervalCheck ( dvoid          *hdl,  
                        OCIError       *err,  
                        CONST OCIInterval *interval,  
                        ub4             *valid );
```

パラメータ

hdl (IN)

OCI ユーザー・セッション・ハンドルまたは環境ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。`OCIErrorGet()` をコールして診断情報を取得します。

interval (IN)

チェックする時間隔です。

valid (OUT)

時間隔が有効な場合は 0 (ゼロ) を戻します。無効な場合は、次のコードの **OR** による組合せを戻します。

マクロ名	ビット数	エラー
OCI_INTER_INVALID_DAY	0x1	無効な日。
OCI_INTER_DAY_BELOW_VALID	0x2	無効な日の、有効な日に対する大 / 小を示すビット (1= 小)。
OCI_INTER_INVALID_MONTH	0x4	無効な月。
OCI_INTER_MONTH_BELOW_VALID	0x8	無効な月の、有効な月に対する大 / 小を示すビット (1= 小)。
OCI_INTER_INVALID_YEAR	0x10	無効な年。
OCI_INTER_YEAR_BELOW_VALID	0x20	無効な年の、有効な年に対する大 / 小を示すビット (1= 小)。
OCI_INTER_INVALID_HOUR	0x40	無効な時刻 (時)。

マクロ名	ビット数	エラー
OCI_INTER_HOUR_BELOW_VALID	0x80	無効な時刻（時）の、有効な時刻（時）に対する大 / 小を示すビット（1= 小）。
OCI_INTER_INVALID_MINUTE	0x100	無効な時刻（分）。
OCI_INTER_MINUTE_BELOW_VALID	0x200	無効な時刻（分）の、有効な時刻（分）に対する大 / 小を示すビット（1= 小）。
OCI_INTER_INVALID_SECOND	0x400	無効な時刻（秒）。
OCI_INTER_SECOND_BELOW_VALID	0x800	無効な時刻（秒）の、有効な時刻（秒）に対する大 / 小を示すビット（1= 小）。
OCI_INTER_INVALID_FRACSEC	0x1000	無効な小数秒。
OCI_INTER_FRACSEC_BELOW_VALID	0x2000	無効な時刻（小数秒）の、有効な時刻（小数秒）に対する大 / 小を示すビット（1= 小）

戻り値

OCI_SUCCESS

OCI_INVALID_HANDLE（`err` が NULL ポインタの場合）

OCI_ERROR（エラーの場合）

関連関数

[OCIIntervalCompare\(\)](#)

OCIIntervalCompare()

用途

2つの時間隔を比較します。

構文

```
sword OCIIntervalCompare( dvoid          *hndl,
                          OCIError       *err,
                          OCIInterval    *inter1,
                          OCIInterval    *inter2,
                          sword          *result );
```

パラメータ

hndl (IN)
OCI ユーザー・セッション・ハンドルまたは環境ハンドルです。

err (IN/OUT)
OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。 `OCIErrorGet()` をコールして診断情報を取得します。

inter1 (IN)
比較する時間隔です。

inter2 (IN)
比較する時間隔です。

result (OUT)
比較結果は次のとおりです。

比較結果	<i>result</i> パラメータの出力
<code>inter1 < inter2</code>	-1
<code>inter1 = inter2</code>	0
<code>inter1 > inter2</code>	1

戻り値

- `OCI_SUCCESS`
- `OCI_INVALID_HANDLE` (`err` が `NULL` ポインタの場合)
- `OCI_ERROR` (入力値が相互に比較可能でない場合)

関連関数

[OCIIntervalAssign\(\)](#)

OCIIntervalDivide()

用途

時間隔を Oracle 数値で除算して、その結果の時間隔を生成します。

構文

```
sword OCIIntervalDivide ( dvoid          *hdl,  
                          OCIError      *err,  
                          OCIInterval    *dividend,  
                          OCINumber     *divisor,  
                          OCIInterval    *result );
```

パラメータ

hdl (IN)

OCI ユーザー・セッション・ハンドルまたは環境ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet () をコールして診断情報を取得します。

dividend (IN)

除算する時間隔です。

divisor (IN)

dividend を除算する Oracle 数値です。

result (OUT)

除算結果の時間隔 (*dividend* / *divisor*) です。

戻り値

OCI_SUCCESS

OCI_INVALID_HANDLE (`err` が NULL ポインタの場合)

関連関数

[OCIIntervalMultiply\(\)](#)

OCIIntervalFromNumber()

用途

Oracle 数値を時間隔に変換します。

構文

```
sword OCIIntervalFromNumber ( dvoid          *hndl,  
                             OCIError       *err,  
                             OCIInterval    *interval,  
                             OCINumber     *number );
```

パラメータ

hndl (IN)

OCI ユーザー・セッション・ハンドルまたは環境ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、*err* に記録され、OCI_ERROR が戻されます。OCIErrorGet () をコールして診断情報を取得します。

interval (OUT)

結果の時間隔です。

number (IN)

変換する Oracle 数値 (YEAR TO MONTH 時間隔の場合は年数、DAY TO SECOND 時間隔の場合は日数) です。

戻り値

OCI_SUCCESS

OCI_INVALID_HANDLE (*err* が NULL ポインタの場合)

関連関数

[OCIIntervalToNumber\(\)](#)

OCIIntervalFromText()

用途

時間隔文字列が指定されている場合、その文字列で表現される時間隔を返します。時間隔の型は、*result* 記述子の型です。

構文

```
sword OCIIntervalFromText ( dvoid          *hdl,  
                           OCIError       *err,  
                           CONST OraText  *inpstring,  
                           size_t         str_len,  
                           OCIInterval    *result );
```

パラメータ

hdl (IN)

OCI ユーザー・セッション・ハンドルまたは環境ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、*err* に記録され、OCI_ERROR が戻されます。OCIErrorGet () をコールして診断情報を取得します。

inpstring (IN)

入力文字列です。

str_len (IN)

入力文字列の長さです。

result (OUT)

結果の時間隔です。

戻り値

OCI_SUCCESS

OCI_INVALID_HANDLE (*err* が NULL ポインタの場合)

OCI_ERROR (次のいずれかの場合)

- リテラル文字列のフィールドが多すぎる場合
- 年が範囲 (-4713 ~ 9999) 外の場合
- 月が範囲 (1 ~ 12) 外の場合
- 日が範囲 (1 ~ 28...31) 外の場合
- 時間が範囲 (0 ~ 23) 外の場合

- 時間が範囲（0 ～ 11）外の場合
- 分が範囲（0 ～ 59）外の場合
- 1 分の秒が範囲（0 ～ 59）外の場合
- 1 日の秒が範囲（0 ～ 86399）外の場合
- 時間隔が無効な場合

関連関数

[OCIIntervalToText\(\)](#)

OCIIntervalFromTZ()

用途

OCIInterval データ型の **OCI_DTYPE_INTERVAL_DS** を、設定されているリージョン ID（入力文字列にリージョンが指定されている場合）と現行の絶対オフセット、またはリージョン ID が 0（ゼロ）に設定されている絶対オフセットとともに戻します。

構文

```
sword OCIIntervalFromTZ ( dvoid          *hndl,  
                          OCIError       *err,  
                          CONST oratext   *inpstring,  
                          size_t          str_len,  
                          OCIInterval     *result ) ;
```

パラメータ

hndl (IN)

OCI ユーザー・セッション・ハンドルまたは環境ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、**OCI_ERROR** が戻されます。OCIErrorGet() をコールして診断情報を取得します。

inpstring (IN)

入力文字列へのポインタです。

str_len (IN)

inpstring の長さです。

result (OUT)

出力時間隔です。

戻り値

OCI_SUCCESS（成功した場合）

OCI_INVALID_HANDLE（**err** が **NULL** の場合）

OCI_ERROR（無効な時間隔型、またはタイム・ゾーン・エラーがある場合）

コメント

入力文字列は、**[+/-]TZH:TZM** または **'TZR [TZD]'** の形式である必要があります。

関連関数

[OCIIntervalFromText\(\)](#)

OCIIntervalGetDaySecond()

用途

時間隔から日、時、分、秒の値を取得します。

構文

```
sword OCIIntervalGetDaySecond (dvoid          *hdl,  
                                OCIError       *err,  
                                sb4            *dy,  
                                sb4            *hr,  
                                sb4            *mm,  
                                sb4            *ss,  
                                sb4            *fsec,  
                                CONST OCIInterval *interval );
```

パラメータ

hdl (IN)

OCI ユーザー・セッション・ハンドルまたは環境ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet () をコールして診断情報を取得します。

dy (OUT)

日数です。

hr (OUT)

時間数です。

mm (OUT)

分数です。

ss (OUT)

秒数です。

fsec (OUT)

小数秒の数です。

interval (IN)

入力時間隔です。

戻り値

OCI_SUCCESS

OCI_INVALID_HANDLE (*err* が NULL ポインタの場合)

関連関数

[OCIIntervalSetDaySecond\(\)](#)

OCIIntervalGetYearMonth()

用途

時間隔から年と月を取得します。

構文

```
sword OCIIntervalGetYearMonth ( dvoid          *hndl,  
                                OCIError       *err,  
                                sb4            *yr,  
                                sb4            *mnth,  
                                CONST OCIInterval *interval );
```

パラメータ

hndl (IN)

OCI ユーザー・セッション・ハンドルまたは環境ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet () をコールして診断情報を取得します。

yr (OUT)

年の値です。

mnth (OUT)

月の値です。

interval (IN)

入力時間隔です。

戻り値

OCI_SUCCESS

OCI_INVALID_HANDLE (err が NULL ポインタの場合)

関連関数

[OCIIntervalSetYearMonth\(\)](#)

OCIIntervalMultiply()

用途

時間隔を Oracle 数値で乗算して、その結果の時間隔を生成します。

構文

```
sword OCIIntervalMultiply ( dvoid          *hdl,  
                           OCIError       *err,  
                           CONST OCIInterval *inter,  
                           OCINumber      *nfactor,  
                           OCIInterval     *result );
```

パラメータ

hdl (IN)

OCI ユーザー・セッション・ハンドルまたは環境ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。 `OCIErrorGet()` をコールして診断情報を取得します。

inter (IN)

乗算する時間隔です。

nfactor (IN)

乗算する Oracle 数値です。

result (OUT)

乗算結果の時間隔 ($inter * nfactor$) です。

戻り値

`OCI_SUCCESS`

`OCI_INVALID_HANDLE` (`err` が NULL ポインタの場合)

`OCI_ERROR` (次のいずれかの場合)

- 結果の年が `SB4MAXVAL` を超える場合
- 結果の年が `SB4MINVAL` を下回る場合

関連関数

[OCIIntervalDivide\(\)](#)

OCIIntervalSetDaySecond()

用途

時間隔に日、時、分、秒を設定します。

構文

```
sword OCIIntervalSetDaySecond ( dvoid          *hndl,  
                                OCIError       *err,  
                                sb4            dy,  
                                sb4            hr,  
                                sb4            mm,  
                                sb4            ss,  
                                sb4            fsec,  
                                OCIInterval    *result );
```

パラメータ

hndl (IN)

OCI ユーザー・セッション・ハンドルまたは環境ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。 `OCIErrorGet()` をコールして診断情報を取得します。

dy (IN)

日数です。

hr (IN)

時間数です。

mm (IN)

分数です。

ss (IN)

秒数です。

fsec (IN)

小数秒の数です。

result (OUT)

結果の時間隔です。

戻り値

OCI_SUCCESS

OCI_INVALID_HANDLE (*err* が NULL ポインタの場合)

関連関数

[OCIIntervalGetDaySecond\(\)](#)

OCIIntervalSetYearMonth()

用途

時間隔に年と月を設定します。

構文

```
sword OCIIntervalSetYearMonth ( dvoid          *hndl,  
                                OCIError       *err,  
                                sb4            yr,  
                                sb4            mnth,  
                                OCIInterval    *result );
```

パラメータ

hndl (IN)

OCI ユーザー・セッション・ハンドルまたは環境ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet () をコールして診断情報を取得します。

yr (IN)

年の値です。

mnth (IN)

月の値です。

result (OUT)

結果の時間隔です。

戻り値

OCI_SUCCESS

OCI_INVALID_HANDLE (err が NULL ポインタの場合)

関連関数

- 結果の年が SB4MAXVAL を超える場合
- 結果の年が SB4MINVAL を下回る場合

関連関数

[OCIIntervalGetYearMonth\(\)](#)

OCIIntervalSubtract()

用途

2 つの時間隔を減算し、その結果を時間隔に格納します。

構文

```
sword OCIIntervalSubtract ( dvoid          *hndl,  
                             OCIError      *err,  
                             OCIInterval   *minuend,  
                             OCIInterval   *subtrahend,  
                             OCIInterval   *result );
```

パラメータ

hndl (IN)

OCI ユーザー・セッション・ハンドルまたは環境ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。 `OCIErrorGet()` をコールして診断情報を取得します。

minuend (IN)

減算される時間隔です。

subtrahend (IN)

minuend から減算する時間隔です。

result (OUT)

減算結果の時間隔 (*minuend* - *subtrahend*) です。

戻り値

`OCI_SUCCESS`

`OCI_INVALID_HANDLE` (`err` が `NULL` ポインタの場合)

`OCI_ERROR` (次のいずれかの場合)

- 結果の年が `SB4MAXVAL` を超える場合
- 結果の年が `SB4MINVAL` を下回る場合
- 入力された 2 つの時間隔が相互に比較可能でない場合

関連関数

[OCIIntervalAdd\(\)](#)

OCIIntervalToNumber()

用途

時間隔を Oracle 数値に変換します。

構文

```
sword OCIIntervalToNumber ( dvoid          *hdl,  
                             OCIError      *err,  
                             OCIInterval    *interval,  
                             OCINumber     *number );
```

パラメータ

hdl (IN)

OCI ユーザー・セッション・ハンドルまたは環境ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。`OCIErrorGet()` をコールして診断情報を取得します。

interval (IN)

変換する時間隔です。

number (OUT)

Oracle 数値の変換結果（`YEARMONTH` 時間隔の年、および `DAYSECOND` の日）です。

コメント

変換された Oracle 数値には、日付の小数部分（たとえば、選択した単位が時間の場合は分と秒）が含まれます。精度を超える部分は切り捨てられます。

戻り値

`OCI_SUCCESS`

`OCI_INVALID_HANDLE` (`err` が `NULL` ポインタの場合)

関連関数

[OCIIntervalFromNumber\(\)](#)

OCIIntervalToText()

用途

時間隔が指定されている場合、その時間隔を表現する文字列を生成します。

構文

```
sword OCIIntervalToText ( dvoid          *hdl,  
                          OCIError       *err,  
                          CONST OCIInterval *interval,  
                          ub1            lfprec,  
                          ub1            fsprec,  
                          OraText        *buffer,  
                          size_t         buflen,  
                          size_t         *resultlen );
```

パラメータ

hdl (IN)

OCI ユーザー・セッション・ハンドルまたは環境ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。 `OCIErrorGet()` をコールして診断情報を取得します。

interval (IN)

変換する時間隔です。

lfprec (IN)

先行フィールド精度です（先行フィールドを表すのに使用される桁数です）。

fsprec (IN)

時間隔の小数秒精度（小数秒を表すのに使用される桁数）です。

buffer (OUT)

結果を保持するバッファです。

buflen (IN)

buffer の長さです。

resultlen (OUT)

buffer に格納された結果の長さです。

コメント

時間隔リテラルは、INTERVAL YEAR TO MONTH 時間隔の '年' または '[年 -] 月 '、および INTERVAL DAY TO SECOND 時間隔の '秒'、'分 [: 秒]'、'時間 [: 分 [: 秒]]' または '日 [時間 [: 分 [: 秒]]]' として出力されます（大カッコで囲まれているフィールドはオプションです）。

戻り値

OCI_SUCCESS

OCI_INVALID_HANDLE (*err* が NULL ポインタの場合)

OCI_ERROR (バッファ領域が不足しているため結果を保持できない場合)

関連関数

[OCIIntervalFromText\(\)](#)

OCI 数値関数

この項では、OCI 数値関数について説明します。

表 18-4 数値関数

関数	用途
OCINumberAbs() (18-97 ページ)	絶対値を計算します。
OCINumberAdd() (18-98 ページ)	数値を加算します。
OCINumberArcCos() (18-99 ページ)	アーク・コサインを計算します。
OCINumberArcSin() (18-100 ページ)	アーク・サインを計算します。
OCINumberArcTan() (18-101 ページ)	アーク・タンジェントを計算します。
OCINumberArcTan2() (18-102 ページ)	2 つの数値のアーク・タンジェントを計算します。
OCINumberAssign() (18-103 ページ)	ある数値を別の数値に代入します。
OCINumberCeil() (18-104 ページ)	数値の上限を計算します。
OCINumberCmp() (18-105 ページ)	数値を比較します。
OCINumberCos() (18-106 ページ)	コサインを計算します。
OCINumberDec() (18-107 ページ)	OCI 数値を減分します。
OCINumberDiv() (18-108 ページ)	2 つの数を除算します。
OCINumberExp() (18-109 ページ)	e を指定の Oracle 数値で累乗します。
OCINumberFloor() (18-110 ページ)	数値の下限を計算します。
OCINumberFromInt() (18-111 ページ)	整数を Oracle 数値に変換します。
OCINumberFromReal() (18-112 ページ)	実数を Oracle 数値に変換します。
OCINumberFromText() (18-113 ページ)	文字列を Oracle 数値に変換します。
OCINumberHypCos() (18-115 ページ)	双曲線コサインを計算します。
OCINumberHypSin() (18-116 ページ)	双曲線サインを計算します。
OCINumberHypTan() (18-117 ページ)	双曲線タンジェントを計算します。
OCINumberInc() (18-118 ページ)	Oracle 数値を増分します。
OCINumberIntPower() (18-119 ページ)	指定の底を整数で累乗します。
OCINumberIsInt() (18-120 ページ)	数値が整数かどうかをテストします。
OCINumberIsZero() (18-121 ページ)	数値が 0 (ゼロ) かどうかをテストします。

表 18-4 数値関数（続き）

関数	用途
OCINumberLn() (18-122 ページ)	自然対数を計算します。
OCINumberLog() (18-123 ページ)	任意の底に対する対数を計算します。
OCINumberMod() (18-124 ページ)	除算の剰余を取得します。
OCINumberMul() (18-125 ページ)	数値を乗算します。
OCINumberNeg() (18-126 ページ)	数値を負の数にします。
OCINumberPower() (18-127 ページ)	底 <i>e</i> を累乗します。
OCINumberPrec() (18-128 ページ)	数値を指定の小数点以下の桁数に丸めます。
OCINumberRound() (18-129 ページ)	Oracle 数値を指定の小数点以下の桁数に丸めます。
OCINumberSetPi() (18-130 ページ)	数値を Pi に初期化します。
OCINumberSetZero() (18-131 ページ)	数値を 0（ゼロ）に初期化します。
OCINumberShift() (18-132 ページ)	10 の累乗を乗算し、小数点以下を指定の桁数にします。
OCINumberSign() (18-133 ページ)	Oracle 数値の符号を取得します。
OCINumberSin() (18-134 ページ)	サインを計算します。
OCINumberSqrt() (18-135 ページ)	数値の平方根を計算します。
OCINumberSub() (18-136 ページ)	数値を減算します。
OCINumberTan() (18-137 ページ)	タンジェントを計算します。
OCINumberToInt() (18-138 ページ)	Oracle 数値を整数に変換します。
OCINumberToReal() (18-139 ページ)	Oracle 数値を実数に変換します。
OCINumberToText() (18-140 ページ)	Oracle 数値を文字列に変換します。
OCINumberTrunc() (18-142 ページ)	Oracle 数値を指定の小数点以下の桁数で切り捨てます。

OCINumberAbs()

用途

Oracle 数値の絶対値を計算します。

構文

```
sword OCINumberAbs ( OCLError          *err,  
                     CONST OCINumber    *number,  
                     OCINumber          *result );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、OCI_ERROR が戻されます。OCLErrorGet () をコールして診断情報を取得します。

number (IN)

入力数値です。

result (OUT)

入力数値の絶対値です。

コメント

この関数は、NULL の数値引数がある場合はエラーを戻します。

関連関数

[OCLErrorGet \(\)](#)、[OCINumberSign \(\)](#)

OCINumberAdd()

用途

2 つの Oracle 数値を加算します。

構文

```
sword OCINumberAdd ( OCLError          *err,  
                     CONST OCINumber    *number1,  
                     CONST OCINumber    *number2,  
                     OCINumber          *result );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet () をコールして診断情報を取得します。

number1、number2 (IN)

加算される数値です。

result (OUT)

`number1` を `number2` に加算した結果です。

コメント

この関数は、NULL の数値引数がある場合はエラーを戻します。

関連関数

[OCIErrorGet \(\)](#)、[OCINumberSub \(\)](#)

OCINumberArcCos()

用途

Oracle 数値のアーク・コサインをラジアン単位で求めます。

構文

```
sword OCINumberArcCos ( OCLError          *err,  
                        CONST OCINumber    *number,  
                        OCINumber          *result );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、OCI_ERROR が戻されます。OCLErrorGet () をコールして診断情報を取得します。

number (IN)

アーク・コサインの引数です。

result (OUT)

ラジアンのアーク・コサインの結果です。

コメント

この関数は、NULL の数値引数がある場合、または *number* が -1 より小さいか *number* が 1 より大きい場合はエラーを戻します。

関連関数

[OCLErrorGet \(\)](#)、[OCINumberCos \(\)](#)

OCINumberArcSin()

用途

Oracle 数値のアーク・サインをラジアン単位で求めます。

構文

```
sword OCINumberArcSin ( OCLError          *err,  
                        CONST OCINumber    *number,  
                        OCINumber          *result );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。 `OCLErrorGet()` をコールして診断情報を取得します。

number (IN)

アーク・サインの引数です。

result (OUT)

ラジアンのアーク・サインの結果です。

コメント

この関数は、`NULL` の数値引数がある場合、または `number` が -1 より小さいか `number` が 1 より大きい場合はエラーを戻します。

関連関数

[OCLErrorGet\(\)](#)、[OCINumberSin\(\)](#)

OCINumberArcTan()

用途

Oracle 数値のアーク・タンジェントをラジアン単位で求めます。

構文

```
sword OCINumberArcTan ( OCLError          *err,  
                        CONST OCINumber    *number,  
                        OCINumber          *result );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、OCI_ERROR が戻されます。OCLErrorGet () をコールして診断情報を取得します。

number (IN)

アーク・タンジェントの引数です。

result (OUT)

ラジアンのアーク・タンジェントの結果です。

コメント

この関数は、NULL の数値引数がある場合はエラーを戻します。

関連関数

[OCLErrorGet \(\)](#)、[OCINumberTan \(\)](#)

OCINumberArcTan2()

用途

2 つの Oracle 数値のアーク・タンジェントを求めます。

構文

```
sword OCINumberArcTan2 ( OCLError          *err,
                          CONST OCINumber    *number1,
                          CONST OCINumber    *number2,
                          OCINumber          *result );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。 `OCLErrorGet()` をコールして診断情報を取得します。

number1 (IN)

アーク・タンジェントの引数 1 です。

number2 (IN)

アーク・タンジェントの引数 2 です。

result (OUT)

ラジアンのアーク・タンジェントの結果です。

コメント

この関数は、NULL の数値引数がある場合、または `number2` が 0 (ゼロ) の場合はエラーを戻します。

関連関数

[OCLErrorGet\(\)](#)、[OCINumberTan\(\)](#)

OCINumberAssign()

用途

ある Oracle 数値を別の Oracle 数値に割り当てます。

構文

```
sword OCINumberAssign ( OCLError          *err,  
                        CONST OCINumber    *from,  
                        OCINumber          *to );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、OCI_ERROR が戻されます。OCLErrorGet () をコールして診断情報を取得します。

from (IN)

割り当てる数値です。

to (OUT)

コピー先の数値です。

コメント

from によって識別される数値を、*to* によって識別される数値に代入します。

この関数は、NULL の数値引数がある場合はエラーを戻します。

関連関数

[OCLErrorGet \(\)](#)、[OCINumberCmp \(\)](#)

OCINumberCeil()

用途

Oracle 数値の上限値を計算します。

構文

```
sword OCINumberCeil ( OCIError          *err,
                     CONST OCINumber    *number,
                     OCINumber          *result );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet () をコールして診断情報を取得します。

number (IN)

入力数値です。

result (OUT)

入力数値の上限値を含む出力です。

コメント

この関数は、NULL の数値引数がある場合はエラーを戻します。

関連関数

[OCIErrorGet \(\)](#)、[OCINumberFloor \(\)](#)

OCINumberCmp()

用途

2 つの Oracle 数値を比較します。

構文

```
sword OCINumberCmp ( OCIError          *err,
                     CONST OCINumber    *number1,
                     CONST OCINumber    *number2,
                     sword                *result );
```

パラメータ

- err (IN/OUT)**
OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。 `OCIErrorGet()` をコールして診断情報を取得します。
- number1、number2 (IN)**
比較する数値です。
- result (OUT)**
比較結果は次のとおりです。

比較結果	result パラメータの出力
number1 < number2	負数
number1 = number2	0
number1 > number2	正数

コメント

この関数は、NULL の数値引数がある場合はエラーを戻します。

関連関数

`OCIErrorGet()`、`OCINumberAssign()`

OCINumberCos()

用途

Oracle 数値のコサインをラジアン単位で計算します。

構文

```
sword OCINumberCos ( OCLError          *err,
                     CONST OCINumber    *number,
                     OCINumber          *result );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet () をコールして診断情報を取得します。

number (IN)

ラジアンのコサインの引数です。

result (OUT)

コサインの結果です。

コメント

この関数は、NULL の数値引数がある場合はエラーを戻します。

関連関数

[OCIErrorGet \(\)](#)、[OCINumberArcCos \(\)](#)

OCINumberDec()

用途

OCINumber を減分します。

構文

```
sword OCINumberDec ( OCLError *err,  
                    OCINumber *number );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet () をコールして診断情報を取得します。

number (IN/OUT)

減分される正数の Oracle 数値です。

コメント

Oracle 数値を適切に減分します。入力値は $0 \sim 100^{21-2}$ の整数であるとみなされます。入力値が大きすぎる場合は 0（ゼロ）として扱われ、その結果、Oracle 数値は 1 になります。入力値が正の整数でない場合は、予想できない結果になります。

この関数は、入力数値が NULL の場合はエラーを戻します。

関連関数

[OCINumberInc\(\)](#)

OCINumberDiv()

用途

2 つの Oracle 数値を除算します。

構文

```
sword OCINumberDiv ( OCLError          *err,
                     CONST OCINumber    *number1,
                     CONST OCINumber    *number2,
                     OCINumber          *result );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet () をコールして診断情報を取得します。

number1 (IN)

分子へのポインタです。

number2 (IN)

分母へのポインタです。

result (OUT)

除算結果です。

コメント

`number1` を `number2` で除算し、その結果を `result` に戻します。

この関数は、次の場合にエラーを戻します。

- NULL の数値引数がある場合
- アンダーフロー・エラーがある場合
- ゼロによる除算エラーがある場合

関連関数

[OCIErrorGet \(\)](#)、[OCINumberMul \(\)](#)

OCINumberExp()

用途

e を指定の Oracle 数値で累乗します。

構文

```
sword OCINumberExp ( OCIError          *err,  
                     CONST OCINumber    *number,  
                     OCINumber          *result );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、OCI_ERROR が戻されます。OCIErrorGet () をコールして診断情報を取得します。

number (IN)

この関数は、 e をこの Oracle 数値で累乗します。

result (OUT)

指数の出力です。

コメント

この関数は、NULL の数値引数がある場合はエラーを戻します。

関連関数

[OCIErrorGet \(\)](#)、[OCINumberLn \(\)](#)

OCINumberFloor()

用途

Oracle 数値の下限値を計算します。

構文

```
sword OCINumberFloor ( OCIError          *err,
                       CONST OCINumber    *number,
                       OCINumber          *result );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、OCI_ERROR が戻されます。OCIErrorGet () をコールして診断情報を取得します。

number (IN)

入力数値です。

result (OUT)

入力数値の下限値です。

コメント

この関数は、NULL の数値引数がある場合はエラーを戻します。

関連関数

[OCIErrorGet \(\)](#)、[OCINumberCeil \(\)](#)

OCINumberFromInt()

用途

整数を Oracle 数値に変換します。

構文

```
sword OCINumberFromInt ( OCLError          *err,  
                        CONST dvoid        *inum,  
                        uword              inum_length,  
                        uword              inum_s_flag,  
                        OCINumber          *number );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、*err* に記録され、OCI_ERROR が戻されます。OCLErrorGet () をコールして診断情報を取得します。

inum (IN)

変換する整数へのポインタです。

inum_length (IN)

整数のサイズです。

inum_s_flag (IN)

整数の符号を指定するフラグで、可能な値は次のとおりです。

- OCI_NUMBER_UNSIGNED — 符号なしの値
- OCI_NUMBER_SIGNED — 符号付きの値

number (OUT)

Oracle 数値に変換される指定の整数です。

コメント

これは、システム固有な型変換関数です。**ub4** や **sb2** などの Oracle 標準マシンネイティブ整数型を Oracle 数値に変換します。

この関数は、数値が大きすぎて Oracle 数値に適合しない場合、*number* または *inum* が NULL の場合、または *inum_s_flag* に無効な符号フラグの値が渡された場合は、エラーを戻します。

関連関数

[OCLErrorGet \(\)](#)、[OCINumberToInt \(\)](#)

OCINumberFromReal()

用途

実数（浮動小数点数）型を Oracle 数値に変換します。

構文

```
sword OCINumberFromReal ( OCIError          *err,
                          CONST dvoid        *rnum,
                          uword              rnum_length,
                          OCINumber          *number );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、OCI_ERROR が戻されます。OCIErrorGet () をコールして診断情報を取得します。

rnum (IN)

変換する浮動小数点数へのポインタです。

rnum_length (IN)

目的の結果のサイズです。sizeof({float | double | long double}) に等しくなります。

number (OUT)

Oracle 数値に変換される指定の float です。

コメント

これは、システム固有な型変換関数です。マシンネイティブ浮動小数点型を Oracle 数値に変換します。

この関数は、**number** または **rnum** が NULL であるか、**rnum_length** が 0（ゼロ）である場合はエラーを戻します。

関連関数

[OCIErrorGet \(\)](#)、[OCINumberToReal \(\)](#)

OCINumberFromText()

用途

文字列を Oracle 数値に変換します

構文

```
sword OCINumberFromText ( OCIError          *err,
                          CONST OraText      *str,
                          ub4                 str_length,
                          CONST OraText      *fmt,
                          ub4                 fmt_length,
                          CONST OraText      *nls_params,
                          ub4                 nls_p_length,
                          OCINumber          *number );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、*err* に記録され、OCI_ERROR が戻されます。OCIErrorGet () をコールして診断情報を取得します。

str (IN)

Oracle 数値に変換する入力文字列です。

str_length (IN)

入力文字列のサイズです。

fmt (IN)

変換書式です。

fmt_length (IN)

fmt パラメータの長さです。

nls_params (IN)

グローバル・サポートのフォーマット指定です。ヌル文字列 ("") の場合は、セッションのデフォルト・パラメータが使用されます。

nls_p_length (IN)

nls_params パラメータの長さです。

number (OUT)

数値に変換された指定文字列です。

コメント

指定された文字列を指定の書式の数値に変換します。書式および多言語パラメータの詳細は、『Oracle9i SQL リファレンス』の TO_NUMBER 変換関数の説明を参照してください。

この関数は、無効な書式、マルチバイト書式または入力文字列がある場合、*number* または *str* が NULL である場合、あるいは *str_length* が 0（ゼロ）である場合はエラーを返します。

関連関数

[OCIErrorGet\(\)](#)、[OCINumberToText\(\)](#)

OCINumberHypCos()

用途

Oracle 数値の双曲線コサインを計算します。

構文

```
sword OCINumberHypCos ( OCLError          *err,  
                        CONST OCINumber    *number,  
                        OCINumber          *result );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、OCI_ERROR が戻されます。OCLErrorGet () をコールして診断情報を取得します。

number (IN)

コサイン双曲線の引数です。

result (OUT)

コサイン双曲線の結果です。

コメント

この関数は、NULL の数値引数がある場合はエラーを戻します。

注意： Oracle 数値のオーバーフローは、予想できない結果値が出力される原因となります。

関連関数

[OCLErrorGet \(\)](#)、[OCINumberHypSin \(\)](#)、[OCINumberHypTan \(\)](#)

OCINumberHypSin()

用途

Oracle 数値の双曲線サインを計算します。

構文

```
sword OCINumberHypSin ( OCLError          *err,
                        CONST OCINumber    *number,
                        OCINumber          *result );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。 `OCLErrorGet()` をコールして診断情報を取得します。

number (IN)

サイン双曲線の引数です。

result (OUT)

サイン双曲線の結果です。

コメント

この関数は、`NULL` の数値引数がある場合はエラーを戻します。

注意： Oracle 数値のオーバーフローは、予想できない結果値が出力される原因となります。

関連関数

[OCLErrorGet\(\)](#)、[OCINumberHypCos\(\)](#)、[OCINumberHypTan\(\)](#)

OCINumberHypTan()

用途

Oracle 数値の双曲線タンジェントを計算します。

構文

```
sword OCINumberHypTan ( OCLError          *err,  
                        CONST OCINumber    *number,  
                        OCINumber          *result );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。 `OCLErrorGet()` をコールして診断情報を取得します。

number (IN)

タンジェント双曲線の引数です。

result (OUT)

タンジェント双曲線の結果です。

コメント

この関数は、NULL の数値引数がある場合はエラーを戻します。

注意： Oracle 数値のオーバーフローは、予想できない結果値が出力される原因となります。

関連関数

[OCLErrorGet\(\)](#)、[OCINumberHypCos\(\)](#)、[OCINumberHypSin\(\)](#)

OCINumberInc()

用途

OCINumber を増分します。

構文

```
sword OCINumberInc ( OCIError   *err,  
                    OCINumber  *number );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet () をコールして診断情報を取得します。

number (IN/OUT)

増分される正数の Oracle 数値です。

コメント

Oracle 数値を適切に増分します。入力値は $0 \sim 100^{21-2}$ の整数であるとみなされます。入力値が大きすぎる場合は 0 (ゼロ) として扱われ、その結果、Oracle 数値は 1 になります。入力値が正の整数でない場合は、予想できない結果になります。

この関数は、入力数値が NULL の場合はエラーを戻します。

関連関数

[OCINumberDec\(\)](#)

OCINumberIntPower()

用途

指定の基数を指定の整数で累乗します。

構文

```
sword OCINumberIntPower ( OCIError          *err,  
                           CONST OCINumber    *base,  
                           CONST sword        exp,  
                           OCINumber          *result );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。 `OCIErrorGet()` をコールして診断情報を取得します。

base (IN)

指数の基数です。

exp (IN)

基数が累乗される指数です。

result (OUT)

指数の出力です。

コメント

この関数は、`NULL` の数値引数がある場合はエラーを戻します。

関連関数

[OCIErrorGet\(\)](#)、[OCINumberPower\(\)](#)

OCINumberIsInt()

用途

OCINumber が整数かどうかをテストします。

構文

```
sword OCINumberIsInt ( OCLError      *err,  
                       CONST OCINumber *number,  
                       boolean        *result );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCLErrorGet () をコールして診断情報を取得します。

number (IN)

テストされる数値です。

result (OUT)

整数値の場合は TRUE、それ以外の場合は FALSE に設定されます。

コメント

この関数は、`number` または `result` が NULL の場合にエラーを戻します。

関連関数

[OCLErrorGet \(\)](#)、[OCINumberRound \(\)](#)、[OCINumberTrunc \(\)](#)

OCINumberIsZero()

用途

指定の数値が 0（ゼロ）に等しいかどうかをテストします。

構文

```
sword OCINumberIsZero ( OCLError          *err,  
                        CONST OCINumber    *number,  
                        boolean            *result );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、OCI_ERROR が戻されます。OCLErrorGet () をコールして診断情報を取得します。

number (IN)

比較する数値です。

result (OUT)

0（ゼロ）の場合は TRUE、それ以外の場合は FALSE が設定されます。

コメント

この関数は、NULL の数値引数がある場合はエラーを戻します。

関連関数

[OCLErrorGet \(\)](#)、[OCINumberSetZero \(\)](#)

OCINumberLn()

用途

Oracle 数値の自然対数（底 e ）を求めます。

構文

```
sword OCINumberLn ( OCLError          *err,
                    CONST OCINumber    *number,
                    OCINumber          *result );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCLErrorGet () をコールして診断情報を取得します。

number (IN)

この数値の対数を計算します。

result (OUT)

対数結果です。

コメント

この関数は、NULL の数値引数がある場合、または `number` が 0（ゼロ）以下の場合はエラーを戻します。

関連関数

[OCLErrorGet \(\)](#)、[OCINumberExp \(\)](#)、[OCINumberLog \(\)](#)

OCINumberLog()

用途

Oracle 数値の任意の基数に対する対数を求めます。

構文

```
sword OCINumberLog ( OCLError          *err,  
                     CONST OCINumber    *base,  
                     CONST OCINumber    *number,  
                     OCINumber          *result );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。 `OCLErrorGet()` をコールして診断情報を取得します。

base (IN)

対数の基数です。

number (IN)

演算子です。

result (OUT)

対数結果です。

コメント

この関数は、次の場合にエラーを戻します。

- `NULL` の数値引数がある場合
- `number <= 0`
- `base <= 0`

関連関数

[OCLErrorGet\(\)](#)、[OCINumberLn\(\)](#)

OCINumberMod()

用途

2 つの Oracle 数値の除算の余り（剰余）を取得します。

構文

```
sword OCINumberMod ( OCIError          *err,  
                     CONST OCINumber   *number1,  
                     CONST OCINumber   *number2,  
                     OCINumber         *result );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet () をコールして診断情報を取得します。

number1 (IN)

分子へのポインタです。

number2 (IN)

分母へのポインタです。

result (OUT)

結果の剰余です。

コメント

この関数は、`number1` または `number2` が NULL か、0（ゼロ）による除算エラーがある場合にエラーを戻します。

関連関数

[OCIErrorGet\(\)](#)、[OCINumberDiv\(\)](#)

OCINumberMul()

用途

2 つの Oracle 数値を乗算します。

構文

```
sword OCINumberMul ( OCLError          *err,  
                     CONST OCINumber    *number1,  
                     CONST OCINumber    *number2,  
                     OCINumber          *result );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。 `OCLErrorGet()` をコールして診断情報を取得します。

number1 (IN)

乗算する数値です。

number2 (IN)

乗算する数値です。

result (OUT)

乗算結果です。

コメント

`number1` に `number2` を乗算し、その結果を `result` に戻します。

この関数は、NULL の数値引数がある場合はエラーを戻します。

関連関数

[OCLErrorGet\(\)](#)、[OCINumberDiv\(\)](#)

OCINumberNeg()

用途

Oracle 数値を負の数値にします。

構文

```
sword OCINumberNeg ( OCLError          *err,
                     CONST OCINumber    *number,
                     OCINumber          *result );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。 `OCLErrorGet()` をコールして診断情報を取得します。

number (IN)

負にする数値です。

result (OUT)

`number` の負の値が入ります。

コメント

この関数は、`NULL` の数値引数がある場合はエラーを戻します。

関連関数

[OCLErrorGet\(\)](#)、[OCINumberAbs\(\)](#)、[OCINumberSign\(\)](#)

OCI_{NumberPower}()

用途

指定の基数を指定の指数で累乗します。

構文

```
sword OCINumberPower ( OCIError          *err,  
                        CONST OCINumber    *base,  
                        CONST OCINumber    *number,  
                        OCINumber          *result );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCI_{ErrorGet}() をコールして診断情報を取得します。

base (IN)

指数の基数です。

number (IN)

基数が累乗される指数です。

result (OUT)

指数の出力です。

コメント

この関数は、NULL の数値引数がある場合はエラーを戻します。

関連関数

[OCI_{ErrorGet}\(\)](#)、[OCI_{NumberExp}\(\)](#)

OCINumberPrec()

用途

OCINumber を指定の小数点以下の桁数に丸めます。

構文

```
sword OCINumberPrec ( OCIError *err,  
                      CONST OCINumber *number,  
                      eword nDigs,  
                      OCINumber *result );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet () をコールして診断情報を取得します。

number (IN)

精度を設定する数値です。

nDigs (IN)

目的の結果の小数点以下の桁数です。

result (OUT)

結果です。

コメント

桁数に基づいて、浮動小数点を丸めます。

この関数は、NULL の数値引数がある場合はエラーを戻します。

関連関数

[OCIErrorGet \(\)](#)、[OCINumberRound \(\)](#)

OCINumberRound()

用途

Oracle 数値を指定の桁数で丸めます。

構文

```
sword OCINumberRound ( OCIError          *err,  
                        CONST OCINumber    *number,  
                        sword               decplace,  
                        OCINumber          *result );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。 `OCIErrorGet()` をコールして診断情報を取得します。

number (IN)

丸める数値です。

decplace (IN)

丸める小数点以下の桁数です。負数も可能です。

result (OUT)

丸めた結果の出力です。

コメント

この関数は、NULL の数値引数がある場合はエラーを戻します。

関連関数

[OCIErrorGet\(\)](#)、[OCINumberTrunc\(\)](#)

OCINumberSetPi()

用途

OCINumber を Pi に設定します。

構文

```
void OCINumberSetPi ( OCIError *err,  
                     OCINumber *num );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、*err* に記録され、OCI_ERROR が戻されます。OCIErrorGet () をコールして診断情報を取得します。

num (OUT)

Pi の値に設定される数値です。

コメント

指定の数値を Pi の値に初期化します。

関連関数

[OCIErrorGet \(\)](#)

OCINumberSetZero()

用途

Oracle 数値を 0（ゼロ）に初期化します。

構文

```
void OCINumberSetZero ( OCLError      *err  
                        OCINumber      *num );
```

パラメータ

err (IN)

有効な OCI エラー・ハンドルです。関数がエラーを生成しないため、この関数はエラーをチェックしません。

num (IN/OUT)

0（ゼロ）値に初期化する数値です。

コメント

なし

関連関数

[OCLErrorGet\(\)](#)

OCINumberShift()

用途

数値を 10 の累乗で乗算し、小数点を指定の桁に移動します。

構文

```
sword OCINumberShift ( OCLError      *err,  
                        CONST OCINumber *number,  
                        CONST sword     nDig,  
                        OCINumber      *result );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。 `OCLErrorGet()` をコールして診断情報を取得します。

number (IN)

桁を移動する Oracle 数値です。

nDig (IN)

移動する小数点以下の桁数です。

result (OUT)

桁を移動した結果です。

コメント

数値を 10^{nDig} で乗算し、`product` に結果を設定します。

この関数は、入力数値が `NULL` の場合はエラーを戻します。

関連関数

[OCLErrorGet\(\)](#)

OCINumberSign()

用途

Oracle 数値の符号を取得します。

構文

```
sword OCINumberSign ( OCIError          *err,  
                      CONST OCINumber    *number,  
                      sword               *result );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet () をコールして診断情報を取得します。

number (IN)

符号が戻される数値です。

result (OUT)

可能な値は次のとおりです。

<i>number</i> の値	<i>result</i> パラメータの出力
<code>number < 0</code>	-1
<code>number == 0</code>	0
<code>number > 0</code>	1

コメント

この関数は、`number` または `result` が NULL の場合にエラーを戻します。

関連関数

[OCIErrorGet \(\)](#)、[OCINumberAbs \(\)](#)

OCINumberSin()

用途

Oracle 数値のサインをラジアン単位で計算します。

構文

```
sword OCINumberSin ( OCLError          *err,
                     CONST OCINumber    *number,
                     OCINumber          *result );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。 `OCLErrorGet()` をコールして診断情報を取得します。

number (IN)

ラジアン単位でのサインの引数です。

result (OUT)

サインの結果です。

コメント

この関数は、`NULL` の数値引数がある場合はエラーを戻します。

関連関数

[OCLErrorGet\(\)](#)、[OCINumberArcSin\(\)](#)

OCINumberSqrt()

用途

Oracle 数値の平方根を計算します。

構文

```
sword OCINumberSqrt ( OCIError          *err,  
                      CONST OCINumber    *number,  
                      OCINumber          *result );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、OCI_ERROR が戻されます。OCIErrorGet () をコールして診断情報を取得します。

number (IN)

入力数値です。

result (OUT)

入力数値の平方根を含む出力です。

コメント

この関数は、*number* が NULL または負数の場合はエラーを戻します。

関連関数

[OCIErrorGet \(\)](#)、[OCINumberPower \(\)](#)

OCINumberSub()

用途

2 つの Oracle 数値を減算します。

構文

```
sword OCINumberSub ( OCLError          *err,  
                     CONST OCINumber    *number1,  
                     CONST OCINumber    *number2,  
                     OCINumber          *result );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。 `OCLErrorGet()` をコールして診断情報を取得します。

number1、number2 (IN)

この関数は、`number1` から `number2` を減算します。

result (OUT)

減算結果です。

コメント

`number1` から `number2` を減算し、その結果を `result` に戻します。

この関数は、`NULL` の数値引数がある場合はエラーを戻します。

関連関数

[OCLErrorGet\(\)](#)、[OCINumberAdd\(\)](#)

OCINumberTan()

用途

Oracle 数値のタンジェントをラジアン単位で計算します。

構文

```
sword OCINumberTan ( OCLError          *err,  
                     CONST OCINumber    *number,  
                     OCINumber          *result );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、OCI_ERROR が戻されます。OCLErrorGet () をコールして診断情報を取得します。

number (IN)

ラジアン単位でのタンジェントの引数です。

result (OUT)

タンジェントの結果です。

コメント

この関数は、NULL の数値引数がある場合はエラーを戻します。

関連関数

[OCLErrorGet \(\)](#)、[OCINumberArcTan \(\)](#)、[OCINumberArcTan2 \(\)](#)

OCINumberToInt()

用途

Oracle 数値型を整数に変換します。

構文

```
sword OCINumberToInt ( OCIError          *err,
                        CONST OCINumber    *number,
                        uword              rsl_length,
                        uword              rsl_flag,
                        dvoid              *rsl );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、*err* に記録され、OCI_ERROR が戻されます。OCIErrorGet () をコールして診断情報を取得します。

number (IN)

変換する数値です。

rsl_length (IN)

適切な結果のサイズです。

rsl_flag (IN)

出力値の符号を指定するフラグで、可能な値は次のとおりです。

- OCI_NUMBER_UNSIGNED — 符号なしの値
- OCI_NUMBER_SIGNED — 符号付きの値

rsl (OUT)

結果を保存する領域へのポインタです。

コメント

これは、システム固有な型変換関数です。指定の Oracle 数値を **ub2**、**ub4**、**sb2** など、**xbn** という書式の整数に変換します。

この関数は、*number* または *rsl* が NULL の場合、*number* が大きすぎる（オーバーフロー）か小さすぎる（アンダーフロー）場合、または無効な符号フラグの値が *rsl_flag* に渡された場合にエラーを戻します。

関連関数

[OCIErrorGet \(\)](#)、[OCINumberFromInt \(\)](#)

OCINumberToReal()

用途

Oracle 数値型を実数に変換します。

構文

```
sword OCINumberToReal ( OCLError          *err,  
                        CONST OCINumber    *number,  
                        uword              rsl_length,  
                        dvoid              *rsl );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。 `OCLErrorGet()` をコールして診断情報を取得します。

number (IN)

変換する数値です。

rsl_length (IN)

目的の結果のサイズです。 `sizeof({float | double | long double})` に等しくなります。

rsl (OUT)

結果を保存する領域へのポインタです。

コメント

これは、システム固有な型変換関数です。Oracle 数値をマシンネイティブ実数型に変換します。この関数は、単に最大 `LDBL_DIG` または `DBL_DIG`、`FLT_DIG` 桁の精度の数値を変換し、後続の 0 (ゼロ) を削除します。これらの定数は、`float.h` に定義されます。

この関数は、`number` または `rsl` が `NULL` であるか、`rsl_length = 0` の場合はエラーを戻します。

関連関数

[OCLErrorGet\(\)](#)、[OCINumberFromReal\(\)](#)

OCINumberToText()

用途

指定された書式に従って、Oracle 数値を文字列に変換します。

構文

```
sword OCINumberToText ( OCLError          *err,  
                        CONST OCINumber    *number,  
                        CONST OraText       *fmt,  
                        ub4                 fmt_length,  
                        CONST OraText       *nls_params,  
                        ub4                 nls_p_length,  
                        ub4                 *buf_size,  
                        text                 *buf );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、*err* に記録され、OCI_ERROR が戻されます。OCIErrorGet () をコールして診断情報を取得します。

number (IN)

変換する Oracle 数値です。

fmt (IN)

変換書式です。

fmt_length (IN)

fmt パラメータの長さです。

nls_params (IN)

グローバル・サポートのフォーマット指定です。ヌル文字列 ((text *)0) の場合は、そのセッションのデフォルト・パラメータが使用されます。

nls_p_length (IN)

nls_params パラメータの長さです。

buf_size (IN)

バッファのサイズです。

buf (OUT)

変換された文字列が配置されるバッファです。

コメント

書式およびグローバル・サポート・パラメータの詳細は、『Oracle9i SQL リファレンス』の TO_NUMBER 変換関数の説明を参照してください。

変換された数字文字列は、最大 *buf_size* バイトまで *buf* に格納されます。この関数は、次の場合にエラーを戻します。

- *number* または *buf* が NULL である場合
- バッファが小さすぎる場合
- 書式またはマルチバイト・フォーマットが無効である場合
- 数値を指定の書式のテキストに翻訳するときにオーバーフローが発生した場合

関連関数

[OCIErrorGet\(\)](#)、[OCINumberFromText\(\)](#)

OCINumberTrunc()

用途

Oracle 数値を指定の桁数で切り捨てます。

構文

```
sword OCINumberTrunc ( OCIError          *err,  
                        CONST OCINumber    *number,  
                        sword               decplace,  
                        OCINumber          *result );
```

パラメータ

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet () をコールして診断情報を取得します。

number (IN)

入力数値です。

decplace (IN)

切り捨てる小数点以下の桁数です。負数も可能です。

result (OUT)

切捨ての出力です。

コメント

この関数は、NULL の数値引数がある場合はエラーを戻します。

関連関数

[OCIErrorGet \(\)](#)、[OCINumberRound \(\)](#)

OCI ロー関数

この項では、OCI ロー関数について説明します。

表 18-5 ロー関数

関数	用途
OCIRawAllocSize() (18-144 ページ)	ロー・メモリーに割り当てられたサイズをバイト単位で取得します。
OCIRawAssignBytes() (18-145 ページ)	ローにロー・バイトを割り当てます。
OCIRawAssignRaw() (18-146 ページ)	ローにローを割り当てます。
OCIRawPtr() (18-147 ページ)	ロー・データのポインタを取得します。
OCIRawResize() (18-148 ページ)	可変長ローのメモリー・サイズを変更します。
OCIRawSize() (18-149 ページ)	ローのサイズを取得します。

OCIRawAllocSize()

用途

ロー・メモリーの割当てサイズをバイト単位で取得します。

構文

```
sword OCIRawAllocSize ( OCIEnv          *env,  
                        OCIError        *err,  
                        CONST OCIRaw     *raw,  
                        ub4              *allocsize );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。`OCIErrorGet()` をコールして診断情報を取得します。

raw (IN)

割り当てられたサイズがバイト単位で戻されるロー・データです。これは、NULL 以外のポインタにしてください。

allocsize (OUT)

ロー・メモリーの割当てサイズをバイト単位で戻します。

コメント

割り当てられるサイズは、実際のロー・サイズより大きいかにそれに等しくなります。

関連関数

[OCIErrorGet\(\)](#)、[OCIRawResize\(\)](#)、[OCIRawSize\(\)](#)

OCIRawAssignBytes()

用途

ub1* 型のロー・バイトを Oracle の **OCIRaw*** データ型に割り当てます。

構文

```
sword OCIRawAssignBytes ( OCIEnv          *env,  
                           OCIError       *err,  
                           CONST ub1      *rhs,  
                           ub4            rhs_len,  
                           OCIRaw        **lhs );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、**OCI_ERROR** が戻されます。OCIErrorGet () をコールして診断情報を取得します。

rhs (IN)

割当ての右側（ソース）となる **ub1** データ型です。

rhs_len (IN)

rhs ロー・バイトの長さです。

lhs (IN/OUT)

割当ての左側（ターゲット）となる **OCIRaw** データです。

コメント

rhs ロー・バイトを *lhs* ロー・データ型に割り当てます。*lhs* ローのサイズは、*rhs* のサイズに応じて変更されます。割り当てられたロー・バイトの型は **ub1** になります。

関連関数

[OCIErrorGet\(\)](#)、[OCIRawAssignRaw\(\)](#)

OCIRawAssignRaw()

用途

ある Oracle RAW データ型を別の Oracle RAW データ型に割り当てます。

構文

```
sword OCIRawAssignRaw ( OCIEnv          *env,  
                        OCIError        *err,  
                        CONST OCIRaw    *rhs,  
                        OCIRaw          **lhs );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。`OCIErrorGet()` をコールして診断情報を取得します。

rhs (IN)

割当ての右側（ソース）となる **OCIRaw** データです。

lhs (IN/OUT)

割当ての左側（ターゲット）となる **OCIRaw** データです。

コメント

`rhs` ローを `lhs` ローに割り当てます。`lhs` ローのサイズは、`rhs` のサイズに応じて変更されます。

関連関数

[OCIErrorGet\(\)](#)、[OCIRawAssignBytes\(\)](#)

OCIRawPtr()

用途

ロー・データへのポインタを取得します。

構文

```
ub1 *OCIRawPtr ( OCIEnv          *env,  
                  CONST OCIRaw    *raw );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

raw (IN)

指定されたロー・データのポインタを戻します。

コメント

なし

関連関数

[OCLErrorGet\(\)](#)、[OCIRawAssignRaw\(\)](#)

OCIRawResize()

用途

指定の可変長ローのメモリー・サイズを変更します。

構文

```
sword OCIRawResize ( OCIEnv          *env,  
                     OCIError       *err,  
                     ub2             new_size,  
                     OCIRaw         **raw );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。`OCIErrorGet()` をコールして診断情報を取得します。

new_size (IN)

バイト単位での新規ロー・データのサイズです。

raw (IN)

可変長ロー・ポインタです。ローのサイズは `new_size` に変更されます。

コメント

この関数は、オブジェクト・キャッシュ内の指定の可変長ローのメモリー・サイズを変更します。そのローの以前の内容は保持されません。この関数は、新しいメモリー・リージョンにローを割り当てる場合があります。その場合、指定のローが占有していたメモリーは解放されます。入力ローが `NULL` (`raw == NULL`) の場合、この関数は、ロー・データ用のメモリーを割り当てます。

`new_size` が 0 (ゼロ) の場合、この関数は `raw` が占有していたメモリーを解放し、`NULL` ポインタ値を戻します。

関連関数

[OCIErrorGet\(\)](#)、[OCIRawAllocSize\(\)](#)、[OCIRawSize\(\)](#)

OCIRawSize()

用途

指定のローのサイズをバイト単位で戻します。

構文

```
ub4 OCIRawSize ( OCIEnv          *env,  
                  CONST OCIRaw    *raw );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

raw (IN/OUT)

サイズが戻されるローです。

コメント

なし

関連関数

[OCLErrorGet\(\)](#)、[OCIRawAllocSize\(\)](#)、[OCIRawSize\(\)](#)

OCI REF 関数

この項では、OCI REF 関数について説明します。

表 18-6 REF 関数

関数	用途
OCIRefAssign() (18-151 ページ)	1 つの REF を別の REF に代入します。
OCIRefClear() (18-152 ページ)	REF を消去または NULL にします。
OCIRefFromHex() (18-153 ページ)	16 進文字列を REF に変換します。
OCIRefHexSize() (18-154 ページ)	REF の 16 進表現のサイズを戻します。
OCIRefIsEqual() (18-155 ページ)	2 つの REF が等しいか比較します。
OCIRefIsNull() (18-156 ページ)	REF が NULL かどうかをテストします。
OCIRefToHex() (18-157 ページ)	REF を 16 進文字列に変換します。

OCIRefAssign()

用途

ある REF を別の REF に割り当て、両方が同じオブジェクトを参照するようにします。

構文

```
sword OCIRefAssign ( OCIEnv          *env,  
                    OCIError        *err,  
                    CONST OCIRef     *source,  
                    OCIRef          **target );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet() をコールして診断情報を取得します。

source (IN)

コピー元の REF です。

target (IN/OUT)

コピー先の REF です。

コメント

`source` REF を `target` REF にコピーし、両方が同じオブジェクトを参照するようにします。`target` REF ポインタが NULL (`*target == NULL`) の場合、OCIRefAssign() は、コピーを実行する前に OCI オブジェクト・キャッシュ内に `target` REF 用のメモリーを割り当てます。

関連関数

[OCIErrorGet\(\)](#)、[OCIRefIsEqual\(\)](#)

OCIRefClear()

用途

指定の REF を消去または無効化します。

構文

```
void OCIRefClear ( OCISvcCtx    *env,  
                  OCIRef       *ref );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

ref (IN/OUT)

消去する REF です。

コメント

オブジェクトを指し示さない REF は、ヌルの REF とみなされます。論理的には、ヌルの REF は参照先がない REF になります。

ヌルの REF は依然として有効な SQL 値であり、SQL の NULL ではないことに注意してください。これは、表内の NOT NULL 列または行属性用の有効な非 NULL 定数 REF 値として使用できます。

REF として NULL ポインタ値が渡されると、この関数による操作は実行されません。

関連関数

[OCIErrorGet\(\)](#)、[OCIRefIsNull\(\)](#)

OCIRefFromHex()

用途

指定の 16 進文字列を REF に変換します。

構文

```
sword OCIRefFromHex ( OCIEnv          *env,  
                      OCIError        *err,  
                      CONST OCISvcCtx  *svc,  
                      CONST OraText    *hex,  
                      ub4              length,  
                      OCIRef           **ref );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet() をコールして診断情報を取得します。

svc (IN)

OCI サービス・コンテキスト・ハンドルです（結果の REF がこのサービス・コンテキストで初期化される場合）。

hex (IN)

REF に変換する 16 進文字列で、以前に OCIRefToHex() によって出力されたものです。

length (IN)

16 進文字列の長さです。

ref (IN/OUT)

16 進文字列の変換先となる REF です。入力時に `*ref` が NULL の場合は、その REF 用の領域がオブジェクト・キャッシュ内に割り当てられます。それ以外の場合は、指定の REF によって占有されているメモリが再利用されます。

コメント

この関数は、変換後の REF の書式が正しいことを保証します。ただし、変換後の REF によって指示されるオブジェクトが存在するかどうかは保証しません。

関連関数

[OCIErrorGet\(\)](#)、[OCIRefToHex\(\)](#)

OCIRefHexSize()

用途

REF の 16 進表現のサイズを返します。

構文

```
ub4 OCIRefHexSize ( OCIEnv          *env,  
                    CONST OCIRef     *ref );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

ref (IN)

16 進表現のサイズがバイト単位で戻される REF です。

戻り値

REF の 16 進表現のサイズです。

コメント

REF の 16 進表現のために必要なバッファ・サイズをバイト単位で返します。最低このサイズのバッファを、REF から 16 進への変換関数（[OCIRefToHex\(\)](#)）に渡す必要があります。

関連関数

[OCIErrorGet\(\)](#)、[OCIRefFromHex\(\)](#)

OCIRefsEqual()

用途

2 つの REF を比較してそれらが同等であるかどうかを判断します。

構文

```
boolean OCIRefIsEqual ( OCIEnv          *env,  
                        CONST OCIRef     *x,  
                        CONST OCIRef     *y );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

x (IN)

比較する REF です。

y (IN)

比較する REF です。

戻り値

2 つの REF が同等な場合は TRUE です。

2 つの REF が等しくない場合、*x* が NULL の場合、または *y* が NULL の場合は FALSE です。

コメント

2 つの REF は、オブジェクトの種類（永続または一時）に関係なく同じオブジェクトを参照している場合のみ同等になります。

注意： 2 つの NULL REF は、この関数では等しくないとみなされます。

関連関数

[OCIErrorGet\(\)](#)、[OCIRefAssign\(\)](#)

OCIRefIsNull()

用途

REF が NULL かどうかを確認します。

構文

```
boolean OCIRefIsNull ( OCIEnv          *env,  
                      CONST OCIRef     *ref );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

ref (IN)

NULL であるかどうかをテストする REF です。

戻り値

指定された REF が NULL の場合は TRUE を返します。それ以外の場合は FALSE を返します。

コメント

REF は、次に該当する場合のみ NULL になります。

- 永続オブジェクトを参照するが、オブジェクトの識別子が NULL である場合
- 一時オブジェクトを参照するが、現時点ではオブジェクトを指し示していない場合

注意： 指し示すオブジェクトが存在しない場合、REF は参照先がない REF です。

関連関数

[OCIErrorGet\(\)](#)、[OCIRefClear\(\)](#)

OCIRefToHex()

用途

REF を 16 進文字列に変換します。

構文

```
sword OCIRefToHex ( OCIEEnv          *env,
                   OCIError         *err,
                   CONST OCIRef      *ref,
                   OraText           *hex,
                   ub4               *hex_length );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、err に記録され、OCI_ERROR が戻されます。OCIErrorGet () をコールして診断情報を取得します。

ref (IN)

16 進文字列に変換する REF です。ref が NULL REF (つまり、OCIRefIsNull(ref) == TRUE) の場合は、0 (ゼロ) の hex_length 値が戻されます。

hex (OUT)

結果の 16 進文字列を格納するのに十分なバッファです。文字列の内容は、コール元にとっては不透明になります。

hex_length (IN/OUT)

入力では hex バッファのサイズを指定し、出力では hex で戻される 16 進文字列の実際のサイズを指定します。

コメント

指定された REF を 16 進文字列に変換し、その文字列の長さを戻します。変換された文字列は、コール元にとっては不透明になります。

この関数は、指定されたバッファが変換後の文字列を保持できるだけの大きさがいない場合はエラーを戻します。

関連関数

[OCIErrorGet\(\)](#)、[OCIRefFromHex\(\)](#)、[OCIRefHexSize\(\)](#)、[OCIRefIsNull\(\)](#)

OCI 文字列関数

この項では、OCI 文字列関数について説明します。

表 18-7 文字列関数

関数	用途
<code>OCIStringAllocSize()</code> (18-159 ページ)	文字列メモリーの割り当てられたサイズ (バイト単位) を取得します。
<code>OCIStringAssign()</code> (18-160 ページ)	文字列に文字列を割り当てます。
<code>OCIStringAssignText()</code> (18-161 ページ)	テキスト文字列を文字列に代入します。
<code>OCIStringPtr()</code> (18-162 ページ)	文字列ポインタを取得します。
<code>OCIStringResize()</code> (18-163 ページ)	文字列メモリーをサイズ変更します。
<code>OCIStringSize()</code> (18-164 ページ)	文字列サイズを取得します。

OCIStringAllocSize()

用途

文字列の割当てメモリー・サイズをコードポイント (Unicode) またはバイト単位で取得します。

構文

```
sword OCIStringAllocSize ( OCIEnv          *env,  
                           OCIError        *err,  
                           CONST OCIString *vs,  
                           ub4             *allocsize );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、OCI_ERROR が戻されます。OCIErrorGet() をコールして診断情報を取得します。

vs (IN)

割り当てられたサイズがバイト単位で戻される文字列です。**vs** は NULL 以外のポインタにしてください。

allocsize (OUT)

文字列メモリーの割当てサイズをバイト単位で戻します。

コメント

割り当てられるサイズは、実際の文字列サイズより大きいかにそれに等しくなります。

関連関数

[OCIErrorGet\(\)](#)、[OCIStringResize\(\)](#)、[OCIStringSize\(\)](#)

OCIStringAssign()

用途

文字列を別の文字列に割り当てます。

構文

```
sword OCIStringAssign ( OCIEnv          *env,  
                        OCIError        *err,  
                        CONST OCIString  *rhs,  
                        OCIString       **lhs );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet() をコールして診断情報を取得します。

rhs (IN)

割当ての右側（ソース）です。UTF-16 が可能です。

lhs (IN/OUT)

割当ての左側（ターゲット）です。`rhs` が UTF-16 の場合、バッファは UTF-16 になります。

コメント

`rhs` 文字列を `lhs` 文字列に割り当てます。`lhs` 文字列のサイズは、`rhs` のサイズに応じて変更されます。割り当てられる文字列は、ヌル文字で終了します。長さのフィールドには、ヌル文字による終了に必要な余分なコードポイントやバイトは含まれません。

この関数は、割当て操作で領域が足りなくなった場合はエラーを戻します。

関連関数

[OCIErrorGet\(\)](#)、[OCIStringAssignText\(\)](#)

OCIStringAssignText()

用途

ソース・テキスト文字列をターゲット文字列に割り当てます。

構文

```
sword OCIStringAssignText ( OCIEnv          *env,  
                             OCIError       *err,  
                             CONST OraText   *rhs,  
                             ub2            rhs_len,  
                             OCIString      **lhs );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet () をコールして診断情報を取得します。

rhs (IN)

割当ての右側（ソース）となるテキスト文字列または UTF-16 Unicode 文字列です。

rhs_len (IN)

`rhs` 文字列の長さで、コードポイント（Unicode の場合）またはバイト単位（Unicode 以外の場合）です。

lhs (IN/OUT)

割当ての左側（ターゲット）です。`rhs` が Unicode の場合、バッファは Unicode になります。

コメント

`rhs` 文字列を `lhs` 文字列に割り当てます。`lhs` 文字列のサイズは、`rhs` のサイズに応じて変更されます。割り当てられる文字列は、ヌル文字で終了します。長さのフィールドには、ヌル文字による終了に必要な余分なコードポイントやバイトは含まれません。

関連関数

[OCIErrorGet\(\)](#)、[OCIStringAssign\(\)](#)

OCIStringPtr()

用途

指定の文字列のテキストへのポインタを取得します。

構文

```
text *OCIStringPtr ( OCIEnv          *env,  
                    CONST OCIStrng   *vs );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

vs (IN)

文字列が戻される **OCIStrng** オブジェクトへのポインタです。vs が UTF-16 の場合は、戻されるバッファも UTF-16 になります。戻されるバッファのエンコーディングを調べる場合は、**OCIStrng** vs 自体の UTF-16 情報をチェックします。これは、特定の **OCIStrng** の設定が必ず env と同じ設定であるとは保証されていないためです。チェックする関数は、オブジェクトのメンバー・フィールドをチェックするために設計されたオブジェクト OCI 関数である必要があります。

コメント

なし

関連関数

[OCIErrorGet\(\)](#)、[OCIStringAssign\(\)](#)

OCIStringResize()

用途

指定の文字列のメモリー・サイズを変更します。

構文

```
sword OCIStringResize ( OCIEnv          *env,  
                        OCIError        *err,  
                        ub4              new_size,  
                        OCIString       **str );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。`OCIErrorGet()` をコールして診断情報を取得します。

new_size (IN)

文字列の新規メモリー・サイズ（バイト単位）です。`new_size` には、文字列終了記号のヌル文字用の領域を含める必要があります。

str (IN/OUT)

OCI オブジェクト・キャッシュから解放された文字列の割当てメモリーです。

コメント

この関数は、オブジェクト・キャッシュ内の指定された可変長文字列のメモリー・サイズを変更します。文字列の内容は保持されません。この関数は、新しいメモリー・リージョンに文字列を割り当てることがあります。この場合、指定の文字列が占有していたメモリーは解放されます。`str` が `NULL` の場合、この関数は文字列のメモリーを割り当てます。`new_size` が 0（ゼロ）の場合、この関数は、`str` が占有していたメモリーを解放し、`NULL` ポインタ値を戻します。

関連関数

[OCIErrorGet\(\)](#)、[OCIStringAllocSize\(\)](#)、[OCIStringSize\(\)](#)

OCIStringSize()

用途

指定の文字列 *vs* のサイズを取得します。

構文

```
ub4 OCIStringSize ( OCIEnv          *env,  
                    CONST OCIStrng   *vs );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

vs (IN)

サイズが戻される文字列です。バイト単位です。

コメント

戻されたサイズには、ヌル文字による終了用の余分なバイトは含まれていません。

関連関数

[OCIErrorGet\(\)](#)、[OCIStringResize\(\)](#)

OCI 表関数

この項では、OCI 表関数について説明します。

表 18-8 表関数

関数	用途
OCITableDelete() (18-166 ページ)	要素を削除します。
OCITableExists() (18-167 ページ)	要素が存在するかどうかをテストします。
OCITableFirst() (18-168 ページ)	表の先頭の索引を戻します。
OCITableLast() (18-169 ページ)	表の最後の索引を戻します。
OCITableNext() (18-170 ページ)	使用可能な次の表索引を戻します。
OCITablePrev() (18-171 ページ)	使用可能な 1 つ前の表索引を戻します。
OCITableSize() (18-172 ページ)	表のカレント・サイズを戻します。

OCI TableDelete()

用途

指定の索引位置にある要素を削除します。

構文

```
sword OCI TableDelete ( OCIEnv          *env,
                        OCIError        *err,
                        sb4              index,
                        OCI Table       *tbl );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。`OCIErrorGet()` をコールして診断情報を取得します。

index (IN)

削除する必要のある要素の索引です。

tbl (IN)

要素を削除する表です。

コメント

この関数は、指定した索引の要素がすでに削除されている場合、または指定した索引が表に対して無効の場合は、エラーを戻します。入力パラメータに `NULL` がある場合もエラーになります。

注意： 表の残りの要素の位置序数は、`OCI TableDelete()` で変更されません。この削除操作により、表の中に空きが生成されます。

関連関数

[OCIErrorGet\(\)](#)、[OCI TableExists\(\)](#)

OCITableExists()

用途

指定の索引位置に要素が存在するかどうかをテストします。

構文

```
sword OCITableExists ( OCIEnv          *env,  
                        OCIError        *err,  
                        CONST OCITable   *tbl,  
                        sb4              index,  
                        boolean          *exists );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet () をコールして診断情報を取得します。

tbl (IN)

指定の索引をチェックする表です。

index (IN)

要素の存在をチェックする索引です。

exists (OUT)

指定する `index` に要素が存在する場合は TRUE です。それ以外の場合は、FALSE です。

コメント

この関数は、入力パラメータに NULL がある場合はエラーを戻します。

関連関数

[OCIErrorGet\(\)](#)、[OCITableDelete\(\)](#)

OCITableFirst()

用途

指定の表の先頭の既存要素の索引を戻します。

構文

```
sword OCITableFirst ( OCIEnv          *env,  
                     OCIError        *err,  
                     CONST OCITable   *tbl,  
                     sb4              *index );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。`OCIErrorGet()` をコールして診断情報を取得します。

tbl (IN)

スキャンする表です。

index (OUT)

指定の表に存在する最初の要素の索引が戻されます。

コメント

たとえば、`OCITableDelete()` によって表の先頭の 5 つの要素が削除されている場合、`OCITableFirst()` は 6 を戻します。

関連項目： 表内のデータのない空きについては、「[OCITableDelete\(\)](#)」の説明を参照してください。

この関数は、表が空の場合はエラーを戻します。

関連関数

[OCIErrorGet\(\)](#)、[OCITableDelete\(\)](#)、[OCITableLast\(\)](#)

OCITableLast()

用途

表の最後の既存要素の索引を戻します。

構文

```
sword OCITableLast ( OCIEnv          *env,  
                     OCIError        *err,  
                     CONST OCITable  *tbl,  
                     sb4             *index );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、err に記録され、OCI_ERROR が戻されます。OCIErrorGet() をコールして診断情報を取得します。

tbl (IN)

スキャンする表です。

index (OUT)

表の最後の既存要素の索引です。

コメント

この関数は、表が空の場合はエラーを戻します。

関連関数

[OCIErrorGet\(\)](#)、[OCITableFirst\(\)](#)、[OCITableNext\(\)](#)、[OCITablePrev\(\)](#)

OCITableNext()

用途

表の次の既存要素の索引を戻します。

構文

```
sword OCITableNext ( OCIEnv          *env,
                     OCIError       *err,
                     sb4             index,
                     CONST OCITable *tbl,
                     sb4             *next_index,
                     boolean         *exists );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。`OCIErrorGet()` をコールして診断情報を取得します。

index (IN)

スキャンの開始位置の索引です。

tbl (IN)

スキャンする表です。

next_index (OUT)

`tbl(index)` の次の既存要素の索引です。

exists (OUT)

次の索引が存在しない場合は `FALSE` で、存在する場合は `TRUE` です。

コメント

`index` よりも大きく、`exists(j)` が `TRUE` になるような最小位置 `j` を戻します。

関連項目： 表内のデータのない空きの存在については、[「OCIStringAllocSize\(\)」](#) の説明を参照してください。

関連関数

[OCIErrorGet\(\)](#)、[OCITablePrev\(\)](#)

OCITablePrev()

用途

表の直前の既存要素の索引を戻します。

構文

```
sword OCITablePrev ( OCIEnv          *env,  
                    OCIError        *err,  
                    sb4             index,  
                    CONST OCITable  *tbl,  
                    sb4             *prev_index  
                    boolean         *exists );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet() をコールして診断情報を取得します。

index (IN)

スキヤンの開始位置の索引です。

tbl (IN)

スキヤンする表です。

prev_index (OUT)

tbl(index) の直前の既存要素の索引です。

exists (OUT)

前の索引が存在しない場合は FALSE で、存在する場合は TRUE です。

コメント

index よりも小さく、exists(j) が TRUE になるような最大位置 j を戻します。

関連項目： 表内のデータのない空きの存在については、[OCIStringAllocSize\(\)](#) の説明を参照してください。

関連関数

[OCITableNext\(\)](#)

OCITableSize()

用途

指定の表のサイズを戻します。削除済みの要素は含まれません。

構文

```
sword OCITableSize ( OCIEnv          *env,
                     OCIError        *err,
                     CONST OCITable   *tbl,
                     sb4              *size );
```

パラメータ

env (IN/OUT)

オブジェクト・モードで初期化された OCI 環境ハンドルです。

関連項目： 15-9 ページの「[OCIEnvCreate\(\)](#)」および 15-18 ページの「[OCIInitialize\(\)](#)」

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。`OCIErrorGet()` をコールして診断情報を取得します。

tbl (IN)

要素の数が戻される NESTED TABLE です。

size (OUT)

NESTED TABLE 内の現行の要素数です。削除された要素はカウントに含まれません。

コメント

カウントは、NESTED TABLE から要素を削除すると減分されます。したがって、このカウントには要素の削除によって生成された空きは含まれていません。削除した要素を含まないカウントを取得するには、[OCICollSize\(\)](#) を使用します。

たとえば、次のようにします。

```
OCITableSize(...);
// assume 'size' returned is equal to 5
OCITableDelete(...); // delete one element
OCITableSize(...);
// 'size' returned is equal to 4
```

削除された要素を含めたカウントを取得するには、`OCICollSize()` を使用します。前述の例を続けます。

```
OCICollSize(...)  
// 'size' returned is still equal to 5
```

この関数は、NESTED TABLE のオブジェクト・キャッシュへのロード中にエラーが発生した場合、または入力パラメータに NULL がある場合はエラーを戻します。

関連関数

`OCICollSize()`

OCI カートリッジ関数

この章では、カートリッジ関数について説明します。

関連項目： コード例は、Oracle のインストールに含まれているデモ・プログラムを参照してください。追加情報については、[付録 B「OCI デモ・プログラム」](#)を参照してください。

この章は、次の項目で構成されています。

- [外部プロシージャ関数およびカートリッジ・サービス関数の概要](#)
- [カートリッジ・サービス – OCI 外部プロシージャ](#)
- [カートリッジ・サービス – メモリー・サービス](#)
- [カートリッジ・サービス – コンテキストのメンテナンス](#)
- [カートリッジ・サービス – パラメータ・マネージャ・インタフェース](#)
- [カートリッジ・サービス – ファイル I/O インタフェース](#)
- [カートリッジ・サービス – 文字列のフォーマット・インタフェース](#)

外部プロシージャ関数およびカートリッジ・サービス関数の概要

この章では、最初に OCI 外部プロシージャ関数について説明します。これらの関数により、外部プロシージャのユーザーは、エラーの通知、メモリーの割当ておよび OCI コンテキスト情報の取得ができます。

関連項目： これらの関数を外部プロシージャで使用方法の詳細は、『Oracle9i アプリケーション開発者ガイド - 基礎編』の外部ルーチンの章を参照してください。

次に、カートリッジ・サービスについて説明します。

関連項目： これらの関数の使用方法の詳細は、『Oracle9i Data Cartridge Developer’s Guide』を参照してください。

関数の構文

各関数について、次の情報が記載されています。

用途

この関数によって実行されるアクションを簡単に説明します。

構文

関数の宣言。

パラメータ

この関数の各パラメータの説明。これにはパラメータのモードが含まれます。パラメータのモードには、次の 3 つの値があります。

モード	説明
IN	Oracle にデータを渡すパラメータ
OUT	このコールまたは後続のコールで Oracle からデータを受け取るパラメータ
IN/OUT	このコールでデータを渡し、このコールまたは後続のコールからの戻りでデータを受け取るパラメータ

コメント

この関数に関する詳細情報。関数の使用上の制約やアプリケーション内でこの関数を使用するときに役に立つ情報が記載されています。

戻り値

この関数に対する戻り値のリスト。

関連関数

関連する関数コールのリスト。カートリッジ・サービスについては、ドキュメントに記載されている、グループ内の他のすべての関数を参照してください。

リターン・コード

成功とエラーのリターン・コードは、特定の外部プロシージャ・インタフェース関数に対して定義されます。特定のインタフェース・ファンクションが `OCIEXTPROC_SUCCESS` または `OCIEXTPROC_ERROR` を戻した場合、アプリケーションはこれらのマクロを使用して戻り値をチェックする必要があります。

- `OCIEXTPROC_SUCCESS` — 外部プロシージャの成功した場合のリターン・コード
- `OCIEXTPROC_ERROR` — 外部プロシージャの失敗した場合のリターン・コード

With_Context 型

PL/SQL 外部プロシージャへの C コール可能インタフェースには、`with_context` パラメータを渡す必要がありますこの構造体の型は、`OCIExtProcContext` で、ユーザーには不透明です。

ユーザーは、`with_context` パラメータをアプリケーションで次のように宣言できます。

```
OCIExtProcContext *with_context;
```

カートリッジ・サービス – OCI 外部プロシージャ

C に対応する OCI 外部プロシージャ関数を次に示します。

表 19-1 外部プロシージャ関数

関数	用途
<code>OCIExtProcAllocCallMemory()</code> (19-5 ページ)	外部プロシージャの継続時間にメモリーを割り当てます。
<code>OCIExtProcRaiseExcp()</code> (19-6 ページ)	PL/SQL に例外を呼び出します。
<code>OCIExtProcRaiseExcpWithMsg()</code> (19-7 ページ)	例外をメッセージ付きで呼び出します。
<code>OCIExtProcGetEnv()</code> (19-8 ページ)	OCI 環境ハンドル、サービス・コンテキスト・ハンドルおよびエラー・ハンドルを取得します。

OCIExtProcAllocCallMemory()

用途

外部プロシージャの継続時間に N バイトのメモリーを割り当てます。

構文

```
dvoid * OCIExtProcAllocCallMemory ( OCIExtProcContext    *with_context,  
                                     size_t                amount );
```

Parameters

with_context (IN)

C 外部プロシージャに渡される **with_context** ポインタです。

関連項目： 19-3 ページ [「With_Context 型」](#)

amount (IN)

割り当てるバイト数です。

コメント

このコールは、外部プロシージャのコール継続時間に *amount* バイトのメモリーを割り当てます。

このコールで割り当てられたメモリーは、外部プロシージャから戻るとすぐに PL/SQL によって解放されます。アプリケーションでは、OCIExtProcAllocCallMemory() で割り当てたメモリーに free() 関数を使用しないでください。この関数を使用して、関数の戻り値用のメモリーを割り当ててください。

0 (ゼロ) の戻り値はエラーとして扱われます。

戻り値

割り当てたメモリーを指し示す未定義の型（不透明な）ポインタを戻します。

例

```
text *ptr = (text *)OCIExtProcAllocCallMemory(wctx, 1024)
```

関連関数

[OCIErrorGet\(\)](#)、[OCIMemoryAlloc\(\)](#)

OCIExtProcRaiseExcp()

用途

PL/SQL に例外を呼び出します。

構文

```
size_t OCIExtProcRaiseExcp ( OCIExtProcContext    *with_context,  
                             int                    errnum );
```

Parameters

with_context (IN)

C 外部プロシージャに渡される **with_context** ポインタです。

関連項目： 19-3 ページ [「With_Context 型」](#)

errnum (IN)

PL/SQL にシグナルを送る Oracle エラー番号です。 *errnum* は、1 ～ 32767 の正数にしてください。

コメント

この関数をコールすると、例外が PL/SQL に通知されます。この関数が問題なく戻されると、外部プロシージャは終了処理を開始し、PL/SQL に戻ります。いったん例外が PL/SQL に通知されると、IN/OUT 引数と OUT 引数（ある場合）は処理されません。

戻り値

この関数は、コールが成功した場合は、OCIEXTPROC_SUCCESS を戻します。コールが失敗した場合は、OCIEXTPROC_ERROR を戻します。

関連関数

[OCIExtProcRaiseExcpWithMsg\(\)](#)

OCIExtProcRaiseExcpWithMsg()

用途

例外をメッセージ付きで呼び出します。

構文

```
size_t OCIExtProcRaiseExcpWithMsg ( OCIExtProcContext  *with_context,
                                     int                errnum,
                                     char                *errmsg,
                                     size_t              msglen );
```

パラメータ

with_context (IN)

C 外部プロシージャに渡される **with_context** ポインタです。

関連項目： 19-3 ページ「[With_Context 型](#)」

errnum (IN)

PL/SQL ヘシグナルを送る Oracle エラー番号です。 *errnum* は、1 ～ 32767 の正数にしてください。

errmsg (IN)

errnum に対応付けられたエラー・メッセージです。

len (IN)

エラー・メッセージの長さです。 *errmsg* がヌル文字で終了する文字列の場合は、0（ゼロ）が渡されます。

コメント

PL/SQL に例外が呼び出されます。さらに、後続のエラー・メッセージ文字列を標準の Oracle エラー・メッセージ内で代入します。

関連項目： 詳細は、「[OCIExtProcRaiseExcp\(\)](#)」の説明を参照してください。

戻り値

この関数は、コールが成功した場合は、OCIEXTPROC_SUCCESS を戻します。コールが失敗した場合は、OCIEXTPROC_ERROR を戻します。

関連関数

[OCIExtProcRaiseExcp\(\)](#)

OCIExtProcGetEnv()

用途

OCI 環境、サービス・コンテキストおよびエラー・ハンドルを取得します。

構文

```
sword OCIExtProcGetEnv ( OCIExtProcContext    *with_context,  
                        OCIEnv                envh,  
                        OCISvcCtx              svch,  
                        OCIError               errh );
```

Parameters

with_context (IN)

C 外部プロシージャに渡される **with_context** ポインタです。19-3 ページの「[With_Context 型](#)」を参照してください。

envh (OUT)

OCI 環境ハンドルです。

svch (OUT)

OCI サービス・ハンドルです。

errh (OUT)

OCI エラー・ハンドルです。

コメント

この関数の主な目的は、OCI コールバックで同じトランザクションのデータベースを使用可能にすることです。この関数で取得する OCI ハンドルは、データベースに対する OCI コールバックで使用する必要があります。これらのハンドルを標準の OCI コールを介して取得した場合、これらのハンドルはデータベースへの新規接続を使用するので、同一トランザクション内のコールバックには使用できません。1 つの外部プロシージャで利用できるのは、コールバックと新規接続のどちらか一方で、両方は使用できません。

戻り値

この関数は、コールが成功した場合は、OCI_SUCCESS を戻します。コールが失敗した場合は、OCI_ERROR を戻します。

関連関数

[OCIEnvCreate\(\)](#)、[OCIAttrGet\(\)](#)、[OCIHandleAlloc\(\)](#)

カートリッジ・サービス – メモリー・サービス

表 19-2 メモリー・サービス関数

関数	用途
OCIDurationBegin() (19-10 ページ)	ユーザー期間を開始します。
OCIDurationEnd() (19-11 ページ)	ユーザー期間を終了します。
OCIMemoryAlloc() (19-12 ページ)	指定の継続時間から指定サイズのメモリーを割り当てます。
OCIMemoryResize() (19-14 ページ)	メモリー・チャンクをサイズ変更します。
OCIMemoryFree() (19-15 ページ)	メモリー・チャンクを解放します。

関連項目： これらの関数の使用方法の詳細は、『Oracle9i Data Cartridge Developer's Guide』を参照してください。

OCIDurationBegin()

用途

ユーザー期間を開始します。

構文

```
sword OCIDurationBegin ( OCIEnv          *env,
                          OCIError       *err,
                          CONST OCISvcCtx *svc,
                          OCIDuration    parent,
                          OCIDuration    *duration );
```

パラメータ

env (IN/OUT)

OCI 環境ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。 `OCIErrorGet()` のコールによって診断情報を取得できます。

svc (IN)

OCI サービス・コンテキスト・ハンドルです。これは、カートリッジ・サービスに対して `NULL` として渡されます。

parent (IN)

親の継続時間の時間番号です。次のいずれかになります。

- 以前に作成されたユーザー期間
- `OCI_DURATION_STATEMENT`
- `OCI_DURATION_SESSION`

duration (OUT)

新しく作成されたユーザー期間固有の識別子です。

コメント

この関数によってユーザー期間が開始されます。ユーザーは複数のアクティブなユーザー期間を同時に利用できます。ユーザー期間をネストする必要はありません。`duration` パラメータは、このコールによって作成された期間を識別するための一意の番号を戻すために使用します。

環境パラメータとサービス・コンテキスト・パラメータの両方を `NULL` にすることはできません。

関連関数

[OCIDurationEnd\(\)](#)

OCIDurationEnd()

用途

ユーザー期間を終了します。

構文

```
sword OCIDurationEnd ( OCIEnv          *env,
                       OCIError        *err,
                       CONST OCISvcCtx *svc,
                       OCIDuration     duration,
                       CONST OCISvcCtx *svc );
```

パラメータ

env (IN/OUT)

OCI 環境ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。 `OCIErrorGet()` のコールによって診断情報を取得できます。

duration (IN)

`OCIDurationBegin()` によって以前作成されたユーザー期間です。

svc (IN)

OCI サービス・コンテキストです（カートリッジ・サービスの場合は `NULL` として渡しますが、それ以外の場合は非 `NULL` として渡します）。

コメント

この関数によってユーザー期間が終了します。

環境パラメータとサービス・コンテキスト・パラメータの両方を `NULL` にすることはできません。

関連関数

[OCIDurationBegin\(\)](#)

OCIMemoryAlloc()

用途

このコールは、指定の継続時間から指定サイズのメモリーを割り当てます。

構文

```
sword OCIMemoryAlloc( dvoid      *hndl,  
                      OCIError   *err,  
                      dvoid      **mem,  
                      OCIDuration dur,  
                      ub4        size,  
                      ub4        flags );
```

パラメータ

hndl (IN)

OCI 環境ハンドルです。

err (IN)

エラー・ハンドルです。

mem (OUT)

割り当てられたメモリーです。

dur (IN)

次のいずれかになります（以前作成されたユーザー期間）。

OCI_DURATION_CALLOUT

OCI_DURATION_STATEMENT

OCI_DURATION_SESSION

OCI_DURATION_PROCESS

size (IN)

割り当てられるメモリーのサイズです。

flags (IN)

OCI_MEMORY_CLEARED ビットを設定して、消去されたメモリーを取得します。

コメント

エージェントのコールアウトの継続時間（外部プロシージャ継続時間）にメモリーを割り当てるには、`OCIExtProcAllocCallMemory()` または `dur` を `OCI_DURATION_CALLOUT` として指定する `OCIMemoryAlloc()` を使用します。

戻り値

エラー・コードを戻します。

OCIMemoryResize()

用途

このコールはメモリー・チャンクを新規サイズに変更します。

構文

```
sword OCIMemoryResize( dvoid      *hndl,  
                       OCIError   *err,  
                       dvoid      **mem,  
                       ub4         newsize,  
                       ub4         flags );
```

パラメータ

hndl (IN)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN)

エラー・ハンドルです。

mem (IN/OUT)

OCIMemoryAlloc() を使用して以前割り当てたメモリーへのポインタです。

newsize (IN)

要求されたメモリーのサイズです。

flags (IN)

OCI_MEMORY_CLEARED ビットを設定して消去されたメモリーを取得します。

コメント

メモリーを割り当ててからこの関数をコールしてサイズ変更する必要があります。

戻り値

エラー・コードを戻します。

OCIMemoryFree()

用途

このコールはメモリー・チャンクを解放します。

構文

```
sword OCIMemoryFree( dvoid      *hdl,  
                     OCIError *err,  
                     dvoid      *mem );
```

パラメータ

hdl (IN)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN)

エラー・ハンドルです。

mem (IN/OUT)

OCIMemoryAlloc() を使用して以前割り当てたメモリーへのポインタです。

戻り値

エラー・コードを戻します。

カートリッジ・サービス – コンテキストのメンテナンス

表 19-3 コンテキストのメンテナンス関数

関数	用途
OCIContextSetValue() (19-17 ページ)	特定の継続時間に対する値（アドレス）を保存します。
OCIContextGetValue() (19-19 ページ)	コンテキストに格納されている値を戻します。
OCIContextClearValue() (19-20 ページ)	コンテキストに格納されている値を削除します。
OCIContextGenerateKey() (19-21 ページ)	コールが実行されるたびに一意の 4 バイト値を戻します。

関連項目： これらの関数の使用方法の詳細は、『Oracle9i Data Cartridge Developer’s Guide』を参照してください。

OCIContextSetValue()

用途

このコールは特定の継続時間に対する値（アドレス）を保存します。

構文

```
sword OCIContextSetValue( dvoid      *hndl,  
                          OCIError   *err,  
                          OCIDuration duration,  
                          ubl        *key,  
                          ubl        keylen,  
                          dvoid      *ctx_value );
```

パラメータ

hndl (IN)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN)

エラー・ハンドルです。

duration (IN)

次のいずれかになります（以前作成されたユーザー期間）。

OCI_DURATION_STATEMENT

OCI_DURATION_SESSION

key (IN)

一意のキー値です。

keylen (IN)

キーの長さです。最大の長さは 64 ビットです。

ctx_value (IN)

コンテキストに保存されるポインタです。

コメント

格納されるコンテキスト値は、渡される継続時間より長いかまたは同じ長さの継続時間のメモリーから割り当てする必要があります。渡されるキーはこのセッション内で一意であることが必要です。同じキーと継続時間でコンテキスト値を再度保存しようとする、古いコンテキスト値は新規のコンテキスト値に上書きされます。通常、クライアントは構造を割り当てて、コンテキストにこのコールを使用してアドレスを格納し、さらに `OCIContextGetValue()` を使用して個別のコールでこのアドレスを取得します。キーおよび値の対応付けは `OCIContextClearValue()` をコールして明示的に削除できます。削除しない場合は、継続時間の終了時に消去されます。

戻り値

- 操作が成功した場合は、`OCI_SUCCESS` を戻します。
- 操作が失敗した場合は、`OCI_ERROR` を戻します。

OCIContextGetValue()

用途

このコールは、OCIContextSetValue() をコールして指定キーに対応付けられたコンテキストに格納された値を戻します。

構文

```
sword OCIContextGetValue( dvoid      *hndl,
                          OCIError   *err,
                          ubl        *key,
                          ubl        keylen,
                          dvoid      **ctx_value );
```

パラメータ

hndl (IN)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN)

エラー・ハンドルです。

key (IN)

一意のキー値です。

keylen (IN)

キーの長さです。最大の長さは 64 ビットです。

ctx_value (IN)

コンテキストに格納されている値へのポインタです（値が格納されていない場合は NULL）。

コメント

ctx_value の場合は、戻される格納済みコンテキストに対する事前割当てポインタへのポインタが必要です。

戻り値

- 操作が成功した場合は、OCI_SUCCESS を戻します。
- 操作が失敗した場合は、OCI_ERROR を戻します。

OCIContextClearValue()

用途

このコールは、OCIContextSetValue() をコールして指定キーに対応付けられたコンテキストに格納されている値を削除します。

構文

```
sword OCIContextClearValue( dvoid      *hndl,
                             OCIError  *err,
                             ub1       *key,
                             ub1       keylen );
```

パラメータ

hndl (IN)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN)

エラー・ハンドルです。

key (IN)

一意のキー値です。

keylen (IN)

キーの長さです。最大の長さは 64 ビットです。

コメント

存在しないキーが渡されるとエラーが戻されます。

戻り値

- 操作が成功した場合は、OCI_SUCCESS を戻します。
- 操作が失敗した場合は、OCI_ERROR を戻します。

OCIContextGenerateKey()

用途

このコールは実行されるたびに一意の 4 バイト値を返します。

構文

```
sword OCIContextGenerateKey( dvoid      *hndl,  
                             OCIError  *err,  
                             ub4       *key );
```

パラメータ

hndl (IN)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN)

エラー・ハンドルです。

key (IN)

一意のキー値です。

keylen (IN)

キーの長さです。最大の長さは 64 ビットです。

コメント

この値はセッションごとに一意になります。

戻り値

- 操作が成功した場合は、OCI_SUCCESS を返します。
- 操作が失敗した場合は、OCI_ERROR を返します。

カートリッジ・サービス – パラメータ・マネージャ・インタフェース

表 19-4 パラメータ・マネージャ・インタフェース関数

関数	用途
OCIExtractInit() (19-23 ページ)	パラメータ・マネージャを初期化します。
OCIExtractTerm() (19-24 ページ)	動的に割り当てたすべての記憶域を解放します。
OCIExtractReset() (19-25 ページ)	メモリーを再初期化します。
OCIExtractSetNumKeys() (19-26 ページ)	登録されるキー数をパラメータ・マネージャに通知します。
OCIExtractSetKey() (19-27 ページ)	キーに関する情報をパラメータ・マネージャに登録します。
OCIExtractFromFile() (19-29 ページ)	指定したファイルのキーおよびそのキーの値を処理します。
OCIExtractFromStr() (19-30 ページ)	指定した文字列のキーおよびそのキーの値を処理します。
OCIExtractToInt() (19-31 ページ)	指定されたキーの整数値を取得します。
OCIExtractToBool() (19-32 ページ)	指定されたキーのブール値を取得します。
OCIExtractToStr() (19-33 ページ)	指定されたキーの文字列値を取得します。
OCIExtractToOCINum() (19-34 ページ)	指定されたキーの数値を取得します。
OCIExtractToList() (19-35 ページ)	メモリーに格納されているパラメータ構造からパラメータのリストを生成します。
OCIExtractFromList() (19-36 ページ)	パラメータ・リストの <i>index</i> に示されるパラメータの値のリストを生成します。

関連項目： これらの関数の使用方法の詳細は、『Oracle9i Data Cartridge Developer’s Guide』を参照してください。

OCIExtractInit()

用途

この関数はパラメータ・マネージャを初期化します。

構文

```
sword OCIExtractInit( dvoid      *hdl,  
                     OCIError  *err);
```

パラメータ

hdl (IN/OUT)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、**OCI_ERROR** が戻されます。OCIErrorGet() のコールによって診断情報を取得できます。

コメント

この関数をコールしてから他のパラメータ・マネージャ・ルーチンをコールする必要があります。また、コールは一度しか実行できません。グローバルゼーション・サポート情報はパラメータ・マネージャ・コンテキスト内部に格納されて、OCIExtract* ルーチンの後続のコールで使用されます。

戻り値

OCI_SUCCESS

OCI_INVALID_HANDLE

OCI_ERROR

OCIExtractTerm()

用途

この関数は、動的に割り当てたすべての記憶域を解放します。

構文

```
sword OCIExtractTerm( dvoid      *hdl,  
                      OCIError   *err );
```

パラメータ

hdl (IN/OUT)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。`OCIErrorGet()` のコールによって診断情報を取得できます。

コメント

この関数は、他の内部ブックキーピング関数に対して実行されます。この関数は、パラメータ・マネージャが使用されなくなってからコールする必要があります。また、コールは 1 度しか実行できません。

戻り値

`OCI_SUCCESS`

`OCI_INVALID_HANDLE`

`OCI_ERROR`

OCIExtractReset()

用途

パラメータ・ストレージ、定義ストレージおよびパラメータ値リスト用に現在使用されているメモリーを解放して構造を再初期化します。

構文

```
sword OCIExtractReset( dvoid      *hdl,  
                       OCIError  *err );
```

パラメータ

hdl (IN/OUT)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、*err* に記録され、OCI_ERROR が戻されます。OCIErrorGet() のコールによって診断情報を取得できます。

戻り値

OCI_SUCCESS

OCI_INVALID_HANDLE

OCI_ERROR

OCIExtractSetNumKeys()

用途

登録されるキー数をパラメータ・マネージャに通知します。

構文

```
sword OCIExtractSetNumKeys( dvoid    *hndl,  
                             CLError *err,  
                             uword    numkeys );
```

パラメータ

hndl (IN/OUT)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、OCI_ERROR が戻されます。OCIErrorGet () のコールによって診断情報を取得できます。

numkeys (IN)

OCIExtractSetKey () で登録されるキー数です。

コメント

このルーチンは OCIExtractSetKey () の最初のコールの前にコールする必要があります。

戻り値

OCI_SUCCESS

OCI_INVALID_HANDLE

OCI_ERROR

OCIExtractSetKey()

用途

キーに関する情報をパラメータ・マネージャに登録します。

構文

```
sword OCIExtractSetKey( dvoid      *hndl,
                        OCIError   *err,
                        CONST text  *name,
                        ub1         type,
                        ub4         flag,
                        CONST dvoid *defval,
                        CONST sb4   *inrange,
                        CONST text  *CONST *strlist );
```

パラメータ

hndl (IN/OUT)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、OCI_ERROR が戻されます。OCIErrorGet () のコールによって診断情報を取得できます。

name (IN)

キーの名前です。

type (IN)

次のキーの型です。

OCI_EXTRACT_TYPE_INTEGER

OCI_EXTRACT_TYPE_OCINUM

OCI_EXTRACT_TYPE_STRING

OCI_EXTRACT_TYPE_BOOLEAN

flag (IN)

キーに複数の値を使用できる場合は OCI_EXTRACT_MULTIPLE に設定し、それ以外の場合は 0 (ゼロ) に設定します。

defval (IN)

キーにデフォルト値を設定します。デフォルト値がない場合は NULL を設定できます。文字列のデフォルト値は (text*) 型、整数のデフォルト値は (sb4*) 型およびブール・デフォルト値は (ub1*) 型にする必要があります。

intrange (IN)

許容範囲の整数値の開始値および終了値です。キーが整数型でない場合、またはすべての整数値が受け入れ可能な場合は NULL になります。

strlist (IN)

0（または NULL）で終了するキーで受け入れ可能なすべてのテキスト文字列のリストです。キーが文字列型でない場合またはすべてのテキスト値が受け入れ可能である場合は NULL になります。

コメント

このルーチンは OCIExtractNumKeys() をコールしてから実行し、その後に OCIExtractFromFile() または OCIExtractFromStr() をコールする必要があります。

戻り値

OCI_SUCCESS

OCI_INVALID_HANDLE

OCI_ERROR

OCIExtractFromFile()

用途

指定したファイルのキーおよびそのキーの値を処理します。

構文

```
sword OCIExtractFromFile( dvoid      *hndl,
                          OCIError   *err,
                          ub4         flag,
                          text        *filename );
```

パラメータ

hndl (IN/OUT)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、OCI_ERROR が戻されます。OCIErrorGet () のコールによって診断情報を取得できます。

flag (IN)

0 (ゼロ) または次の 1 つ以上のビットを設定します。

OCI_EXTRACT_CASE_SENSITIVE

OCI_EXTRACT_UNIQUE_ABBREVS

OCI_EXTRACT_APPEND_VALUES

filename (IN)

ヌル文字で終了するファイル名文字列です。

コメント

このルーチンをコールする前に、OCIExtractSetNumKeys () および OCIExtractSetKey () ルーチンをコールしてすべてのキーを定義する必要があります。

戻り値

OCI_SUCCESS

OCI_INVALID_HANDLE

OCI_ERROR

OCIExtractFromStr()

用途

指定した文字列のキーおよびそのキーの値を処理します。

構文

```
sword OCIExtractFromStr( dvoid      *hndl,
                        OCIError *err,
                        ub4       flag,
                        text      *input );
```

パラメータ

hndl (IN/OUT)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet() のコールによって診断情報を取得できます。

flag (IN)

0 (ゼロ) または次の 1 つ以上のビットを設定します。

OCI_EXTRACT_CASE_SENSITIVE

OCI_EXTRACT_UNIQUE_ABBREVS

OCI_EXTRACT_APPEND_VALUES

input (IN)

ヌル文字で終了する入力文字列です。

コメント

このルーチンをコールする前に、OCIExtractSetNumKeys() および OCIExtractSetKey() ルーチンをコールしてすべてのキーを定義する必要があります。

戻り値

OCI_SUCCESS

OCI_INVALID_HANDLE

OCI_ERROR

OCIExtractToInt()

用途

指定されたキーの整数値を取得します。valno 値（0 から開始）が戻されます。

構文

```
sword OCIExtractToInt( dvoid      *hndl,
                      OCIError   *err,
                      text       *keyname,
                      uword      valno,
                      sb4        *retval );
```

パラメータ

hndl (IN)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、err に記録され、OCI_ERROR が戻されます。OCIErrorGet () のコールによって診断情報を取得できます。

keyname (IN)

キー名 (IN) です。

valno (IN)

このキーで取得する値です。

retval (OUT)

実際の整数値です。

戻り値

OCI_SUCCESS

OCI_INVALID_HANDLE

OCI_NO_DATA

OCI_ERROR

OCI_NO_DATA は、このキーには valno 値がないことを意味します。

OCIExtractToBool()

用途

指定されたキーのブール値を取得します。**valno** 値（0 から開始）が戻されます。

構文

```
sword OCIExtractToBool( dvoid      *hndl,
                        OCIError   *err,
                        text       *keyname,
                        uword      valno,
                        ub1        *retval );
```

パラメータ

hndl (IN)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、OCI_ERROR が戻されます。OCIErrorGet() のコールによって診断情報を取得できます。

keyname (IN)

キー名です。

valno (IN)

このキーで取得する値です。

retval (OUT)

実際のブール値です。

戻り値

OCI_SUCCESS

OCI_INVALID_HANDLE

OCI_NO_DATA

OCI_ERROR

OCI_NO_DATA は、このキーには **valno** 値がないことを意味します。

OCIExtractToStr()

用途

指定されたキーの文字列値を取得します。valno 値 (0 から開始) が戻されます。

構文

```
sword OCIExtractToStr( dvoid *hndl,  
                      OCIError *err,  
                      text *keyname,  
                      uword valno,  
                      text *retval,  
                      uword buflen );
```

パラメータ

hndl (IN)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、err に記録され、OCI_ERROR が戻されます。OCIErrorGet() のコールによって診断情報を取得できます。

keyname (IN)

キー名です。

valno (IN)

このキーで取得する値です。

retval (OUT)

ヌル文字で終了する実際の文字列値です。

buflen

retval のバッファの長さです。

戻り値

OCI_SUCCESS

OCI_INVALID_HANDLE

OCI_NO_DATA

OCI_ERROR

OCI_NO_DATA は、このキーには valno 値がないことを意味します。

OCIExtractToOCINum()

用途

指定されたキーの OCINumber 値を取得します。valno 値 (0 から開始) が戻されます。

構文

```
sword OCIExtractToOCINum( dvoid      *hndl,
                          OCIError   *err,
                          text        *keyname,
                          uword       valno,
                          OCINumber  *retval );
```

パラメータ

hndl (IN)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、*err* に記録され、OCI_ERROR が戻されます。OCIErrorGet() のコールによって診断情報を取得できます。

keyname (IN)

キー名です。

valno (IN)

このキーで取得する値です。

retval (OUT)

実際の OCINumber 値です。

戻り値

OCI_SUCCESS

OCI_INVALID_HANDLE

OCI_NO_DATA または OCI_ERROR

OCI_NO_DATA は、このキーには valno 値がないことを意味します。

OCIExtractToList()

用途

メモリーに格納されているパラメータ構造からパラメータのリストを生成します。
OCIExtractValues() をコールする前にコールする必要があります。

構文

```
sword OCIExtractToList( dvoid      *hndl,  
                        OCIError   *err,  
                        uword      *numkeys );
```

パラメータ

hndl (IN)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、OCI_ERROR が戻されます。OCIErrorGet() のコールによって診断情報を取得できます。

numkeys (OUT)

メモリーに格納されている個別キーの数です。

戻り値

OCI_SUCCESS

OCI_INVALID_HANDLE

OCI_ERROR

OCIExtractFromList()

用途

パラメータ・リストの索引で示されるパラメータの値のリストを生成します。

構文

```
sword OCIExtractFromList( dvoid      *hndl,
                          OCIError   *err,
                          uword      index,
                          text       **name,
                          ubl        *type,
                          uword      *numvals,
                          dvoid      ***values );
```

パラメータ

hndl (IN)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet () のコールによって診断情報を取得できます。

index (IN)

パラメータ・リストから取り出すパラメータを示します。

name (OUT)

現行のパラメータのキー名です。

type (OUT)

現行のパラメータの型です。

OCI_EXTRACT_TYPE_STRING

OCI_EXTRACT_TYPE_INTEGER

OCI_EXTRACT_TYPE_OCINUM

OCI_EXTRACT_TYPE_BOOLEAN

numvals (OUT)

このパラメータの値の数です。

values (OUT)

このパラメータの値です。

コメント

`OCIExtractToList()` は、このルーチンのコールの前にコールして、メモリーに格納されているパラメータ構造からパラメータ・リストを生成する必要があります。

戻り値

`OCI_SUCCESS`

`OCI_INVALID_HANDLE`

`OCI_ERROR`

カートリッジ・サービス – ファイル I/O インタフェース

表 19-5 ファイル I/O インタフェース関数

関数	用途
<code>OCIFileInit()</code> (19-39 ページ)	OCIFile パッケージを初期化します。
<code>OCIFileTerm()</code> (19-40 ページ)	OCIFile パッケージを終了します。
<code>OCIFileOpen()</code> (19-41 ページ)	ファイルをオープンします。
<code>OCIFileClose()</code> (19-43 ページ)	オープンされているファイルをクローズします。
<code>OCIFileRead()</code> (19-44 ページ)	ファイルからバッファにデータを読み込みます。
<code>OCIFileWrite()</code> (19-45 ページ)	<i>buflen</i> バイトをファイルに書き込みます。
<code>OCIFileSeek()</code> (19-46 ページ)	ファイルの現行の位置を変更します。
<code>OCIFileExists()</code> (19-48 ページ)	ファイルの存在を確認するためにテストします。
<code>OCIFileGetLength()</code> (19-49 ページ)	ファイルの長さを取得します。
<code>OCIFileFlush()</code> (19-50 ページ)	バッファ・データをファイルに書き込みます。

関連項目： これらの関数の使用方法の詳細は、『Oracle9i Data Cartridge Developer’s Guide』を参照してください。

OCIFileObject

OCIFileObject データ構造には、ファイルのオープン方法およびオープンしてからそのファイルへアクセスする方法に関する情報が保持されています。この構造を `OCIFileOpen()` で初期化すると、ファイルで実行可能な操作によって識別子となります。このパラメータは、オープンされているファイルで実行するすべての機能に必要です。このデータ構造は OCIFile クライアントには不透明です。これは `OCIFileOpen()` で初期化し、`OCIFileClose()` で終了します。

OCIFileInit()

用途

OCIFile パッケージを初期化します。この関数をコールしてから、他の OCIFile ルーチンをコールする必要があります。

構文

```
sword OCIFileInit( dvoid      *hdl,  
                  OCIError *err );
```

パラメータ

hdl (IN)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、*err* に記録され、OCI_ERROR が戻されます。OCIErrorGet() のコールによって診断情報を取得できます。

戻り値

OCI_SUCCESS

OCI_INVALID_HANDLE

OCI_ERROR

OCIFileTerm()

用途

OCIFile パッケージを終了します。この関数は、OCIFile パッケージを使用しなくなつてからコールする必要があります。

構文

```
sword OCIFileTerm( dvoid      *hdl,  
                   OCIError *err );
```

パラメータ

hdl (IN)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、*err* に記録され、OCI_ERROR が戻されます。OCIErrorGet () のコールによって診断情報を取得できます。

戻り値

OCI_SUCCESS

OCI_INVALID_HANDLE

OCI_ERROR

OCIFileOpen()

用途

ファイルをオープンします。

構文

```
sword OCIFileOpen( dvoid *hndl,
                   OCIError      *err,
                   OCIFileObject **filep,
                   OraText        *filename,
                   OraText        *path,
                   ub4             mode,
                   ub4             create,
                   ub4             type );
```

パラメータ

hndl (IN)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、OCI_ERROR が戻されます。OCIErrorGet () のコールによって診断情報を取得できます。

filep (IN/OUT)

ファイル識別子です。

filename (IN)

ヌル文字で終了する文字列のファイル名です。

path (IN)

ヌル文字で終了する文字列のファイルのパスです。

mode (IN)

ファイルをオープンするモードです。次のモードが有効です。

OCI_FILE_READ_ONLY

OCI_FILE_WRITE_ONLY

OCI_FILE_READ_WRITE

create (IN)

ファイルが存在しない場合に作成するかどうか指定します。有効な値は次のとおりです。

OCI_FILE_TRUNCATE – ファイルをその存在の有無に関係なく作成します。ファイルが存在する場合は、既存のファイルが上書きされます。

OCI_FILE_EXCL – ファイルが存在する場合は失敗し、存在しない場合はファイルが作成されます。

OCI_FILE_CREATE – ファイルが存在する場合はそのファイルをオープンし、存在しない場合はファイルが作成されます。

OCI_FILE_APPEND – 書込みの前にファイルの終わりへのファイル・ポインタを設定します。このフラグは OCI_FILE_CREATE で論理和をとることができます。

type (IN)

ファイルの型。次の値が有効です。

OCI_FILE_TEXT

OCI_FILE_BIN

OCI_FILE_STDIN

OCI_FILE_STDOUT

OCI_FILE_STDERR

戻り値

OCI_SUCCESS

OCI_INVALID_HANDLE

OCI_ERROR

OCIFileClose()

用途

オープンされているファイルをクローズします。

構文

```
sword OCIFileClose( dvoid          *hdl,  
                   OCIError       *err,  
                   OCIFileObject *filep );
```

パラメータ

hdl (IN)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet() のコールによって診断情報を取得できます。

filep (IN/OUT)

クローズされるファイル識別子へのポインタです。

コメント

これが戻されると、`filep` によって指し示された **OCIFileObject** 構造は破棄されます。したがって、この関数が戻されてからこの構造にはアクセスしないでください。

戻り値

OCI_SUCCESS

OCI_INVALID_HANDLE

OCI_ERROR

OCIFileRead()

用途

ファイルからバッファにデータを読み込みます。

構文

```
sword OCIFileRead( dvoid          *hndl,
                   OCIError       *err,
                   OCIFileObject *filep,
                   dvoid          *bufp,
                   ub4            buf1,
                   ub4            *bytesread );
```

パラメータ

hndl (IN)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。 `OCIErrorGet()` のコールによって診断情報を取得できます。

filep (IN/OUT)

ファイルを参照する一意のファイル識別子です。

bufp(IN)

データの読み込み先バッファへのポインタです。 `buf1` のメモリー長が割り当てられるとみなされます。

buf1 (IN)

バイトで示したバッファの長さです。

bytesread (OUT)

読み込みバイト数です。

コメント

可能なバイト数をユーザー・バッファに読み込みます。読み込みはユーザー・バッファがいっぱいになった場合、または EOF に達すると終了します。

戻り値

`OCI_SUCCESS`

`OCI_INVALID_HANDLE`

`OCI_ERROR`

OCIFileWrite()

用途

buflen バイトをファイルに書き込みます。

構文

```
sword OCIFileWrite( dvoid          *hdl,  
                    OCIError      *err,  
                    OCIFileObject *filep,  
                    dvoid         *bufp,  
                    ub4           buflen,  
                    ub4           *byteswritten );
```

パラメータ

hdl (IN)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、*err* に記録され、OCI_ERROR が戻されます。OCIErrorGet() のコールによって診断情報を取得できます。

filep (IN/OUT)

ファイルを参照する一意のファイル識別子です。

bufp(IN)

データの書き込み先バッファへのポインタです。*buflen* のメモリー長が割り当てられるとみなされます。

buflen (IN)

バイトで示したバッファの長さです。

bytesread (OUT)

書き込みバイト数です。

戻り値

OCI_SUCCESS

OCI_INVALID_HANDLE

OCI_ERROR

OCIFileSeek()

用途

ファイルの現行の位置を変更します。

構文

```
sword OCIFileSeek( dvoid          *hndl,
                   OCIError       *err,
                   OCIFileObject *filep,
                   uword          origin,
                   ubig_ora       offset,
                   sb1            dir );
```

パラメータ

hndl (IN)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet() のコールによって診断情報を取得できます。

filep (IN/OUT)

ファイルを参照する一意のファイル識別子です。

origin(IN)

検索の開始地点です。次の地点を開始地点に指定できます。

OCI_FILE_SEEK_BEGINNING (先頭)

OCI_FILE_SEEK_CURRENT (現行の位置)

OCI_FILE_SEEK_END (EOF)

offset (IN)

読み込み開始地点からのバイト数です。

dir (IN)

開始地点から移動する方向です。

注意： 方向は OCIFILE_FORWARD または OCIFILE_BACKWARD のいずれかです。

コメント

この関数では、ファイルの終わりを超えて検索できます。このような位置から読み込みを実行すると、EOF 条件がレポートされます。そのような位置への書き込みは一部のファイル・システムでしか機能しません。これは、ファイルの動的成長が許可されていないシステムがあるためです。このようなシステムでは、ファイルを固定サイズで事前割当てする必要があります。この関数はバイト位置に対して検索を実行するということに注意してください。

戻り値

OCI_SUCCESS

OCI_INVALID_HANDLE

OCI_ERROR

OCIFileExists()

用途

ファイルの存在を確認するためにテストします。

構文

```
sword OCIFileExists( dvoid      *hndl,  
                    OCIError *err,  
                    OraText  *filename,  
                    OraText  *path,  
                    ub1       *flag );
```

パラメータ

hndl (IN)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet() のコールによって診断情報を取得できます。

filename (IN)

ヌル文字で終了する文字列のファイル名です。

path (IN)

ヌル文字で終了する文字列のファイルのパスです。

flag (OUT)

ファイルが存在する場合は TRUE に設定し、存在しない場合は FALSE に設定します。

戻り値

OCI_SUCCESS

OCI_INVALID_HANDLE

OCI_ERROR

OCIFileGetLength()

用途

ファイルの長さを取得します。

構文

```
sword OCIFileGetLength( dvoid      *hndl,
                        OCIError *err,
                        OraText  *filename,
                        OraText  *path,
                        ubig_ora *lenp );
```

パラメータ

hndl (IN)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet () のコールによって診断情報を取得できます。

filename (IN)

ヌル文字で終了する文字列のファイル名です。

path (IN)

ヌル文字で終了する文字列のファイルのパスです。

lenp (OUT)

バイトで示したバッファの長さに設定します。

戻り値

OCI_SUCCESS

OCI_INVALID_HANDLE

OCI_ERROR

OCIFileFlush()

用途

バッファ・データをファイルに書き込みます。

構文

```
sword OCIFileFlush( dvoid          *h  
                    OCIError      *err,  
                    OCIFileObject *filep );
```

パラメータ

hndl (IN)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、OCI_ERROR が戻されます。OCIErrorGet () のコールによって診断情報を取得できます。

filep (IN/OUT)

ファイルを参照する一意のファイル識別子です。

戻り値

OCI_SUCCESS

OCI_INVALID_HANDLE

OCI_ERROR

カートリッジ・サービス – 文字列のフォーマット・インタフェース

表 19-6 文字列のフォーマット関数

関数	用途
<code>OCIFormatInit()</code> (19-52 ページ)	OCIFormat パッケージを初期化します。
<code>OCIFormatTerm()</code> (19-53 ページ)	OCIFormat パッケージを終了します。
<code>OCIFormatString()</code> (19-54 ページ)	テキスト文字列を指定されたテキスト・バッファに書き込みます。

関連項目： これらの関数の使用方法の詳細は、『Oracle9i Data Cartridge Developer’s Guide』を参照してください。

OCIFormatInit()

用途

OCIFormat パッケージを初期化します。

構文

```
sword OCIFormatInit( dvoid      *hndl,  
                    OCIError   *err);
```

パラメータ

hndl (IN)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。`OCIErrorGet()` のコールによって診断情報を取得できます。

コメント

このルーチンをコールしてから、他の `OCIFormat` ルーチンをコールする必要があります。またこのルーチンは 1 度しかコールできません。

戻り値

`OCI_SUCCESS`

`OCI_INVALID_HANDLE`

`OCI_ERROR`

OCIFormatTerm()

用途

OCIFormat パッケージを終了します。

構文

```
sword OCIFormatTerm( dvoid      *hdl,  
                     OCIError *err);
```

パラメータ

hdl (IN)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、*err* に記録され、OCI_ERROR が戻されます。OCIErrorGet() のコールによって診断情報を取得できます。

コメント

この関数は、OCIFormat パッケージを使用しなくなってからコールする必要があります。また、コールは1度しか実行できません。

戻り値

OCI_SUCCESS

OCI_INVALID_HANDLE

OCI_ERROR

OCIFormatString()

用途

送られた引数リストを使用し、指定のフォーマット文字列に従って、指定されたテキスト・バッファにテキスト文字列を書き込みます。

構文

```
sword OCIFormatString( dvoid          *hndl,
                      OCIError      *err,
                      text          *buffer,
                      sbig_ora      bufferLength,
                      sbig_ora      *returnLength,
                      CONST text    *formatString,... );
```

パラメータ

hndl (IN)

OCI 環境ハンドルまたはユーザー・セッション・ハンドルです。

err (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet () のコールによって診断情報を取得できます。

buffer (OUT)

文字列を含むバッファです。

bufferLength (IN)

バイトで示したバッファの長さです。

returnLength (OUT)

バッファに書き込むバイト数です（終了のヌル文字を除く）。

formatString (IN)

リテラル・テキストとフォーマット指定の任意の組合せを指定できるフォーマット文字列です。フォーマット指定は % 文字で区切られ、任意の数（ない場合もあります）のオプションのフォーマット修飾子が続き、必須の書式コードで終了します。フォーマット文字列が % で終わる場合、つまり、後続のフォーマット修飾子やフォーマット指定がない場合、アクションは実行されません。使用可能なフォーマット修飾子および書式コードについては、次に説明します。

...(IN)

フォーム <OCIFormat type wrapper>(<variable>) の引数の変数値です。この <variable> は、使用される値を含む変数にする必要があります。定数または式は OCIFormat type wrappers の引数にはできません。使用可能な OCIFormat type wrappers を次にリストします。引数リストは OCIFormatEnd で終了してください。

```
OCIFormatUb1(ub1 variable);
OCIFormatUb2(ub2 variable);
OCIFormatUb4(ub4 variable);
OCIFormatUword(uword variable);
OCIFormatUbig_ora(ubig_ora variable);
OCIFormatSb1(sb1 variable);
OCIFormatSb2(sb2 variable);
OCIFormatSb4(sb4 variable);
OCIFormatSword(sword variable);
OCIFormatSbig_ora(sbig_ora variable);
OCIFormatEb1(eb1 variable);
OCIFormatEb2(eb2 variable);
OCIFormatEb4(eb4 variable);
OCIFormatEword(eword variable);
OCIFormatChar (text variable);
OCIFormatText(CONST text *variable);
OCIFormatDouble(double variable);
OCIFormatDvoid(CONST dvoid *variable);
OCIFormatEnd
```

コメント

このルーチンの最初のコールを実行する前に、OCIFormatInit ルーチンのコールを実行して OCIFormat パッケージを使用できるように初期化する必要があります。このルーチンが必要なくなった場合は、OCIFormatTerm をコールして OCIFormat パッケージを終了します。

戻り値

```
OCI_SUCCESS
OCI_INVALID_HANDLE
OCI_ERROR
```

フォーマット修飾子

フォーマット修飾子は、より詳細な出力が可能になるようにフォーマット指定を変更または拡張します。フォーマット修飾子は任意の順序で指定でき、すべてオプションです。

フラグ（任意の順序）

フラグ	操作
'-'	フィールド内の出力を左揃えにします。
'+'	数値型の符号（'+'または'-'）を常に出力します。
' '	数値の符号が出力されない場合は、符号の位置が空白で出力されます。
'0'	数値出力に空白ではなく 0（ゼロ）を埋め込みます。

- '+' および ' ' の両方のフラグが同じフォーマット指定で使用されている場合、' ' フラグは無視されます。
- '-' および '0' の両方のフラグが同じフォーマット指定で使用されている場合、'-' フラグは無視されます。

代替出力：

- 8 進書式コードの場合は、先行 0（ゼロ）を追加します。
- 16 進書式コードの場合は、先行 '0x' を追加します。
- 浮動小数点書式コードの場合は、出力で常に基数文字が利用されます。

フィールド幅

<w>。この <w> は最小のフィールド幅を指定する数値です。変換された引数は、この最小幅で出力され、必要に応じてより広い幅にもできます。変換された引数がフィールド幅よりも少ない表示位置を使用する場合は、フィールド幅を調整するために左側（左文字位置の場合は右側）に埋め込まれます。埋込み文字は通常空白ですが、0（ゼロ）の埋込みフラグが指定された場合は 0（ゼロ）になります。特殊文字 '*' は <w> のかわりに使用でき、現行の引数はフィールド幅値のために使用されることを示します。実際のフィールドまたは精度が次の後続引数として続きます。

精度

.<p> は数値 <p> の前にピリオドを指定して、文字列から出力する表示位置の最大数、または 10 進数値の小数点の後の桁、あるいは整数型で出力する場合の最小桁数を指定します (先行 0 (ゼロ) は相違を補うために追加されます)。特殊文字 '*' は、<p> のかわりに使用され、現行の引数に精度値があることを示します。

引数索引

(<n>)。この <n> は、最初の引数が 1 の引数リストに対する整数索引です。引数索引がフォーマット指定で指定されていない場合は、最初の引数が選択されます。次に引数索引をフォーマット指定で指定しないときは 2 番目の引数が選択され、それ以降も同じように続きます。引数索引の使用の有無に関係なくフォーマット指定は任意の順序にすることができ、個別に実行されます。

たとえば、フォーマット文字列 "%u % (4)u %u % (2)u %u" では、OCIFormatString() に指定された 1 番目、4 番目、2 番目、2 番目および 3 番目の引数が選択されます。

書式コード

書式コードは文字列中の引数のフォーマット方法を指定します。

これらの書式コードは大文字にすると、テキスト文字列を除くすべてのアルファベット文字が大文字で出力されます。テキスト文字列には、この変換は行われません。

コード	操作
'c'	コンパイラ・キャラクタ・セット内のシングルスバイト文字。
'd'	符号付き 10 進整数。
'e'	フォーム [-] <d><r> [<d>...] e+ [<d>] <d><d> の指数 (科学) 表記法。この <r> は、現行の言語の基数文字で、<d> は任意の単一桁になります。デフォルトの精度は定数 OCIFormatDP で指定されます。精度はフォーマット修飾子としても指定できます。精度 0 (ゼロ) を使用すると基数文字は示されず、指数は常に 2 桁以上で出力され 3 桁 (たとえば、1e+01、1e+10、1e+100) まで使用できます。
'f'	フォーム [-] <d> [<d>...] <r> [<d>...] の固定 10 進表記法。この <r> は、現行の言語に対して適切な基数文字で、<d> は任意の単一桁です。精度はフォーマット修飾子としても指定できます。精度 0 (ゼロ) を使用すると基数文字は示されません。デフォルトの精度は定数 OCIFormatDP で指定されます。

コード	操作
'g'	変数浮動小数点表記法。'e' または 'f' を選択します。数値が指定された精度に適合する場合は 'f' を選択します（指定されていない場合はデフォルトの精度）。指数フォーマットでより多くの桁を出力できる場合のみ 'e' を選択します。数値に分数部分がない場合は基数文字は出力されません。
'i'	'd' と同一。
'o'	符号なし 8 進整数。
'p'	プラットフォーム固有のポインタ印刷出力。
's'	次の型ではデフォルトの書式コードを使用して引数を入力します。 ocifformatub<n>、ocifformatuword、ocifformatubig_ora、 ocifformatseb<n> および ocifformateword 使用される書式コードは 'u' です。 ocifformatsb<n>、ocifformatsword および ocifformatsbig_ora 使用される書式コードは 'd' です。 ocifformatchar 使用される書式コードは 'c' です。 ocifformattext 後続の NULL が検出されるまでテキストを出力します。 ocifformatdouble 使用される書式コードは 'g' です。 ocifformatdvoid 使用される書式コードは 'p' です。 ' %' - print a '%'
'u'	符号なし 10 進整数。
'x'	符号なし 16 進整数。

例

```
/* This example shows the power of arbitrary argument */
/* selection in the context of internationalization. A */
/* date is formatted in 2 different ways for 2 different */
/* countries according to the format string yet the */
/* argument list submitted to OCIFormatString remains */
/* invariant. */

text      buffer[255];
ub1       day, month, year;
OCLError *err;
dvoid     *hndl;

/* Set the date. */

day       = 10;
month     = 3;
year      = 97;

/* Work out the date in United States' style: mm/dd/yy */
OCIFormatString(hndl, err,
                buffer, (sbig_ora)sizeof(buffer),
                (CONST text *)"% (2) 02u/% (1) 02u/% (3) 02u",
                OCIFormatUb1(day),
                OCIFormatUb1(month),
                OCIFormatUb1(year),
                OCIFormatEnd); /* Buffer is "03/10/97". */

/* Work out the date in New Zealand style: dd/mm/yy */
OCIFormatString(hndl, err,
                buffer, (sbig_ora)sizeof(buffer),
                (CONST text *)"% (1) 02u/% (2) 02u/% (3) 02u",
                OCIFormatUb1(day),
                OCIFormatUb1(month),
                OCIFormatUb1(year),
                OCIFormatEnd); /* Buffer is "10/03/97". */
```

OCI の任意型関数および任意データ関数

この章では、OCI の任意型関数および任意データ関数について説明します。

関連項目： コード例は、Oracle のインストールに含まれているデモ・プログラムを参照してください。追加情報については、[付録 B「OCI デモ・プログラム」](#)を参照してください。

この章は、次の項目で構成されています。

- [任意型インタフェースおよび任意データ・インタフェースの概要](#)
- [OCI の型インタフェース関数](#)
- [OCI 任意データ・インタフェース関数](#)
- [OCI 任意データ・セット・インタフェース関数](#)

任意型インタフェースおよび任意データ・インタフェースの概要

この章では、OCI の任意型関数および任意データ関数の詳細を説明します。

関連項目： この章にリストされている関数の詳細は、11-30 ページの「[AnyType インタフェース](#)、[AnyData インタフェース](#)および [AnyDataSet インタフェース](#)」を参照してください。

関数の構文

関数ごとに次の情報が記載されています。

用途

関数の用途を簡単に説明します。

構文

関数の宣言。

パラメータ

この関数の各パラメータの説明。これにはパラメータのモードが含まれます。パラメータのモードには、次の 3 つの値があります。

モード	説明
IN	Oracle にデータを渡すパラメータ
OUT	このコールまたは後続のコールで Oracle からデータを受け取るパラメータ
IN/OUT	このコールでデータを渡し、このコールまたは後続のコールからの戻りでデータを受け取るパラメータ

コメント

この関数に関する詳細情報。関数の使用上の制約やアプリケーション内でこの関数を使用するときに役に立つ情報が記載されています。コメントの記載がない場合もあります。

この章に記載されているすべての関数は、相互に関連しています。

関数戻り値

OCI の任意型関数および任意データ関数は、通常、次の値のいずれかを返します。

表 20-1 関数戻り値

戻り値	意味
OCI_SUCCESS	操作は成功しました。
OCI_ERROR	操作は失敗しました。関数に渡されたエラー・ハンドルに対して <code>OCIErrorGet()</code> をコールすることで特定のエラーを取り出せます。
OCI_INVALID_HANDLE	関数に渡された OCI ハンドルが無効です。

関連項目： リターン・コードおよびエラー処理の詳細は、2-31 ページの「[エラー処理](#)」を参照してください。

OCI の型インタフェース関数

この項では、型インタフェース関数について説明します。

表 20-2 型インタフェース関数

関数	用途
<code>OCITypeAddAttr()</code> (20-5 ページ)	型コード <code>OCI_TYPECODE_OBJECT</code> を使用して以前作成したオブジェクト型に属性を追加します。
<code>OCITypeBeginCreate()</code> (20-6 ページ)	一時的な型に対する構成プロセスを開始します。この型は無名(名前なし)です。
<code>OCITypeEndCreate()</code> (20-8 ページ)	型の記述の構成を終了します。終了後は、アクセスのみ可能です。
<code>OCITypeSetBuiltin()</code> (20-9 ページ)	組込みの型情報を設定します。このコールは、型が組込みの型コード (<code>OCI_TYPECODE_NUMBER</code> など) を使用して構成されている場合のみ実行されます。
<code>OCITypeSetCollection()</code> (20-10 ページ)	コレクション型情報を設定します。このコールは、型がコレクション型コードを使用して構成されている場合のみ実行されます。

OCITypeAddAttr()

用途

型コード `OCI_TYPECODE_OBJECT` を使用して以前作成したオブジェクト型に属性を追加します。

構文

```
sword OCITypeAddAttr ( OCISvcCtx   *svchp,
                      OCIError    *errhp,
                      OCIType     *type,
                      CONST text   *a_name,
                      ub4          a_length,
                      OCIPParam    *attr_info );
```

パラメータ

svchp (IN)

OCI サービス・コンテキストです。

errhp (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。 `OCIErrorGet()` をコールして診断情報を取得します。

type (IN/OUT)

構成される型の記述です。

a_name (IN)

オプションです。属性の名前です。

a_length (IN)

オプションです。属性名の長さ（バイト単位）です。

attr_info (IN)

属性の情報です。この情報は、**OCIPParam** パラメータ・ハンドルを割り当て、`OCIAttrSet()` コールを使用して型情報を **OCIPParam** に設定して取得します。

OCITypeBeginCreate()

用途

一時的な型に対する構成プロセスを開始します。この型は無名（名前なし）です。

構文

```
sword OCITypeBeginCreate ( OCISvcCtx      *svchp,  
                           OCIError      *errhp,  
                           OCITypeCode   tc,  
                           OCIDuration   dur,  
                           OCIType      **type );
```

パラメータ

svchp (IN)

OCI サービス・コンテキストです。

errhp (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet() をコールして診断情報を取得します。

tc (IN)

型に対する型コードです。この型コードは、オブジェクト型または組込みの型に対応している場合があります。

現在、ユーザー定義型に対して可能な値は次のとおりです。

- OCI_TYPECODE_OBJECT（構成済みのオブジェクト型の場合）
- OCI_TYPECODE_VARRAY（VARRAY コレクション型の場合）
- OCI_TYPECODE_TABLE（NESTED TABLE コレクション型の場合）

オブジェクト型の場合は、OCITypeAddAttr() をコールして、各属性型を追加します。コレクション型の場合は、OCITypeSetCollection() をコールします。次に、OCITypeEndCreate() をコールして、作成プロセスを終了します。

組込みの型コードに対して可能な値は、3-29 ページの「[型コード](#)」で指定されています。組込みの型に関する追加情報（精度、数値のスケール、VARCHAR2 のキャラクタ・セット情報など）がある場合は、後続の OCITypeSetBuiltin() コールで設定してください。最後に、OCITypeEndCreate() を使用して、作成プロセスを終了します。

dur (IN)

型に対する割当て時間です。次のいずれかになります。

- 以前に作成されたユーザー期間。この期間は、`OCIDurationBegin()` を使用して作成できます。
- `OCI_DURATION_SESSION` などの事前定義の期間。

type (OUT)

構成される **OCIType**（型の記述子）です。

コメント

永続的な名前付きの型を作成するには、SQL 文 `CREATE TYPE` を使用してください。一時的な型には識別情報がありません。これらは純粋な値です。

OCITypeEndCreate()

用途

型の記述の構成を終了します。終了後は、アクセスのみ可能です。

構文

```
sword OCITypeEndCreate ( OCISvcCtx   *svchp,  
                        OCIError    *errhp,  
                        OCIType     *type );
```

パラメータ

svchp (IN)

OCI サービス・コンテキストです。

errhp (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet () をコールして診断情報を取得します。

type (IN/OUT)

構成される型の記述です。

OCISetBuiltin()

用途

組込みの型情報を設定します。このコールは、型が組込みの型コード (OCI_TYPECODE_NUMBER など) を使用して構成されている場合のみ実行されます。

構文

```
sword OCISetBuiltin ( OCISvcCtx   *svchp,  
                     OCIError    *errhp,  
                     OCISetBuiltin *type,  
                     OCISetBuiltin *builtin_info );
```

パラメータ

svchp (IN)

OCI サービス・コンテキストです。

errhp (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、*err* に記録され、OCI_ERROR が戻されます。OCIErrorGet() をコールして診断情報を取得します。

type (IN/OUT)

構成される型の記述です。

builtin_info (IN)

組込みに関する情報 (精度、スケール、キャラクタ・セットなど) を提供します。この情報は、**OCISetBuiltin** パラメータ・ハンドルを割り当て、OCIAttrSet() コールを使用して型情報を **OCISetBuiltin** に設定して取得します。

OCICollection()

用途

コレクション型情報を設定します。このコールは、型がコレクション型コードを使用して構成されている場合のみ実行されます。

構文

```
sword OCICollection ( OCISvcCtx   *svchp,
                      OCIError    *errhp,
                      OCICollection *type,
                      OCICollection *collem_info,
                      ub4          coll_count );
```

パラメータ

svchp (IN)

OCI サービス・コンテキストです。

errhp (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、OCI_ERROR が戻されます。OCIErrorGet () をコールして診断情報を取得します。

type (IN/OUT)

構成される型の記述子です。

collem_info (IN)

collem_info によって、コレクション要素に関する情報が提供されます。この情報は、OCIParam パラメータ・ハンドルを割り当て、OCIAttrSet () コールを使用して型情報をOCIParam に設定して取得します。

coll_count (IN)

コレクション内の要素数です。NESTED TABLE (非有界) の場合は、0 (ゼロ) が渡されます。

OCI 任意データ・インタフェース関数

この項では、任意データ・インタフェース関数について説明します。

表 20-3 任意データ関数

関数	用途
OCIAnyDataAccess() (20-12 ページ)	OCIAnyData のデータ値を取り出します。
OCIAnyDataAttrGet() (20-14 ページ)	OCIAnyData の現行の位置にある属性の値を取得します。
OCIAnyDataAttrSet() (20-17 ページ)	現行の位置にある属性に指定の値を設定します。
OCIAnyDataBeginCreate() (20-20 ページ)	指定の継続時間に対して OCIAnyData を割り当て、型情報を使用して初期化します。
OCIAnyDataCollAddElem() (20-22 ページ)	現行の属性位置にある OCIAnyData のコレクション属性に、次のコレクション要素を追加します。
OCIAnyDataCollGetElem() (20-24 ページ)	現行の位置にある OCIAnyData のコレクション属性内の要素に順次アクセスします。
OCIAnyDataConvert() (20-26 ページ)	指定のデータ値を使用して OCIAnyData を構成します。データ値は指定の型になります。
OCIAnyDataDestroy() (20-28 ページ)	AnyData を解放します。
OCIAnyDataEndCreate() (20-29 ページ)	OCIAnyData 作成の終わりをマークします。
OCIAnyDataGetCurrAttrNum() (20-30 ページ)	OCIAnyData の現行の属性番号を戻します。
OCIAnyDataGetType() (20-31 ページ)	AnyData 値に対応する型を取得します。
OCIAnyDataIsNull() (20-32 ページ)	OCIAnyData が NULL かどうかをチェックします。
OCIAnyDataTypeCodeToSqlit() (20-33 ページ)	AnyData 値の OCITypeCode を OCIAnyData API が戻す値の表現に対応する SQLT コードに変換します。

OCIAnyDataAccess()

用途

OCIAnyData のデータ値を取り出します。データ値の型は、**OCIAnyData** が初期化された型である必要があります。このコールを使用して **OCIAnyData** 全体にアクセスでき、その型は、OCI_TYPECODE_OBJECT 型、任意のコレクション型、または任意の組込みの型が可能です。

構文

```
sword OCIAnyDataAccess ( OCISvcCtx      *svchp,
                          OCIError       *errhp,
                          OCIAnyData     *sdata,
                          OCITypeCode    tc,
                          OCIType        *inst_type,
                          dvoid          *null_ind,
                          dvoid          *data_value,
                          ub4            *length );
```

パラメータ

svchp (IN)

OCI サービス・コンテキストです。

errhp (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、OCI_ERROR が戻されます。OCIErrorGet () をコールして診断情報を取得します。

sdata (IN)

OCIAnyData への初期化済みポインタです。

tc (IN)

データ値の型コードです。この型コードを使用して型をチェックします (**OCIAnyData** が初期化された型)。

inst_type (IN)

データ値の **OCIType** です (基本形でない場合)。**tc** パラメータが次のいずれかの場合、このパラメータは NULL にできません。

- OCI_TYPECODE_OBJECT
- OCI_TYPECODE_REF
- OCI_TYPECODE_VARRAY
- OCI_TYPECODE_TABLE

これ以外のパラメータの場合は、NULL でもかまいません。

null_ind (OUT)

`data_value` が NULL かどうかを示します。OCI_TYPECODE_OBJECT 以外のすべての型コードについて、(OCIInd *) を渡します。戻される値は、値が NULL 以外の場合は OCI_IND_NOTNULL に、値が NULL の場合は OCI_IND_NULL になります。型コードが OCI_TYPECODE_OBJECT の場合は、`data_value` のインジケータ構造体へのポインタがここの引数として渡されます。詳細は、「[OCIAnyDataAttrGet\(\)](#)」を参照してください。

data_value (OUT)

データ値です (OCIAnyData の初期化で使用された型になります)。可能な各型コードに対応する適切な C 型とこのパラメータに渡される値に応じたメモリー割当て動作については、「[OCIAnyDataAttrGet\(\)](#)」を参照してください。

length (OUT)

現在、このパラメータは無視されます。今後、このパラメータは、データ表現自体が長さ (バイト単位) を暗黙的に渡さないような特定の型コードに使用される予定です。

OCIAnyDataAttrGet()

用途

OCIAnyData の現行の位置にある属性の値を取得します。属性値には順次アクセスできません。

構文

```
sword OCIAnyDataAttrGet ( OCISvcCtx      *svchp,
                           OCIError       *errhp,
                           OCIAnyData     *sdata,
                           OCITypeCode    tc,
                           OCIType        *attr_type,
                           dvoid          *null_ind,
                           dvoid          *attr_value,
                           ub4             *length,
                           boolean        is_any );
```

パラメータ

svchp (IN)

OCI サービス・コンテキストです。

errhp (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、**OCI_ERROR** が戻されます。OCIErrorGet () をコールして診断情報を取得します。

sdata (IN/OUT)

初期化済みの **OCIAnyData** 型へのポインタです。

tc (IN)

属性の型コードです。型のチェックは、**tc**、**attr_type** および **OCIAnyData** の型情報に基づいて行われます。

attr_type (IN) [オプション]

attr_type は、参照先の型 (**OCI_TYPECODE_REF** の場合)、コレクション型 (**OCI_TYPECODE_VARRAY**、**OCI_TYPECODE_TABLE** の場合) またはオブジェクト (**OCI_TYPECODE_OBJECT** の場合) の型の記述を指定します。組込みの型コードの場合、このパラメータは必要ありません。

null_ind (OUT)

attr_value が NULL かどうかを示します。OCI_TYPECODE_OBJECT 以外のすべての型コードについて、(OCIInd *) を *null_ind* で渡します。

型コードが OCI_TYPECODE_OBJECT の場合は、ポインタ (dvoid **) を *null_ind* で渡します。

戻されるインジケータは、値が NULL 以外の場合は OCI_IND_NOTNULL に、値が NULL の場合は OCI_IND_NULL になります。

attr_value (IN/OUT)

属性の値です。

length (IN/OUT)

現在、このパラメータは無視されます。ここでは、0（ゼロ）を渡してください。今後、このパラメータは、データ表現自体が長さ（バイト単位）を暗黙的に渡さないような特定の型コードに使用される予定です。

is_any (IN)

属性が **OCIAnyData** の形式で戻されるかどうかを示します。

コメント

このコールは、型コードが OCI_TYPECODE_OBJECT の **OCIAnyData** でのみ使用できます。

- このコールは、**OCIAnyData** の現行の位置にある属性の値を取得します。
- *tc* は、現行の位置にある属性の型と一致する必要があります。一致しない場合はエラーが戻されます。
- *is_any* は、属性の型コードが次のいずれかの場合のみ適用できます。
 - OCI_TYPECODE_OBJECT
 - OCI_TYPECODE_VARRAY
 - OCI_TYPECODE_TABLE

is_any が TRUE の場合、*attr_value* は **OCIAnyData*** の形式で戻されます。

- 関数をコールする前に、属性に対してメモリーを割り当てる必要があります。メモリーは OCIObjectNew() を使用して割り当てることができます。NUMBER、VARCHAR などの組込みの型の場合、属性は単なるスタック変数へのポインタである場合があります。次のリストには、オブジェクトの属性型として使用できる Oracle データ型、およびそれに対応して渡される属性値の型が示されています。

データ型	attr_value
VARCHAR2、VARCHAR、CHAR	OCIStrng **
NUMBER、REAL、INT、FLOAT、DECIMAL	OCINumber **
DATE	OCIDate **
TIMESTAMP	OCIDateTime **
TIMESTAMP WITH TIME ZONE	OCIDateTime **
TIMESTAMP WITH LOCAL TIME ZONE	OCIDateTime **
INTERVAL YEAR TO MONTH	OCIInterval **
INTERVAL DAY TO SECOND	OCIInterval **
BLOB	OCILobLocator ** または OCIBlobLocator **
CLOB	OCILobLocator ** または OCIClobLocator *
BFILE	OCILobLocator**
REF	OCISRef**
RAW	OCIRaw**
VARRAY	OCIArray ** (is_any が TRUE の場合は OCIAnyData *)
TABLE	OCITable ** (is_any が TRUE の場合は OCIAnyData *)
OBJECT	dvoid ** (is_any が TRUE の場合は OCIAnyData *)

OCIAnyDataAttrSet()

用途

現行の位置にある属性に指定の値を設定します。

構文

```
sword OCIAnyDataAttrSet ( OCISvcCtx      *svchp,
                          OCIError       *errhp,
                          OCIAnyData     *sdata,
                          OCITypeCode    tc,
                          OCIType       *attr_type,
                          dvoid          *null_ind,
                          dvoid          *attr_value,
                          ub4            length,
                          boolean        is_any );
```

パラメータ

svchp (IN)

OCI サービス・コンテキストです。

errhp (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、*err* に記録され、OCI_ERROR が戻されます。OCIErrorGet() をコールして診断情報を取得します。

sdata (IN/OUT)

初期化済みの OCIAnyData です。

tc (IN)

属性の型コードです。型のチェックは、*tc*、*attr_type* および OCIAnyData の型情報に基づいて行われます。

attr_type (IN)

オブションです。

attr_type は、参照先の型 (OCI_TYPECODE_REF の場合)、コレクション型 (OCI_TYPECODE_VARRAY、OCI_TYPECODE_TABLE の場合) またはオブジェクト (OCI_TYPECODE_OBJECT の場合) の型の記述を指定します。組込みの型コードの場合、または OCI_TYPECODE_NONE が指定されている場合、このパラメータは必要ありません。

null_ind (IN)

attr_value が NULL かどうかを示します。OCI_TYPECODE_OBJECT 以外のすべての型コードについて、(OCIInd *) を渡します。インジケータは、値が NULL 以外の場合は OCI_IND_NOTNULL に、値が NULL の場合は OCI_IND_NULL になります。

型コードが `OCI_TYPECODE_OBJECT` の場合は、引数として `attr_value` のインジケータ構造体へのポインタが渡されます。

attr_value (IN)
属性の値です。

length (IN)
現在、このパラメータは無視されます。ここでは、0（ゼロ）を渡してください。今後、このパラメータは、データ表現自体が長さを暗黙的に渡さないような特定の型コードに使用される予定です。

is_any (IN)
属性が **OCIAnyData** の形式かどうかを示します。

コメント

`OCIAnyDataBeginCreate()` によって、空のスケルトン・インスタンスを持つ **OCIAnyData** が作成されます。属性値を入力するには、`OCIAnyDataAttrSet()` (`OCI_TYPECODE_OBJECT` の場合) または `OCIAnyDataCollAttrAddElem()` (コレクション型コードの場合) を使用します。

属性値は、最初の属性から最後の属性まで順に設定する必要があります。現行の属性番号は、**OCIAnyData** 内部のメンテナンス状態として記憶されています。埋込み属性およびコレクション要素のピース単位の構成はサポートされていません。

このコールによって、現行の位置にある属性に `attr_value` が設定されます。**OCIAnyData** インスタンスに対するピース単位の構成を開始すると、`OCIAnyDataConstruct()` はコールできなくなります。

`tc` は、現行の位置にある属性の型と一致する必要があります。一致しない場合は、エラーが戻されます。

`is_any` が `TRUE` の場合、属性は `OCIAnyData*` の形式であることが必要で、変換されずに、囲まれた **OCIAnyData** (データ) にコピーされます。

次のリストには、オブジェクトの属性型として使用できるデータ型、およびそれに対応して渡される属性値の型が示されています。

データ型	attr_value
VARCHAR2、VARCHAR、CHAR	OCIString *
NUMBER、REAL、INT、FLOAT、DECIMAL	OCINumber*
DATE	OCIDate*
TIMESTAMP	OCIDateTime *
TIMESTAMP WITH TIME ZONE	OCIDateTime *

データ型	attr_value
TIMESTAMP WITH LOCAL TIME ZONE	OCIDateTime *
INTERVAL YEAR TO MONTH	OCIInterval *
INTERVAL DAY TO SECOND	OCIInterval *
BLOB	OCILobLocator * または OCIBlobLocator *
CLOB	OCILobLocator * または OCIClobLocator *
BFILE	OCILobLocator *
REF	OCISRef *
RAW	OCIRaw *
VARRAY	OCIArray * (<i>is_any</i> が TRUE の場合は OCIAnyData *)
TABLE	OCITable * (<i>is_any</i> が TRUE の場合は OCIAnyData *)
OBJECT	dvoid * (<i>is_any</i> が TRUE の場合は OCIAnyData *)

OCIAnyDataBeginCreate()

用途

指定の継続時間に対して **OCIAnyData** を割り当て、型情報を使用して初期化します。

構文

```
sword OCIAnyDataBeginCreate ( OCISvcCtx      *svchp,
                              OCIError       *errhp,
                              OCITypeCode    tc,
                              OCIType       *type,
                              OCIDuration    dur,
                              OCIAnyData     **sdata );
```

パラメータ

svchp (IN)

OCI サービス・コンテキストです。

errhp (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、**OCI_ERROR** が戻されます。OCIErrorGet () をコールして診断情報を取得します。

sdata (IN/OUT)

初期化済みの **OCIAnyData** です。

tc (IN)

OCIAnyData に対応する型コードです。組込みの型コード、または次のようなユーザー定義型の型コードが可能です。

- OCI_TYPECODE_OBJECT
- OCI_TYPECODE_REF
- OCI_TYPECODE_VARRAY

type (IN)

OCIAnyData に対応する型です。型コードが組込みの型 (OCI_TYPECODE_NUMBER など) に対応している場合、このパラメータは NULL にできます。ユーザー定義型 (OCI_TYPECODE_OBJECT、OCI_TYPECODE_REF、コレクション型など) の場合は、NULL 以外になります。

dur (IN)

OCIAnyData が割り当てられる継続時間です。次のいずれかになります。

- 以前に作成されたユーザー期間。この期間は、**OCIDurationBegin()** を使用して作成できます。
- **OCI_DURATION_SESSION** などの事前定義の期間。

sdata (OUT)

初期化済みの **OCIAnyData** です。コールの開始時に (***sdata**) が **NULL** でない場合は、**OCIAnyData** に領域を再割当てするかわりにメモリーを再使用できます。

したがって、初期化していないポインタをここで渡さないでください。

コメント

OCIAnyDataBeginCreate() によって、空のスケルトン・インスタンスを持つ

OCIAnyData が作成されます。属性値を入力するには、**OCIAnyDataAttrSet()**

(**OCI_TYPECODE_OBJECT** の場合) または **OCIAnyDataCollAttrAddElem()** (コレクション型コードの場合) を使用します。

属性値は順に設定する必要があります。最初の属性から最後の属性に順に設定してください。現行の属性番号は、**OCIAnyData** 内部のメンテナンス状態として記憶されています。埋込み属性およびコレクション要素のピース単位の構成はサポートされていません。

パフォーマンス向上のために、**OCIAnyData** は最終的に、渡された **OCIType** パラメータを指し示します。**OCIType** の継続時間が延長されていることの確認は、プログラマの責任で行ってください (**OCIType** が一時パラメータの場合は割当て時間 \geq **OCIAnyData** の継続時間、**OCIType** が永続パラメータの場合は割当て / 確保継続時間 \geq **OCIAnyData** の継続時間)。

OCIAnyDataCollAddElem()

用途

現行の属性位置にある **OCIAnyData** のコレクション属性に、次のコレクション要素を追加します。**OCIAnyData** がコレクション型の場合は、属性位置の概念がなく、このコールでは次のコレクション要素が追加されます。

構文

```
sword OCIAnyDataCollAddElem ( OCISvcCtx      *svchp,
                               OCIError       *errhp,
                               OCIAnyData     *sdata,
                               OCITypeCode    collelem_tc,
                               OCIType       *collelem_type,
                               dvoid          *null_ind,
                               dvoid          *elem_value,
                               ub4            length,
                               boolean        is_any,
                               boolean        last_elem );
```

パラメータ

svchp (IN)

OCI サービス・コンテキストです。

errhp (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、**OCI_ERROR** が戻されます。OCIErrorGet() をコールして診断情報を取得します。

sdata (IN/OUT)

初期化済みの **OCIAnyData** です。

collelem_tc (IN)

追加するコレクション要素の型コードです。型のチェックは、**collelem_tc**、**collelem_type** および **OCIAnyData** の型情報に基づいて行われます。

collelem_type (IN)

オプションです。

collelem_type は、参照先の型 (**OCI_TYPECODE_REF** の場合)、コレクション型 (**OCI_TYPECODE_NAMEDCOLLECTION** の場合) またはオブジェクト (**OCI_TYPECODE_OBJECT** の場合) の型の記述を指定します。

組込みの型コードの場合、このパラメータは必要ありません。

null_ind (IN)

elem_value が NULL かどうかを示します。OCI_TYPECODE_OBJECT 以外のすべての型コードについて、(OCIInd *) を渡します。インジケータは、値が NULL 以外の場合は OCI_IND_NOTNULL に、値が NULL の場合は OCI_IND_NULL になります。

型コードが OCI_TYPECODE_OBJECT の場合は、*elem_value* のインジケータ構造体へのポインタがここの引数として渡されます。

elem_value (IN)

コレクション要素の値です。

length (IN)

コレクション要素の長さです。

is_any (IN)

属性が OCIAnyData の形式かどうかを示します。

last_elem (IN)

追加する要素がコレクション内で最後の要素かどうかを示します。

コメント

このコールは、OCI_TYPECODE_OBJECT 型または任意のコレクション型の OCIAnyData に対して呼び出すことができます。OCIAnyData インスタンスに対するピース単位の構成を開始すると、OCIAnyDataConstruct() はコールできなくなります。

OCIAnyDataAttrSet() と同様に、*is_any* は、*collelem_tc* の型コードが OCI_TYPECODE_OBJECT またはコレクション型コードの場合のみ適用できます。*is_any* が TRUE の場合、属性は OCIAnyData* の形式であることが必要です。

追加する要素がコレクション内で最後の要素の場合は、*last_elem* を TRUE に設定する必要があります。

NULL 要素を追加するには、NULL インジケータの *null_ind* を OCI_IND_NULL に設定します。この場合、他のすべての引数は無視されます。それ以外の場合、*null_ind* は OCI_IND_NOTNULL に設定する必要があります。

可能なすべての型のコレクション要素に渡される属性の型については、「OCIAnyDataAttrSet()」を参照してください。

OCIAnyDataCollGetElem()

用途

現行の位置にある **OCIAnyData** のコレクション属性内の要素に順次アクセスします。

構文

```
sword OCIAnyDataCollGetElem ( OCISvcCtx      *svchp,
                              OCIError       *errhp,
                              OCIAnyData     *sdata,
                              OCITypeCode    collelem_tc,
                              OCIType       *collelem_type,
                              dvoid          *null_ind,
                              dvoid          *collelem_value,
                              ub4            *length,
                              boolean        is_any );
```

パラメータ

svchp (IN)

OCI サービス・コンテキストです。

errhp (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。 `OCIErrorGet()` をコールして診断情報を取得します。

sdata (IN/OUT)

初期化済みの **OCIAnyData** です。

collelem_tc (IN)

取り出すコレクション要素の型コードです。型のチェックは、`collelem_tc`、`collelem_type` および **OCIAnyData** の型情報に基づいて行われます。

collelem_type (IN)

オブションです。

`collelem_type` は、参照先の型 (`OCI_TYPECODE_REF` の場合)、コレクション型 (`OCI_TYPECODE_NAMEDCOLLECTION` の場合) またはオブジェクト (`OCI_TYPECODE_OBJECT` の場合) の型の記述を指定します。

組込みの型コードの場合、このパラメータは必要ありません。

null_ind (OUT)

collelem_value が NULL かどうかを示します。OCI_TYPECODE_OBJECT 以外のすべての型コードについて、(OCIInd *) を渡します。インジケータは、値が NULL 以外の場合は OCI_IND_NOTNULL に、値が NULL の場合は OCI_IND_NULL になります。

型コードが OCI_TYPECODE_OBJECT の場合は、*collelem_value* のインジケータ構造体へのポインタ (dvoid **) がここの引数として渡されます。

collelem_value (IN/OUT)

コレクション要素の値です。

length (IN/OUT)

コレクション要素の長さです。現在は無視されます。入力時には、0 (ゼロ) を設定してください。

is_any (IN)

attr_value が **OCIAnyData** の形式で戻されるかどうかを示します。

コメント

OCIAnyData データは、トップレベルのコレクションに対応させることもできます。

OCIAnyData が OCI_TYPECODE_OBJECT 型の場合、現行の位置の属性は適切な型のコレクションである必要があります。それ以外の場合はエラーが戻されます。

OCIAnyDataAttrGet() と同様に、*is_any* パラメータは、*collelem_tc* の型コードが OCI_TYPECODE_OBJECT である場合のみ適用されます。*is_any* が TRUE の場合、*attr_value* は **OCIAnyData *** の形式である必要があります。

このコールは、コレクションの終わりに達すると **OCI_NO_DATA** を戻します。成功時には **OCI_SUCCESS** を戻し、エラーが発生した場合は **OCI_ERROR** を戻します。

可能なすべての型のコレクション要素に渡される属性の型については、[「OCIAnyDataAttrGet\(\)」](#) を参照してください。

OCIAnyDataConvert()

用途

指定のデータ値を使用して **OCIAnyData** を構成します。データ値は指定の型になります。このコールを使用すると、OCI_TYPECODE_OBJECT 型、任意のコレクション型または任意の組込みの型など、**OCIAnyData** 全体を構成できます。

構文

```
sword OCIAnyDataConvert ( OCISvcCtx      *svchp,
                          OCIError       *errhp,
                          OCITypeCode    tc,
                          OCIType        *inst_type,
                          OCIDuration    dur,
                          dvoid          *null_ind,
                          dvoid          *data_value,
                          ub4            length,
                          OCIAnyData     **sdata );
```

パラメータ

svchp (IN)

OCI サービス・コンテキストです。

errhp (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、OCI_ERROR が戻されます。OCIErrorGet () をコールして診断情報を取得します。

tc (IN)

データ値の型コードです。組込みの型コードまたはユーザー定義型の型コード (OCI_TYPECODE_OBJECT、OCI_TYPECODE_REF、OCI_TYPECODE_VARRAY など) が可能です。

(*sdata) が NULL 以外で、[OCIAnyDataSetAddInstance\(\)](#) のコール時に戻されたスケルトン・インスタンスを表す場合、tc パラメータおよび inst_type パラメータは、このオプションになります。これは、このようなスケルトン・インスタンスの型情報はすでに認識されているためです。この状況で、tc パラメータと inst_type パラメータがここで指定されると、これらのパラメータは型のチェックにのみ使用されます。

inst_type (IN)

OCIAnyData に対応する型です。型コードが組込みの型 (OCI_TYPECODE_NUMBER など) に対応している場合、このパラメータは NULL にできます。ユーザー定義型 (OCI_TYPECODE_OBJECT、OCI_TYPECODE_REF、コレクション型など) の場合は、NULL 以外になります。

dur (IN)

OCIAnyData が割り当てられる継続時間です。次のいずれかになります。

- 以前に作成されたユーザー期間。この期間は、**OCIDurationBegin()** を使用して作成できます。
- **OCI_DURATION_SESSION** などの事前定義の期間。

null_ind

data_value が NULL かどうかを示します。**OCI_TYPECODE_OBJECT** 以外のすべての型コードについて、(**OCIInd ***) を渡します。インジケータは、値が NULL 以外の場合は **OCI_IND_NOTNULL** に、値が NULL の場合は **OCI_IND_NULL** になります。

型コードが **OCI_TYPECODE_OBJECT** の場合は、*data_value* のインジケータ構造体へのポインタがこの引数として渡されます。

data_value (IN)

データ値です。(OCIAnyData の初期化で使用された型です)。可能な各型コードに対応する適切な C 型については、「**OCIAnyDataAttrSet()**」を参照してください。

length (IN)

現在、このパラメータは無視されます。ここでは、0 (ゼロ) を渡してください。今後、このパラメータは、データ表現自体が長さを暗黙的に渡さないような特定の型コードに使用される予定です。

sdata (IN/OUT)

初期化済みの **OCIAnyData** です。コールの開始時に (**sdata*) が NULL でない場合は、**OCIAnyData** に領域を再割当てするかわりにメモリーを再使用できます。

したがって、初期化していないポインタをここで渡さないでください。

(**sdata*) が **OCIAnyDataSetAddInstance()** のコール時に戻されたスケルトン・インスタンスを表す場合、*tc* パラメータおよび *inst_type* パラメータは、必要に応じて型のチェックに使用されます。

コメント

パフォーマンス向上のため、**OCIAnyData** は最終的に、渡された **OCIType** パラメータを指し示します。**OCIType** の継続時間が延長されていることの確認は、プログラマの責任で行ってください (**OCIType** が一時パラメータの場合は割当て時間 \geq **OCIAnyData** の継続時間、**OCIType** が永続パラメータの場合は割当て / 確保継続時間 \geq **OCIAnyData** の継続時間)。

OCIAnyDataDestroy()

用途

AnyData を解放します。

構文

```
sword OCIAnyDataDestroy ( OCISvcCtx      *svchp,  
                           OCIError      *errhp,  
                           OCIAnyData     *sdata );
```

パラメータ

svchp (IN)

OCI サービス・コンテキストです。

errhp (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet () をコールして診断情報を取得します。

sdata (IN/OUT)

解放する **OCIAnyData** 型へのポインタです。

OCIAnyDataEndCreate()

用途

OCIAnyData 作成の終わりをマークします。この関数は、インスタンスのすべての属性を適切な値で初期化してからコールしてください。このコールは、**OCIAnyData** に対して **OCIAnyDataBeginCreate()** がすでにコールされている場合のみ有効です。

構文

```
sword OCIAnyDataEndCreate ( OCISvcCtx      *svchp,  
                             OCIError      *errhp,  
                             OCIAnyData    *data );
```

パラメータ

svchp (IN)

OCI サービス・コンテキストです。

errhp (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、**OCI_ERROR** が戻されます。 **OCIErrorGet()** をコールして診断情報を取得します。

data (IN/OUT)

初期化済みの **OCIAnyData** です。

OCIAnyDataGetCurrAttrNum()

用途

OCIAnyData の現行の属性番号を返します。**OCIAnyData** を構成する場合、この関数は設定する現行の属性を参照します。また、**OCIAnyData** にアクセスする場合は、アクセスする属性を参照します。

構文

```
sword OCIAnyDataGetCurrAttrNum( OCISvcCtx      *svchp,  
                                OCIError       *errhp,  
                                OCIAnyData     *sdata,  
                                ub4            *attrnum );
```

パラメータ

svchp (IN)

OCI サービス・コンテキストです。

errhp (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、**OCI_ERROR** が戻されます。**OCIErrorGet()** をコールして診断情報を取得します。

sdata (IN)

初期化済みの **OCIAnyData** です。

attrnum (OUT)

属性番号です。

OCIAnyDataGetType()

用途

AnyData 値に対応する型を取得します。この関数は、**OCIAnyData** の内部でメンテナンスされている型への実際のポインタを戻します。パフォーマンス向上のため、コピーは行われません。**OCIAnyData** を解放した後（または継続時間が終了した後）、この型を使用しないようにすることは、プログラマの責任で行ってください。

構文

```
sword OCIAnyDataGetType( OCISvcCtx      *svchp,  
                          OCIError      *errhp,  
                          OCIAnyData     *data,  
                          OCITypeCode    *tc,  
                          OCIType       **type );
```

パラメータ

svchp (IN)

OCI サービス・コンテキストです。

errhp (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、**OCI_ERROR** が戻されます。OCIErrorGet () をコールして診断情報を取得します。

data (IN)

初期化済みの **OCIAnyData** です。

tc (OUT)

OCIAnyData に対応する型コードです。

type (OUT)

OCIAnyData に対応する型です。**OCIAnyData** が組込みの型に対応している場合は、NULL になります。

OCIAnyDataIsNull()

用途

OCIAnyData 内の型の内容が NULL かどうかをチェックします。

構文

```
sword OCIAnyDataIsNull ( OCISvcCtx      *svchp,  
                          OCIError      *errhp,  
                          CONST OCIAnyData *sdata,  
                          boolean        *isNull) ;
```

パラメータ

svchp (IN)

OCI サービス・コンテキストです。

errhp (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、*err* に記録され、OCI_ERROR が戻されます。OCIErrorGet() をコールして診断情報を取得します。

sdata (IN)

チェックする **OCIAnyData** です。

isNull (IN/OUT)

NULL の場合は TRUE、それ以外の場合は FALSE です。

OCIAnyDataTypeCodeToSqlt()

用途

AnyData 値の **OCITypeCode** を OCIAnyData API が戻す値の表現に対応する SQLT コードに変換します。

構文

```
sword OCIAnyDataTypeCodeToSqlt ( OCIError      *errhp,  
                                OCITypeCode   tc,  
                                ub1            *sqltcode,  
                                ub1            *csfrm) ;
```

パラメータ

errhp (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、errhp に記録され、OCI_ERROR が戻されます。OCIErrorGet() のコールによって診断情報を取得できます。

tc (IN)

AnyData 値に対応する **OCITypeCode** です。

sqltcode (OUT)

型コードのユーザー・フォーマットに対応する SQLT コードです。

csfrm (OUT)

型コードのユーザー・フォーマットに対応するキャラクタ・セット・フォームです。キャラクタ・タイプの場合のみ有効です。(NCHAR 型の) SQLCS_IMPLICIT または SQLCS_NCHAR を戻します。

コメント

この関数は、OCI_TYPECODE_VARCHAR2 および OCI_TYPECODE_CHAR を、SQLCS_IMPLICIT のキャラクタ・セット・フォームで SQLT_VST (**OCIString** マッピングに相当する) に変換します。また、OCI_TYPECODE_NVARCHAR2 は、SQLCS_NCHAR のキャラクタ・セット・フォームで SQLT_VST (**OCIString** マッピングは、OCIAnyData API が使用) を戻します。

関連項目： 詳細は、11-34 ページの「[OCIAnyData 関数の NCHAR 型コード](#)」を参照してください。

OCI 任意データ・セット・インタフェース関数

この項では、任意データ・セット・インタフェース関数について説明します。

表 20-4 任意データ・セット関数

関数	用途
OCIAnyDataSetAddInstance() (20-35 ページ)	新規のスケルトン・インスタンスを OCIAnyDataSet に追加し、そのインスタンスのすべての属性を NULL に設定します。
OCIAnyDataSetBeginCreate() (20-37 ページ)	指定の継続時間に対して OCIAnyDataSet を割り当て、型情報を使用して初期化します。
OCIAnyDataSetDestroy() (20-39 ページ)	OCIAnyDataSet を解放します。
OCIAnyDataSetEndCreate() (20-40 ページ)	OCIAnyDataSet 作成の終わりをマークします。
OCIAnyDataSetGetCount() (20-41 ページ)	OCIAnyDataSet 内のインスタンス数を取得します。
OCIAnyDataSetGetInstance() (20-42 ページ)	現行の位置にあるインスタンスに対応する OCIAnyData を戻し、現行の位置を更新します。
OCIAnyDataSetGetType() (20-43 ページ)	OCIAnyDataSet に対応する型を取得します。

OCIAnyDataSetAddInstance()

用途

新規のスケルトン・インスタンスを **OCIAnyDataSet** に追加し、そのインスタンスのすべての属性を NULL に設定します。

構文

```
sword OCIAnyDataSetAddInstance ( OCISvcCtx      *svchp,
                                OCIError       *errhp,
                                OCIAnyDataSet  *data_set,
                                OCIAnyData     **data );
```

パラメータ

svchp (IN)

OCI サービス・コンテキストです。

errhp (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、*err* に記録され、OCI_ERROR が戻されます。OCIErrorGet() をコールして診断情報を取得します。

data_set (IN/OUT)

新規インスタンスを追加する **OCIAnyDataSet** です。

data (IN/OUT)

新規に追加するインスタンスに対応する **OCIAnyData** です。(*data) が NULL の場合、新規の **OCIAnyData** は **OCIAnyDataSet** と同じ継続時間の間割り当てられます。(*data) が NULL 以外の場合は再使用されます。この **OCIAnyData** は、後で OCIAnyDataConvert() コールを使用して構成したり、OCIAnyDataAttrSet() コールまたは OCIAnyDataCollAddElem() コールを使用してピース単位で構成することができます。

コメント

このコールは、このスケルトン・インスタンスを **OCIAnyData** パラメータを介して戻します。このインスタンスは、後で OCIAnyData API を呼び出して構成できます。

注意： ここでは古い値は破棄されません。これら一連のコールを開始する前に、(*data) が指し示す古い値を破棄し、(*data) を NULL ポインタに設定することは、プログラマの責任で行ってください。戻された **OCIAnyData** では、(OCIType 情報またはデータ部分の) ディープ・コピーは行われません。この **OCIAnyData** は、**OCIAnyDataSet** の割当て時間を超えて使用することはできません (**OCIAnyDataSet** への参照と同様です)。戻された **OCIAnyData** をこの関数に対する後続のコールで再使用すると、新規のデータ・インスタンスを **OCIAnyDataSet** に後で追加できます。

OCIAnyDataSetBeginCreate()

用途

指定の継続時間に対して **OCIAnyDataSet** を割り当て、型情報を使用して初期化します。
OCIAnyDataSet は、指定の型のインスタンスを複数保持できます。

構文

```
sword OCIAnyDataSetBeginCreate ( OCISvcCtx      *svchp,
                                OCIError       *errhp,
                                OCITypeCode    typecode,
                                CONST OCIType   *type,
                                OCIDuration    dur,
                                OCIAnyDataSet  **data_set );
```

パラメータ

svchp (IN)

OCI サービス・コンテキストです。

errhp (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、OCI_ERROR が戻されます。OCIErrorGet () をコールして診断情報を取得します。

typecode (IN)

OCIAnyDataSet に対応する型コードです。

type (IN)

OCIAnyDataSet に対応する型です。型コードが組込みの型 (OCI_TYPECODE_NUMBER など) に対応している場合、このパラメータは NULL にできます。ユーザー定義型 (OCI_TYPECODE_OBJECT、OCI_TYPECODE_REF、コレクション型など) の場合は、NULL 以外になります。

dur (IN)

OCIAnyDataSet が割り当てられる継続時間です。次のいずれかになります。

- 以前に作成されたユーザー期間。この期間は、OCIDurationBegin () を使用して作成できます。
- OCI_DURATION_SESSION などの事前定義の期間。

data_set (OUT)

初期化済みの **OCIAnyDataSet** です。

コメント

パフォーマンス向上のため、**OCIAnyDataSet** は最終的に、渡された **OCIType** パラメータを指し示します。**OCIType** の継続時間が延長されていることの確認は、プログラマの責任で行ってください（**OCIType** が一時パラメータの場合は割当て時間 \geq **OCIAnyData** の継続時間、**OCIType** が永続パラメータの場合は割当て / 確保継続時間 \geq **OCIAnyData** の継続時間）。

OCIAnyDataSetDestroy()

用途

OCIAnyDataSet を解放します。

構文

```
sword OCIAnyDataSetDestroy ( OCISvcCtx      *svchp,  
                             OCIError       *errhp,  
                             OCIAnyDataSet  *data_set );
```

パラメータ

svchp (IN)

OCI サービス・コンテキストです。

errhp (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、OCI_ERROR が戻されます。OCIErrorGet () をコールして診断情報を取得します。

data_set (IN/OUT)

解放される OCIAnyDataSet です。

OCIAnyDataSetEndCreate()

用途

OCIAnyDataSet 作成の終わりをマークします。この関数は、すべてのインスタンスを構成した後にコールしてください。

構文

```
sword OCIAnyDataSetEndCreate ( OCISvcCtx      *svchp,  
                                OCIError       *errhp,  
                                OCIAnyDataSet  *data_set );
```

パラメータ

svchp (IN)

OCI サービス・コンテキストです。

errhp (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、*err* に記録され、OCI_ERROR が戻されます。OCIErrorGet() をコールして診断情報を取得します。

data_set (IN/OUT)

初期化済みの **OCIAnyDataSet** です。

OCIAnyDataSetGetCount()

用途

OCIAnyDataSet のインスタンス数を取得します。

構文

```
sword OCIAnyDataSetGetCount ( OCISvcCtx      *svchp,  
                               OCIError       *errhp,  
                               OCIAnyDataSet  *data_set,  
                               ub4            *count );
```

パラメータ

svchp (IN)

OCI サービス・コンテキストです。

errhp (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、**err** に記録され、**OCI_ERROR** が戻されます。OCIErrorGet () をコールして診断情報を取得します。

data_set (IN/OUT)

正しい形式の **OCIAnyDataSet** です。

count (OUT)

OCIAnyDataSet 内のインスタンス数です。

OCIAnyDataSetGetInstance()

用途

現行の位置にあるインスタンスに対応する **OCIAnyData** を戻し、現行の位置を更新します。

構文

```
sword OCIAnyDataSetGetInstance ( OCISvcCtx      *svchp,  
                                OCIError       *errhp,  
                                OCIAnyDataSet  *data_set,  
                                OCIAnyData     **data );
```

パラメータ

svchp (IN)

OCI サービス・コンテキストです。

errhp (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、*err* に記録され、OCI_ERROR が戻されます。OCIErrorGet() をコールして診断情報を取得します。

data_set (IN/OUT)

正しい形式の **OCIAnyDataSet** です。

data (IN/OUT)

インスタンスに対応する **OCIAnyData** です。(*data) が NULL の場合、新規の **OCIAnyData** は **OCIAnyDataSet** と同じ継続時間の間割り当てられます。(*data) が NULL 以外の場合は再使用されます。

コメント

OCIAnyDataSet 内のインスタンスには、順次アクセスのみ可能です。このコールは、現行の位置にあるインスタンスに対応する **OCIAnyData** を戻し、現行の位置を更新します。後で、OCIAnyData アクセス・ルーチンを使用して、インスタンスにアクセスできます。

OCIAnyDataSetGetType()

用途

OCIAnyDataSet に対応する型を取得します。

構文

```
sword OCIAnyDataSetGetType ( OCISvcCtx      *svchp,  
                             OCIError       *errhp,  
                             OCIAnyDataSet  *data_set,  
                             OCITypeCode    *tc,  
                             OCIType       **type );
```

パラメータ

svchp (IN)

OCI サービス・コンテキストです。

errhp (IN/OUT)

OCI エラー・ハンドルです。エラーがある場合は、`err` に記録され、`OCI_ERROR` が戻されます。OCIErrorGet() をコールして診断情報を取得します。

data_set (IN)

初期化済みの OCIAnyDataSet です。

tc (OUT)

OCIAnyDataSet の型に対応する型コードです。

type (OUT)

OCIAnyDataSet に対応する型です。OCIAnyData が組込みの型に対応している場合は、NULL になります。

第 IV 部

付録

第 IV 部は、付録で構成されています。

- [付録 A「ハンドルおよび記述子の属性」](#) には、各種 OCI ハンドルの属性のリストを示します。
- [付録 B「OCI デモ・プログラム」](#) には、OCI 機能のコード例の重要なデモ・プログラムのリストを示します。
- [付録 C「OCI 関数のサーバー・ラウンドトリップ」](#) では、ほとんどの OCI 関数に必要なサーバー・ラウンドトリップに関して説明します。

ハンドルおよび記述子の属性

この付録では、OCI のハンドルおよび記述子の属性について説明します。この属性は `OCIAttrGet()` 関数で読み込み、`OCIAttrSet()` 関数で変更することができます。

- 規則
- 環境ハンドル属性
- エラー・ハンドル属性
- サービス・コンテキスト・ハンドル属性
- サーバー・ハンドル属性
- ユーザー・セッション・ハンドル属性
- 接続プール・ハンドル属性
- セッション・プール・ハンドル属性
- トランザクション・ハンドル属性
- 文ハンドル属性
- バインド・ハンドル属性
- 定義ハンドル属性
- 記述ハンドル属性
- パラメータ記述子属性
- LOB ロケータ属性
- 複合オブジェクト属性
- アドバンスド・キューイング記述子の属性
- サブスクリプション・ハンドル属性
- ダイレクト・パス・ロード・ハンドル属性
- プロセス・ハンドル属性

規則

それぞれのハンドル・タイプで読み込みまたは変更が可能な属性をリストします。各属性リストには、次の情報が含まれています。

モード

有効なモードは次の 3 つです。

読み込み — `OCIAttrGet()` を使用して属性を読み込みます。

書き込み — `OCIAttrSet()` を使用して属性を変更します。

読み込み / 書き込み — `OCIAttrGet()` を使用して属性を読み込み、`OCIAttrSet()` を使用して属性を変更します。

説明

これは、属性の用途の説明です。

属性データ型

これは、属性のデータ型です。必要に応じて、読み込みモードと書き込みモードのデータ型が区別されます。

有効な値

一定の値のみが許可される場合には、その値を記載します。

例

一部の属性については例が記載されています。

環境ハンドル属性

OCI_ATTR_BIND_DN

モード

読み込み / 書き込み

説明

LDAP サーバーへの接続時に使用するログイン名 (DN) です。

属性データ型

text *

OCI_ATTR_CACHE_ARRAYFLUSH

モード

読み込み / 書き込み

説明

この属性が TRUE に設定されている場合は、`OCICacheFlush()` のコールによって、共通の表に属するオブジェクトと一緒にフラッシュされるため、パフォーマンスを大幅に向上することができます。このモードは、オブジェクトのフラッシュの順序が重要でない場合にのみ使用します。このモードの間は、オブジェクトに使用済みマークが設定される順序が保持される保証はありません。

関連項目： 13-5 ページ「オブジェクト・キャッシュ・パラメータ」および 13-11 ページ「変更をサーバーにフラッシュ」

属性データ型

boolean

OCI_ATTR_CACHE_MAX_SIZE

モード

読み込み / 書き込み

説明

最適サイズのパーセンテージとして、クライアント側オブジェクト・キャッシュの最大サイズ（最高水位標）を設定します。値を最適サイズ（OCI_ATTR_CACHE_OPT_SIZE）の110% に設定します。オブジェクト・キャッシュは最大の最適値を使用して、オブジェクト・キャッシュ内の未使用メモリーを解放します。

関連項目： 13-5 ページ「オブジェクト・キャッシュ・パラメータ」

属性データ型

ub4 *

OCI_ATTR_CACHE_OPT_SIZE

モード

読み込み / 書き込み

説明

クライアント側オブジェクト・キャッシュの最適サイズをバイト数で設定します。デフォルト値は 8MB です。

関連項目： 13-5 ページ「オブジェクト・キャッシュ・パラメータ」

属性データ型

ub4 *

OCI_ATTR_ENV_CHARSET_ID

モード

読み込み

説明

ローカル（クライアント側）キャラクタ・セット ID です。ユーザーは、環境ハンドルの作成後にこの設定を更新できますが、他の OCI 関数をコールする前に行う必要があります。この制限によって、同じ環境ハンドル内のデータおよびメタデータ間の整合性が保証されます。UTF-16 モードでは、この属性を取得できません。

属性データ型

ub2 *

OCI_ATTR_ENV_NCHARSET_ID

モード

読み込み

説明

ローカル（クライアント側）各国語キャラクタ・セット ID です。ユーザーは、環境ハンドルの作成後にこの設定を更新できますが、他の OCI 関数をコールする前に行う必要があります。この制限によって、同じ環境ハンドル内のデータおよびメタデータ間の整合性が保証されます。UTF-16 モードでは、この属性を取得できません。

属性データ型

ub2 *

OCI_ATTR_ENV_UTF16

モード

読み込み

説明

エンコーディング方法は UTF-16 です。値 1 は環境ハンドルが UTF-16 モードで作成されたことを示し、値 0 はそれ以外であることを示します。このモードは OCIEnvCreate() のコールでのみ設定でき、後で変更することはできません。

属性データ型

ub1 *

OCI_ATTR_LDAP_AUTH

モード

読み込み / 書き込み

説明

認証モードです。次に示す値が有効です。

- 0x0: 認証なし。匿名のバインドです。
- 0x1: 簡易認証。ユーザー名 / パスワードによる認証です。
- 0x5: 認証なしの SSL 接続。
- 0x6: SSL: サーバー認証のみ必要です。

0x7: SSL: サーバー認証とクライアント認証の両方が必要です。

0x8: 認証方式は実行時に判断されます。

属性データ型

ub2

OCI_ATTR_LDAP_CRED

モード

読み込み / 書き込み

説明

認証方式が簡易認証（ユーザー名 / パスワードによる認証）の場合、この属性には、LDAP サーバーへの接続時に使用するパスワードが保持されます。

属性データ型

text *

OCI_ATTR_LDAP_CTX

モード

読み込み / 書き込み

説明

クライアントの管理コンテキストです。通常は、LDAP サーバー内の Oracle RDBMS LDAP スキーマのルートです。

属性データ型

text *

OCI_ATTR_LDAP_HOST

モード

読み込み / 書き込み

説明

LDAP サーバーが実行されているホスト名です。

属性データ型

text *

OCI_ATTR_LDAP_PORT

モード

読込み / 書込み

説明

LDAP サーバーがリスニングするポートです。

属性データ型

ub2

OCI_ATTR_OBJECT

モード

読込み

説明

環境がオブジェクト・モードで初期化されている場合は、TRUE を返します。

属性データ型

boolean *

OCI_ATTR_PINOPTION

モード

読込み / 書込み

説明

この属性は、環境ハンドルに対応付けられたアプリケーションに OCI_PIN_DEFAULT の値を設定します。

たとえば、OCI_ATTR_PINOPTION が OCI_PIN_RECENT に設定されている場合に、pin_option パラメータが OCI_PIN_DEFAULT に設定されている *OCIObjectPin()* をコールすると、オブジェクトは OCI_PIN_RECENT モードで確保されます。

属性データ型

OCIPinOpt *

OCI_ATTR_ALLOC_DURATION

モード

読み込み / 書き込み

説明

この属性は、環境ハンドルに対応付けられたアプリケーションの割当て時間に OCI_DURATION_DEFAULT の値を設定します。

属性データ型

OCIDuration *

OCI_ATTR_PIN_DURATION

モード

読み込み / 書き込み

説明

この属性は、環境ハンドルに対応付けられたアプリケーションの確保継続時間に OCI_DURATION_DEFAULT の値を設定します。

属性データ型

OCIDuration *

OCI_ATTR_HEAPALLOC

モード

読み込み

説明

環境ハンドルから割り当てられたメモリーのカレント・サイズです。これは、アプリケーション内でメモリーが最も使用されている部分を追跡するのに役立ちます。

属性データ型

ub4 *

OCI_ATTR_OBJECT_NEWNOTNULL

モード

読み込み / 書き込み

説明

この属性が TRUE に設定されている場合は、新規に作成されるオブジェクトに NULL 以外の属性が設定されます。

関連項目： 10-32 ページ「オブジェクトの作成」

属性データ型

boolean *

OCI_ATTR_OBJECT_DETECTCHANGE

モード

読み込み / 書き込み

説明

この属性が TRUE に設定されている場合は、コミットされた別のトランザクションによってサーバー内で変更されたオブジェクトのフラッシュを試行すると、アプリケーションが ORA-08179 エラーを受け取ります。

関連項目： 13-14 ページ「コミット時ロックの実装」

属性データ型

boolean *

OCI_ATTR_SHARED_HEAPALLOC

モード

読み込み

説明

共有プールから現在割り当てられているメモリのサイズを戻します。この属性はどの環境ハンドルでも使用できますが、有効な値を戻すには、プロセスを共有モードで初期化する必要があります。この属性は次のように読み込まれます。

```
ub4 heapsz = 0;
OCIAttrGet((dvoid *)envhp, (ub4)OCI_HTYPE_ENV,
            (dvoid *) &heapsz, (ub4 *) 0,
            (ub4)OCI_ATTR_SHARED_HEAPALLOC, errhp);
```

属性データ型

ub4 *

OCI_ATTR_WALL_LOC

モード
読み込み / 書き込み

説明
認証方式が SSL 認証の場合、この属性にはクライアント Wallet の位置が格納されます。

属性データ型
text *

エラー・ハンドル属性

OCI_ATTR_DML_ROW_OFFSET

モード
読み込み

説明
エラーが発生した（DML 配列の）オフセットを戻します。

属性データ型
ub4 *

サービス・コンテキスト・ハンドル属性

OCI_ATTR_ENV

モード

読み込み

説明

サービス・コンテキストに対応付けられた環境コンテキストを返します。

属性データ型

OCIEnv **

OCI_ATTR_IN_V8_MODE

モード

読み込み

説明

アプリケーションが (OCISvcCtxToLda()) コールなどによって) Oracle バージョン 7 モードに切り替えられているかどうかを判断できます。戻り値が 0 (ゼロ) 以外 (TRUE) の場合、アプリケーションは現在 Oracle バージョン 8 モードで実行されており、戻り値が 0 (ゼロ) (FALSE) の場合、アプリケーションは現在 Oracle バージョン 7 モードで実行されています。

属性データ型

ub1 *

例

次のコード例で、このパラメータの使用方法を示します。

```
in_v8_mode = 0;
OCIAttrGet ((dvoid *)svchp, (ub4)OCI_HTYPE_SVCCTX, (ub1 *)&in_v8_mode,
            (ub4) 0, OCI_ATTR_IN_V8_MODE, errhp);
if (in_v8_mode)
    fprintf (stdout, "In V8 mode\n");
else
    fprintf (stdout, "In V7 mode\n");
```

OCI_ATTR_SERVER

モード

読み込み / 書き込み

説明

読み込みモードでは、サービス・コンテキストのサーバー・コンテキスト属性へのポインタを返します。

書き込みモードでは、サービス・コンテキストのサーバー・コンテキスト属性を設定します。

属性データ型

OCI`Server` ** / OCI`Server` *

OCI_ATTR_SESSION

モード

読み込み / 書き込み

説明

読み込みモードでは、サービス・コンテキストの認証コンテキスト属性へのポインタを返します。

書き込みモードでは、サービス・コンテキストの認証コンテキスト属性を設定します。

属性データ型

OCI`Session` ** / OCI`Session` *

OCI_ATTR_STMTCACHE_SIZE

モード

読み込み / 書き込み

説明

文キャッシュに対応したセッションの場合、デフォルトの文キャッシュのサイズ値は 20 です。このデフォルト値は、この属性をサービス・コンテキスト・ハンドルで設定して増減できます。

属性データ型

ub4 * / ub4

OCI_ATTR_TRANS

モード

読み込み / 書き込み

説明

読み込みモードでは、サービス・コンテキストのトランザクション・コンテキスト属性へのポインタを戻します。

書き込みモードでは、サービス・コンテキストのトランザクション・コンテキスト属性を設定します。

属性データ型

OCITrans ** / OCITrans *

サーバー・ハンドル属性

OCI_ATTR_ENV

モード

読み込み

説明

サーバー・コンテキストに対応付けられた環境コンテキストを戻します。

属性データ型

OCIEnv **

OCI_ATTR_EXTERNAL_NAME

モード

読み込み / 書き込み

説明

外部名は使用しやすいグローバルな名前です。これは `sys.props$.value$` に格納されています (`name = 'GLOBAL_DB_NAME'`)。外部名は、すべてのデータベースでネットワーク・ディレクトリ・サービスを使用して登録しないかぎり、一意である保証はありません。

分散トランザクションを調整する場合に、データベース名をサーバーと交換できます。サーバー・データベース名には、`OCISessionBegin()` コールの発行時、データベースがオープンされていた場合のみ、アクセスすることができます。

属性データ型

text **（読み込み）/**text ***（書き込み）

OCI_ATTR_FOCBK

モード

読み込み / 書き込み

説明

関連項目： 9-41 ページ「アプリケーション・フェイルオーバー・コールバック」

属性データ型

OCIFocbkStruct *

OCI_ATTR_INTERNAL_NAME

モード

読み込み / 書き込み

説明

グローバル・トランザクションの実行時に記録されるクライアント・データベース名を設定します。この名前は、障害のために準備状態で保留になっている可能性があるトランザクションを追跡するために、DBA で使用できます。

属性データ型

text **（読み込み）/**text ***（書き込み）

OCI_ATTR_IN_V8_MODE

モード

読み込み

説明

アプリケーションが（`OCISvcCtxToLda()` コールなどによって）Oracle バージョン 7 モードに切り替えられているかどうかを判断できます。戻り値が 0（ゼロ）以外（TRUE）の場合、アプリケーションは現在 Oracle バージョン 8 モードで実行されており、戻り値が 0（ゼロ）（FALSE）の場合、アプリケーションは現在 Oracle バージョン 7 モードで実行されています。

属性データ型

ub1 *

OCI_ATTR_NONBLOCKING_MODE

モード

読み込み / 書き込み

説明

この属性は、ブロック化モードを判断します。

サーバー・コンテキストが非ブロック化モードの場合、読み込み時にこの属性値は TRUE を返します。設定すると、非ブロック化モード属性が切り替えられます。

関連項目： 2-41 ページ「[非ブロック化モード](#)」

属性データ型

ub1

OCI_ATTR_SERVER_GROUP

モード

読み込み / 書き込み

説明

サーバー・グループを指定する、30 文字以内の英数字文字列です。

関連項目： 8-11 ページ「[パスワードおよびセッションの管理](#)」

属性データ型

ub4

OCI_ATTR_SERVER_STATUS

モード

読み込み

説明

サーバー・ハンドルの現行の状態を返します。値は次のとおりです。

- OCI_SERVER_NORMAL — サーバーへのアクティブな接続があります。これは、接続の最後のコールが終了したことを意味します。次のコールが終了する保証はありません。
- OCI_SERVER_NOT_CONNECTED — サーバーへの接続はありません。

属性データ型

ub4

例

次のコード例で、このパラメータの使用方法を示します。

```
ub4 serverStatus = 0
OCIAttrGet((dvoid *)srvhp, OCI_HTYPE_SERVER,
           (dvoid *)&serverStatus, (ub4 *)0, OCI_ATTR_SERVER_STATUS, errhp);
if (serverStatus == OCI_SERVER_NORMAL)
    printf("Connection is up.\n");
else if (serverStatus == OCI_SERVER_NOT_CONNECTED)
    printf("Connection is down.\n");
```

認証情報ハンドル

これらの属性は、ユーザー・セッション・ハンドルにも適用されます。

関連項目： A-17 ページ「[ユーザー・セッション・ハンドル属性](#)」

ユーザー・セッション・ハンドル属性

これらの属性は、認証情報ハンドルにも適用されます。

OCI_ATTR_APPCTX_ATTR

モード

書込み

説明

外部で初期化されたコンテキストの属性名を指定します。

属性データ型

text *

OCI_ATTR_APPCTX_LIST

モード

読込み

説明

セッションに対するアプリケーション・コンテキスト・リスト記述子を取得します。

属性データ型

text *

OCI_ATTR_APPCTX_NAME

モード

書込み

説明

外部で初期化されたコンテキストの名前空間を指定します。

属性データ型

text *

OCI_ATTR_APPCTX_SIZE

モード
書込み

説明
外部で初期化されたコンテキストの配列サイズを、属性数を使用して初期化します。

属性データ型
text *

OCI_ATTR_APPCTX_VALUE

モード
書込み

説明
外部で初期化されたコンテキストの値を指定します。

属性データ型
text *

OCI_ATTR_CERTIFICATE

モード
書込み

説明
プロキシ認証で使用するクライアントの証明書を指定します。

属性データ型
ub1 *

OCI_ATTR_CERTIFICATE_TYPE

モード
書込み

説明
プロキシ認証のタイプを指定します。指定されていない場合は、デフォルトのタイプ X.509 が使用されます。

属性データ型
text *

OCI_ATTR_CLIENT_IDENTIFIER

モード

書込み

説明

セッション・ハンドルにユーザー識別子を指定します。軽量ユーザーの監査用に使用します。識別子の最初の文字に ':' は使用できません。この文字を使用すると、予期しない動作が発生する場合があります。

属性データ型

text *

例

```
OCIAttrSet(session, OCI_HTYPE_SESSION, (dvoid *)"appuser1",  
            (ub4)strlen("appuser1"), OCI_ATTR_APPCTX_IDENTIFIER, error_handle) ;
```

OCI_ATTR_DISTINGUISHED_NAME

モード

書込み

説明

プロキシ認証で使用するクライアントの識別名を指定します。

属性データ型

text *

OCI_ATTR_INITIAL_CLIENT_ROLES

モード

書込み

説明

アプリケーション・サーバーがクライアントにかわって Oracle に接続する場合にクライアントが初めに所有する 1 つまたは複数のロールを指定します。

属性データ型

ub4

OCI_ATTR_MIGSESSION

モード

読み込み / 書き込み

説明

セッション・ハンドル用に識別されたセッションを指定します。同じプロセス内または複数プロセス間で、セッションをある環境から別の環境へ複製することができます。これは、同じマシン上のプロセスでも異なるマシン上のプロセスでも可能です。複製するセッションは、移行可能と認証されている必要があります。

関連項目： 8-11 ページ [「パスワードおよびセッションの管理」](#)

属性データ型

ub1 *

例

次のコード例で、この属性の使用方法を示します。

```
OCIAttrSet ((dvoid *) authp, (ub4) OCI_HTYPE_SESSION, (dvoid *) mig_session,  
            (ub4) sz, (ub4) OCI_ATTR_MIGSESSION, errhp);
```

OCI_ATTR_PASSWORD

モード

書き込み

説明

認証に使用するパスワードを指定します。

属性データ型

text *

OCI_ATTR_PROXY_CREDENTIALS

モード

書き込み

説明

アプリケーション・サーバーの資格証明をプロキシ認証で使用するよう指定します。

属性データ型

ub4

OCI_ATTR_USERNAME

モード
書込み

説明
認証に使用するユーザー名を指定します。

属性データ型
text *

接続プール・ハンドル属性

OCI_ATTR_CONN_TIMEOUT

モード
読込み / 書込み

説明
この時間の値を超えた接続アイドルを終了し、オープンする接続を最適な数に維持します。
この属性は動的に設定できます。この属性を設定していない場合、接続がタイムアウトになることはありません。

属性データ型
ub4 */ub4

OCI_ATTR_CONN_NOWAIT

モード
読込み / 書込み

説明
この属性は、プール内のすべての接続がビジーで、接続数がすでに最大数に達しているとき、接続の再試行を行う必要があるかどうかを判断します。

この属性が設定されている場合は、すべての接続がビジーで、それ以上接続をオープンできないときにエラーが発生します。設定されていない場合、コールは接続が確立するまで待機します。

この属性が設定されている場合、読込み時には、属性値が TRUE で戻されます。

属性データ型
ub1 */ub1

OCI_ATTR_CONN_BUSY_COUNT

モード

読み込み

説明

ビジーの接続の数を返します。

属性データ型

ub4 *

OCI_ATTR_CONN_OPEN_COUNT

モード

読み込み

説明

オープンしている接続の数を返します。

属性データ型

ub4 *

OCI_ATTR_CONN_MIN

モード

読み込み

説明

最小接続数を返します。

属性データ型

ub4 *

OCI_ATTR_CONN_MAX

モード

読み込み

説明

最大接続数を返します。

属性データ型

ub4 *

OCI_ATTR_CONN_INCR

モード
読み込み

説明
接続の増分パラメータを戻します。

属性データ型
ub4 *

セッション・プール・ハンドル属性

セッション・プーリングで使用する属性は、次のとおりです。

OCI_ATTR_SPOOL_BUSY_COUNT

モード
読み込み

説明
ビジーのセッションの数を戻します。

属性データ型
ub4 *

OCI_ATTR_SPOOL_GETMODE

モード
読み込み / 書き込み

説明
この属性は、プール内のすべてのセッションがビジーで、いくつかのセッションがすでに最大に達していることが判明したときに、セッション・プールの動作を判断します。値は次のとおりです。

- OCI_SPOOL_ATTRVAL_WAIT — スレッドはセッションが解放されるまで待機してブロックします。これがデフォルト値です。
- OCI_SPOOL_ATTRVAL_NOWAIT — エラーが戻ります。

- **OCI_SPOOL_ATTRVAL_FORCEGET** — すべてのセッションがビジーで、セッション数が最大に達している場合でも、新しいセッションを作成します。OCI_SessionGet() によって警告が戻されます。この場合、OCI_SessionGet() が警告を戻すのは、新規作成のセッションがセッションの最大数を越えた場合です。

この値を設定した場合は、Oracle データベース・サーバーのインスタンスでサポートできるセッション数よりも多いセッションが作成される可能性があります。この場合は、次のエラーがサーバーから戻されます。

ORA 00018 — 最大セッション数を超過しました

このエラーは、セッション・プールのユーザーに伝播されます。

読み込みの場合は、適切な属性値が戻されます。

属性データ型

ub1 */ub1

OCI_ATTR_SPOOL_INCR

モード

読み込み

説明

セッションの増分パラメータを戻します。

属性データ型

ub4 *

OCI_ATTR_SPOOL_MAX

モード

読み込み

説明

最大セッション数を戻します。

属性データ型

ub4 *

OCI_ATTR_SPOOL_MIN

モード

読込み

説明

最小セッション数を戻します。

属性データ型

ub4 *

OCI_ATTR_SPOOL_OPEN_COUNT

モード

読込み

説明

オープンしているセッションの数を戻します。

属性データ型

ub4 *

OCI_ATTR_SPOOL_TIMEOUT

モード

読込み / 書込み

説明

この時間（秒）を超えたセッション・アイドルは定期的に終了し、オープンしているセッションの数を最適に維持します。この属性は動的に設定できます。この属性が設定されておらず、プール内に領域が必要な場合は、最低使用頻度のセッションがタイムアウトとなります。

属性データ型

ub4 */ub4

トランザクション・ハンドル属性

OCI_ATTR_TRANS_NAME

モード

読込み / 書込み

説明

この属性を使用して、トランザクションを識別するテキスト文字列の構築または読込みができます。これは、XID を使用してトランザクションを識別するかわりに使用される方法です。テキスト文字列は、64 バイトまで指定できます。

属性データ型

`text **`（読込み） / `text *`（書込み）

OCI_ATTR_TRANS_TIMEOUT

モード

読込み / 書込み

説明

準備時間に使用するタイムアウト値を設定または読込みができます。

属性データ型

`ub4 *`（読込み） / `ub4`（書込み）

OCI_ATTR_XID

モード

読込み / 書込み

説明

トランザクションを識別する XID の設定または読込みができます。

属性データ型

`XID **`（読込み） / `XID *`（書込み）

文ハンドル属性

OCI_ATTR_CURRENT_POSITION

モード

読み込み

説明

結果セット内の現在の位置を示します。この属性は読み込みのみ可能です。設定はできません。

属性データ型

ub4 *

OCI_ATTR_ENV

モード

読み込み

説明

文に対応付けられた環境コンテキストを返します。

属性データ型

OCIEnv **

OCI_ATTR_NUM_DML_ERRORS

モード

読み込み

説明

DML 操作でのエラーの数を返します。

属性データ型

ub4 *

OCI_ATTR_PARAM_COUNT

モード

読み込み

説明

この属性を使用して、文ハンドルに対応付けられた文の選択リストにある列数を取得できます。

属性データ型

ub4 *

例

次のコード例で、この属性の使用方法を示します。

```
/* Describe of a select-list */
text *selstmt = "SELECT * FROM EMP";
ub4 parmcnt;
OCIParam *parmdp;

err = OCISStmtPrepare (stmhp, errhp, selstmt,
                      (ub4)strlen((char *)selstmt),
                      (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT);
err = OCISStmtExecute (svchp, stmhp, errhp, (ub4)1, (ub4)0,
                      (const OCISnapshot*) 0, (OCISnapshot*)0, OCI_DESCRIBE_ONLY);

/* get the number of columns in the select list */
err = OCIAttrGet ((dvoid *)stmhp, (ub4)OCI_HTYPE_STMT, (dvoid *)
                 &parmcnt, (ub4 *) 0, (ub4)OCI_ATTR_PARAM_COUNT, errhp);

/* get describe information for each column */
for (i = 1; i < parmcnt; i++) {
    OCIParamGet (dvoid *)stmhp, OCI_HTYPE_STMT, errhp, &parmdp, i);
/* get the attributes for each column */
}
```

OCI_ATTR_PARSE_ERROR_OFFSET

モード

読み込み

説明

文の解析エラー・オフセットを戻します。

属性データ型

ub2 *

OCI_ATTR_PREFETCH_MEMORY

モード

書込み

説明

プリフェッチされるトップレベル行のメモリー・レベルを設定します。このメモリー・レベルが、指定したメモリー使用量の制限以下である場合は、指定したトップレベル行カウントまでの行がフェッチされます。デフォルト値は 0（ゼロ）です。これは、プリフェッチされた行数の計算にメモリー・サイズが含まれていないことを意味します。

属性データ型

ub4 *

OCI_ATTR_PREFETCH_ROWS

モード

書込み

説明

プリフェッチされるトップレベル行の数を設定します。デフォルト値は 1 行です。

属性データ型

ub4 *

OCI_ATTR_ROW_COUNT

モード

読込み

説明

これまでに処理した行の数を戻します。デフォルト値は 1 です。

スクロール不可カーソルの場合、OCI_ATTR_ROW_COUNT は、この文ハンドルの実行後に発行された OCISstmtFetch2 () コールでユーザー・バッファにフェッチされた行の合計数です。スクロール不可カーソルは前方への順次アクセスのみであるため、この属性は、アプリケーションで参照する最後の行番号を表します。

スクロール・カーソルの場合、OCI_ATTR_ROW_COUNT は、ユーザー・バッファにフェッチされた行の最大（絶対）数を表します。アプリケーションでは、任意の位置でフェッチを行うことができるため、この属性は、（スクロール可能な）文の実行後にユーザーのバッファにフェッチされた行の合計数である必要はありません。

属性データ型

ub4 *

OCI_ATTR_ROWID

モード
読み込み

説明

OCIDescriptorAlloc() によって割り当てられる ROWID 記述子を戻します。

関連項目： 2-39 ページ [「位置指定の更新および削除」](#) および 3-14 ページ [「DATE」](#)

属性データ型
OCIRowid *

OCI_ATTR_ROWS_FETCHED

モード
読み込み

説明

前回のフェッチまたは 1 回以上の反復処理でユーザーのバッファに正常にフェッチされた行の数を示します。スクロール可能およびスクロール不可の文ハンドルの両方で使用できます。

属性データ型
ub4 *

例

```
ub4 rows;
ub4 sizep = sizeof(ub4);
OCIAttrGet((dvoid *) stmhp, (ub4) OCI_HTYPE_STMT,
            (dvoid *)& rows, (ub4 *) &sizep, (ub4)OCI_ATTR_ROWS_FETCHED, errhp));
```

OCI_ATTR_SQLFNCODE

モード
読み込み

説明

文に対応付けられた SQL コマンドの関数コードを戻します。

属性データ型
ub2 *

注意

関連項目： SQL コマンド・コードのリストを A-31 ページの表 A-1 「SQL コマンド・コード」に示します。

表 A-1 SQL コマンド・コード

コード	SQL 関数	コード	SQL 関数	コード	SQL 関数
01	CREATE TABLE	43	DROP EXTERNAL DATABASE	85	TRUNCATE TABLE
02	SET ROLE	44	CREATE DATABASE	86	TRUNCATE CLUSTER
03	INSERT	45	ALTER DATABASE	87	CREATE BITMAPFILE
04	SELECT	46	CREATE ROLLBACK SEGMENT	88	ALTER VIEW
05	UPDATE	47	ALTER ROLLBACK SEGMENT	89	DROP BITMAPFILE
06	DROP ROLE	48	DROP ROLLBACK SEGMENT	90	SET CONSTRAINTS
07	DROP VIEW	49	CREATE TABLESPACE	91	CREATE FUNCTION
08	DROP TABLE	50	ALTER TABLESPACE	92	ALTER FUNCTION
09	DELETE	51	DROP TABLESPACE	93	DROP FUNCTION
10	CREATE VIEW	52	ALTER SESSION	94	CREATE PACKAGE
11	DROP USER	53	ALTER USER	95	ALTER PACKAGE
12	CREATE ROLE	54	COMMIT (WORK)	96	DROP PACKAGE
13	CREATE SEQUENCE	55	ROLLBACK	97	CREATE PACKAGE BODY
14	ALTER SEQUENCE	56	SAVEPOINT	98	ALTER PACKAGE BODY
15	(使用されていません)	57	CREATE CONTROL FILE	99	DROP PACKAGE BODY
16	DROP SEQUENCE	58	ALTER TRACING	157	CREATE DIRECTORY
17	CREATE SCHEMA	59	CREATE TRIGGER	158	DROP DIRECTORY
18	CREATE CLUSTER	60	ALTER TRIGGER	159	CREATE LIBRARY
19	CREATE USER	61	DROP TRIGGER	160	CREATE JAVA
20	CREATE INDEX	62	ANALYZE TABLE	161	ALTER JAVA
21	DROP INDEX	63	ANALYZE INDEX	162	DROP JAVA
22	DROP CLUSTER	64	ANALYZE CLUSTER	163	CREATE OPERATOR
23	VALIDATE INDEX	65	CREATE PROFILE	164	CREATE INDEXTYPE

表 A-1 SQL コマンド・コード (続き)

コード	SQL 関数	コード	SQL 関数	コード	SQL 関数
24	CREATE PROCEDURE	66	DROP PROFILE	165	DROP INDEXTYPE
25	ALTER PROCEDURE	67	ALTER PROFILE	166	ALTER INDEXTYPE
26	ALTER TABLE	68	DROP PROCEDURE	167	DROP OPERATOR
27	EXPLAIN	69	(使用されていません)	168	ASSOCIATE STATISTICS
28	GRANT	70	ALTER RESOURCE COST	169	DISASSOCIATE STATISTICS
29	REVOKE	71	CREATE SNAPSHOT LOG	170	CALL METHOD
30	CREATE SYNONYM	72	ALTER SNAPSHOT LOG	171	CREATE SUMMARY
31	DROP SYNONYM	73	DROP SNAPSHOT LOG	172	ALTER SUMMARY
32	ALTER SYSTEM SWITCH LOG	74	CREATE SNAPSHOT	73	DROP SUMMARY
33	SET TRANSACTION	75	ALTER SNAPSHOT	174	CREATE DIMENSION
34	PL/SQL EXECUTE	76	DROP SNAPSHOT	175	ALTER DIMENSION
35	LOCK	77	CREATE TYPE	176	DROP DIMENSION
36	NOOP	78	DROP TYPE	177	CREATE CONTEXT
37	RENAME	79	ALTER ROLE	178	DROP CONTEXT
38	COMMENT	80	ALTER TYPE	179	ALTER OUTLINE
39	AUDIT	81	CREATE TYPE BODY	180	CREATE OUTLINE
40	NO AUDIT	82	ALTER TYPE BODY	181	DROP OUTLINE
41	ALTER INDEX	83	DROP TYPE BODY	182	UPDATE INDEXES
42	CREATE EXTERNAL DATABASE	84	DROP LIBRARY	183	ALTER OPERATOR

OCI_ATTR_STATEMENT

モード

読み込み

説明

プリコンパイルされた SQL 文のテキストを文ハンドルに戻します。UTF-16 モードでは、戻された文は UTF-16 エンコーディングになります。長さは、常にバイト単位です。

属性データ型

text *

OCI_ATTR_STMT_STATE

モード

読み込み

説明

文のフェッチ状態に戻します。コール元はこの属性を使用して、このセッションが別のサービス・コンテキストで利用できるかどうか、現行のデータ・アクセス・コール・セットでまだ必要かどうかを判断できます。基本的に、フェッチ実行サイクルの途中では、別の文を実行するためにセッション・ハンドルを解放することはありません。次の値が有効です。

- OCI_STMT_STATE_INITIALIZED
- OCI_STMT_STATE_EXECUTED
- OCI_STMT_STATE_END_OF_FETCH

属性データ型

ub4 *

OCI_ATTR_STMT_TYPE

モード

読み込み

説明

ハンドルに対応付けられた文のタイプです。次の値が有効です。

- OCI_STMT_SELECT
- OCI_STMT_UPDATE
- OCI_STMT_DELETE
- OCI_STMT_INSERT

- OCI_STMT_CREATE
- OCI_STMT_DROP
- OCI_STMT_ALTER
- OCI_STMT_BEGIN (PL/SQL 文)
- OCI_STMT_DECLARE (PL/SQL 文)

属性データ型

ub2 *

バインド・ハンドル属性

OCI_ATTR_CHAR_COUNT

モード

書込み

説明

関連項目： 5-37 ページ [「バインド時のバッファの拡張」](#)

属性データ型

ub4 *

OCI_ATTR_CHARSET_FORM

モード

読込み / 書込み

説明

バインド・ハンドルのキャラクタ・セット・フォームです。デフォルトのフォームは SQLCS_IMPLICIT です。この属性を設定すると、バインド・ハンドルでクライアント側のデータベースまたは各国語キャラクタ・セットを使用できます。この属性は、各国語キャラクタ・セットの場合は SQLCS_NCHAR に設定し、データベース・キャラクタ・セットの場合は SQLCS_IMPLICIT に設定します。

属性データ型

ub1 *

OCI_ATTR_CHARSET_ID

モード

読み込み / 書き込み

説明

バインド・ハンドルのキャラクタ・セット ID です。入力データのキャラクタ・セットが UTF-16（下位互換性のために保持されている、使用できない OCI_UCS2ID の置換え）の場合、ユーザーはキャラクタ・セット ID を OCI_UTF16ID に設定する必要があります。バインド値バッファは **utext** バッファであるとみなされ、入力長ポインタおよび戻り値の長さセマンティクスは、キャラクタ・セマンティクス（**utext** の数）に変更されます。ただし、先行する OCIBind コールでは、バインド値バッファのサイズをバイト単位で指定する必要があります。

OCI_ATTR_CHARSET_ID は、OCI_ATTR_CHARSET_FORM（設定する場合）の後に設定する必要があります。OCI_ATTR_CHARSET_FORM を設定する前に OCI_ATTR_CHARSET_ID を設定すると、予想しない結果になります。

関連項目： 5-34 ページ「[バインドおよび定義における文字変換の問題](#)」

属性データ型

ub2 *

OCI_ATTR_MAXCHAR_SIZE

モード

書き込み

説明

バインドするデータを格納するためにアプリケーションがサーバー上に予約する文字数を設定します。

関連項目： 5-37 ページ「[OCI_ATTR_MAXCHAR_SIZE 属性の使用](#)」

属性データ型

sb4 *

OCI_ATTR_MAXDATA_SIZE

モード

読み込み / 書き込み

説明

関連項目： 5-36 ページ [「OCI_ATTR_MAXDATA_SIZE 属性の使用」](#)

属性データ型

sb4 *

OCI_ATTR_PDPRC

モード

書き込み

説明

パック 10 進数の精度を指定します。SQLT_PDN 値の場合、精度は **2*(value_sz-1)** に等しくしてください。SQLT_SLS 値の場合、精度は **(value_sz-1)** に等しくしてください。

バインドまたは定義後に、この値は 0（ゼロ）に初期化されます。最初に

OCI_ATTR_PDPRC 属性を設定して、次に OCI_ATTR_PDSCL を設定します。これらの値のどちらかを変更する必要がある場合は、最初に再バインドまたは再定義を行って、次にこれらの属性を適切に再設定する必要があります。

属性データ型

ub2 *

OCI_ATTR_PDSCL

モード

書き込み

説明

パック 10 進値のスケールを指定します。

バインドまたは定義後に、この値は 0（ゼロ）に初期化されます。最初に

OCI_ATTR_PDPRC 属性を設定して、次に OCI_ATTR_PDSCL を設定します。これらの値のどちらかを変更する必要がある場合は、最初に再バインドまたは再定義を行って、次にこれらの属性を適切に再設定する必要があります。

属性データ型

sb2 *

OCI_ATTR_ROWS_RETURNED

モード

読み込み

説明

この属性は、RETURNING 句で DML 文をバインドする OUT のコールバック関数を実行しているとき、現行の反復で戻される行数を戻します。

属性データ型

ub4 *

定義ハンドル属性

OCI_ATTR_CHAR_COUNT

モード

書き込み

説明

この属性は使用できません。

キャラクタ・タイプ・データの文字数を設定します。この属性は、定義バッファに必要な文字数を指定します。定義コールで指定した定義バッファ長は、文字数より大きくする必要があります。

属性データ型

ub4 *

OCI_ATTR_CHARSET_FORM

モード

読み込み / 書き込み

説明

定義ハンドルのキャラクタ・セット・フォームです。デフォルトのフォームは SQLCS_IMPLICIT です。この属性を設定すると、定義ハンドルでクライアント側のデータベースまたは各国語キャラクタ・セットを使用できます。この属性は、各国語キャラクタ・セットの場合は SQLCS_NCHAR に設定し、データベース・キャラクタ・セットの場合は SQLCS_IMPLICIT に設定します。

属性データ型

ub1 *

OCI_ATTR_CHARSET_ID

モード

読み込み / 書き込み

説明

定義ハンドルのキャラクタ・セット ID です。出力データのキャラクタ・セットが UTF-16 である場合、ユーザーはキャラクタ・セット IDOTT を OCI_UTF16ID に設定する必要があります。定義値バッファは `utext` バッファであるとみなされ、インジケータおよび戻り値の長さセマンティクスは、キャラクタ・セマンティクス (`utext` の数) に変更されます。ただし、先行する OCIDefine コールでは、定義値バッファのサイズをバイト単位で指定する必要があります。

OCI_ATTR_CHARSET_ID は、OCI_ATTR_CHARSET_FORM (設定する場合) の後に設定する必要があります。OCI_ATTR_CHARSET_FORM を設定する前に OCI_ATTR_CHARSET_ID を設定すると、予想しない結果になります。

関連項目： 5-34 ページ [「バインドおよび定義における文字変換の問題」](#)

属性データ型

ub2 *

OCI_ATTR_MAXCHAR_SIZE

モード

書き込み

説明

クライアント・アプリケーションが定義バッファに格納できる最大文字数を指定します。

関連項目： 5-37 ページ [「OCI_ATTR_MAXCHAR_SIZE 属性の使用」](#)

属性データ型

sb4 *

OCI_ATTR_PDPRC

モード
書込み

説明

パック 10 進数の精度を指定します。SQLT_PDN 値の場合、精度は $2 * (\text{value_sz} - 1)$ に等しくしてください。SQLT_SLS 値の場合、精度は $(\text{value_sz} - 1)$ に等しくしてください。

バインドまたは定義後に、この値は 0（ゼロ）に初期化されます。最初に OCI_ATTR_PDPRC 属性を設定して、次に OCI_ATTR_PDSCL を設定します。これらの値のどちらかを変更する必要がある場合は、最初に再バインドまたは再定義を行って、次にこれらの属性を適切に再設定する必要があります。

属性データ型
ub2 *

OCI_ATTR_PDSCL

モード
書込み

説明

パック 10 進値のスケールを指定します。

バインドまたは定義後に、この値は 0（ゼロ）に初期化されます。最初に OCI_ATTR_PDPRC 属性を設定して、次に OCI_ATTR_PDSCL を設定します。これらの値のどちらかを変更する必要がある場合は、最初に再バインドまたは再定義を行って、次にこれらの属性を適切に再設定する必要があります。

属性データ型
sb2 *

記述ハンドル属性

OCI_ATTR_PARAM

モード

読み込み

説明

記述のルートを指し示します。後続の OCIAttrGet () および OCIParamGet () コールのために使用します。

属性データ型

ub4 *

OCI_ATTR_PARAM_COUNT

モード

読み込み

説明

記述ハンドル内のパラメータ数を戻します。記述ハンドルが選択リストの記述である場合は、選択リスト内の列数を示します。

属性データ型

ub4 *

パラメータ記述子属性

関連項目： パラメータ記述子の属性の詳細なリストは、[第6章「スキーマ・メタデータの記述」](#)を参照してください。

LOB ロケータ属性

OCI_ATTR_LOBEMPTY

モード

書込み

説明

内部 LOB ロケータを Empty 値に設定します。これによって、そのロケータは INSERT 文または UPDATE 文のバインド変数として使用でき、LOB を Empty 値に初期化できます。LOB が Empty 値になると、OCIlobWrite() をコールしてデータを LOB に移入できます。この属性は内部 LOB (BLOB、CLOB、NCLOB) に対してのみ有効です。

アプリケーションから、宣言などの 0 (ゼロ) の値を持つ **ub4** のアドレスを渡します。

```
ub4 lobEmpty = 0
```

と宣言した後にはアドレス &lobEmpty を渡します。

属性データ型**ub4 ***

複合オブジェクト属性

関連項目： 10-20 ページ [「複合オブジェクト検索」](#)

複合オブジェクト検索ハンドル属性

OCI_ATTR_COMPLEXOBJECT_LEVEL

モード

書込み

説明

複合オブジェクト検索のネスト・レベルです。

属性データ型**ub4 ***

OCI_ATTR_COMPLEXOBJECT_COLL_OUTOFLINE

モード

書込み

説明

オブジェクト型アウト・ラインにあるコレクション属性をフェッチするかどうかを示します。

属性データ型

ub1 *

複合オブジェクト検索記述子の属性

OCI_ATTR_COMPLEXOBJECTCOMP_TYPE

モード

書込み

説明

複合オブジェクト検索に続く REF のタイプです。

属性データ型

dvoid *

OCI_ATTR_COMPLEXOBJECTCOMP_TYPE_LEVEL

モード

書込み

説明

後続く OCI_ATTR_COMPLEXOBJECTCOMP_TYPE 型の REF のネスト・レベルです。

属性データ型

ub4 *

アドバンスト・キューイング記述子の属性

関連項目： アドバンスト・キューイングのプロジェクトおよびオプションの詳細は、『Oracle9i アプリケーション開発者ガイド - アドバンスト・キューイング』を参照してください。

OCIAQEnqOptions 記述子の属性

次に示す属性は、OCIAQEnqOptions 記述子のプロパティです。

OCI_ATTR_RELATIVE_MSGID

モード

読み込み / 書き込み

説明

順序逸脱操作で参照されるメッセージのメッセージ識別子を指定します。この値は、OCI_ATTR_SEQUENCE_DIVISION で OCI_ENQ_BEFORE が指定されている場合にのみ有効です。順序逸脱が未指定の場合には、この値は無視されます。

属性データ型

OCIRaw *

OCI_ATTR_SEQUENCE_DEVIATION

モード

読み込み / 書き込み

説明

現在エンキューされているメッセージを、すでにキューにあるその他のメッセージより先にデキューするかどうかを指定します。

属性データ型

ub4

有効な値

次に示す値のみ有効です。

- OCI_ENQ_BEFORE — メッセージは、OCI_ATTR_RELATIVE_MSGID で指定したメッセージより先にエンキューされます。
- OCI_ENQ_TOP — メッセージは、その他のあらゆるメッセージより先にエンキューされます。

OCI_ATTR_VISIBILITY

モード

読み込み / 書き込み

説明

エンキュー要求のトランザクション動作を指定します。

属性データ型

ub4

有効な値

次に示す値のみ有効です。

- OCI_ENQ_ON_COMMIT — エンキューはカレント・トランザクションの一部です。操作は、トランザクションがコミットするときに完了します。これは、デフォルトです。
- OCI_ENQ_IMMEDIATE — エンキューはカレント・トランザクションの一部ではありません。固有のトランザクションで操作が行われます。

OCIAQDeqOptions 記述子の属性

次に示す属性は、OCIAQDeqOptions 記述子のプロパティです。

OCI_ATTR_CONSUMER_NAME

モード

読み込み / 書き込み

説明

コンシューマ名です。コンシューマ名に一致するメッセージのみがアクセスされます。キューが複数のコンシューマに対して設定されていない場合、このフィールドは NULL に設定します。

属性データ型

text *

OCI_ATTR_CORRELATION

モード

読み込み / 書き込み

説明

デキューするメッセージの相関識別子を指定します。パーセント符号 (%) やアンダースコア (_) などの特殊なパターン・マッチング文字を使用できます。2 つ以上のメッセージがパターンに一致する場合、デキューの順序は確定されません。

属性データ型

text *

OCI_ATTR_DEQ_MODE

モード

読み込み / 書き込み

説明

デキューに対応付けられたロック動作を指定します。

属性データ型

ub4

有効な値

次に示す値のみ有効です。

- OCI_DEQ_BROWSE — ロックを取得することなくメッセージを読み込みます。これは、SELECT 文と同じです。
- OCI_DEQ_LOCKED — メッセージを読み込み、書き込みロックを取得します。ロックは、トランザクションの継続時間中は有効です。これは、SELECT FOR UPDATE 文と同じです。
- OCI_DEQ_REMOVE — メッセージを読み込み、それを更新または削除します。これは、デフォルトです。メッセージは、保存プロパティに基づいてキュー・テーブルに保存されます。
- OCI_DEQ_NO_DATA — メッセージの受取りを確認しますが、実際のメッセージ内容は送付しません。

OCI_ATTR_DEQ_MSGID

モード

読み込み / 書き込み

説明

デキューするメッセージのメッセージ識別子を指定します。

属性データ型

OCIRaw *

OCI_ATTR_NAVIGATION

モード

読み込み / 書き込み

説明

取り出すメッセージの位置を指定します。最初に位置が判断されます。次に検索基準が適用されます。最後にメッセージが取り出されます。

属性データ型

ub4

有効な値

次に示す値のみ有効です。

- OCI_DEQ_FIRST_MSG — 使用可能なメッセージの中から、検索基準に一致する最初のメッセージを取り出します。これにより、位置はキューの先頭にリセットされます。
- OCI_DEQ_NEXT_MSG — 使用可能なメッセージの中から、検索基準に一致する次のメッセージを取り出します。前のメッセージがメッセージ・グループに属していた場合は、検索基準に一致し、なおかつメッセージ・グループに属するメッセージが AQ により取り出されます。これは、デフォルトです。
- OCI_DEQ_NEXT_TRANSACTION — 残りのカレント・トランザクション・グループをスキップし、次のトランザクション・グループから最初のメッセージを取り出します。このオプションは、現行のキューに対してメッセージをグループ化できる場合のみ使用できます。

OCI_ATTR_VISIBILITY

モード

読み込み / 書き込み

説明

新しいメッセージをカレント・トランザクションの一部としてデキューするかどうかを指定します。BROWSE モードを使用しているときは、この可視性（visibility）パラメータは無視されます。

属性データ型

ub4

有効な値

次に示す値のみ有効です。

- OCI_DEQ_ON_COMMIT — デキューはカレント・トランザクションの一部になります。これは、デフォルトです。
- OCI_DEQ_IMMEDIATE — デキューされたメッセージはカレント・トランザクションの一部ではありません。固有のトランザクションが構成されます。

OCI_ATTR_WAIT

モード

読み込み / 書き込み

説明

検索基準に一致するメッセージがないときの待機時間を指定します。同じグループのメッセージがデキューされている場合、このパラメータは無視されます。

属性データ型

ub4

有効な値

ub4 値はいずれも有効ですが、さらに次のような定数が事前定義されています。

- OCI_DEQ_WAIT_FOREVER — いつまでも待機します。これは、デフォルトです。
- OCI_DEQ_NO_WAIT — 待機しません。

OCIAQMsgProperties 記述子の属性

次に示す属性は、OCIAQMsgProperties 記述子のプロパティです。

OCI_ATTR_ATTEMPTS

モード

読込み

説明

あるメッセージに対して今までに何回デキューが試みられたかを指定します。このパラメータを、エンキュー時刻に設定することはできません。

属性データ型

sb4

有効な値

sb4 値はいずれも有効です。

OCI_ATTR_CORRELATION

モード

読込み / 書込み

説明

エンキュー時にプロデューサがメッセージに付けた ID を指定します。

属性データ型

text *

有効な値

文字列は 128 バイトまで有効です。

OCI_ATTR_DELAY

モード

読込み / 書込み

説明

エンキューしたメッセージを遅延させる秒数を指定します。遅延はメッセージがデキュー可能になった後の秒数を示します。msgid によりデキューすると、遅延指定は上書きされます。遅延を設定すると、エンキューされたメッセージは WAITING 状態となり、遅延時間が終了するとともに READY 状態となります。DELAY 処理にはキュー・モニターを起動させる

必要があります。遅延は、メッセージをエンキューするプロデューサが設定することに注意してください。

属性データ型

sb4

有効な値

すべての sb4 値が有効ですが、さらに次のような定数が事前定義されています。

- OCI_MSG_NO_DELAY – メッセージが即時にデキューできることを示します。

OCI_ATTR_ENQ_TIME

モード

読み込み

説明

メッセージがエンキューされた時間を指定します。この値はシステムにより判断されるため、ユーザーは設定できません。

属性データ型

OCIDate

OCI_ATTR_EXCEPTION_QUEUE

モード

読み込み / 書き込み

説明

正常に処理できないメッセージの移動先となるキューの名前を指定します。メッセージが移動されるのは次の 2 つの場合です。デキューが *max_retries* 以内の回数で成功しなかった場合とメッセージの有効期限が切れた場合です。例外キューのメッセージはすべて EXPIRED 状態です。

デフォルトは、キュー・テーブルと対応付けられた例外キューです。移動時に指定の例外キューが存在しない場合、メッセージはキュー・テーブルと対応付けられたデフォルトの例外キューに移動し、警告がアラート・ファイルにログ記録されます。デフォルトの例外キューが使用されている場合は、デキュー時に NULL 値が戻されます。

この属性は有効なキュー名を参照する必要があります。

属性データ型

text *

OCI_ATTR_EXPIRATION

モード

読み込み / 書き込み

説明

メッセージの期限切れを指定します。この時間（秒）を経過したメッセージはデキューされません。このパラメータは、遅延からのオフセットです。期限切れ処理にはキュー・モニターを起動させる必要があります。

期限切れ前のメッセージは **READY** 状態です。期限切れ前にデキューされないメッセージは、**EXPIRED** 状態となって例外デキューに移動します。

属性データ型

sb4

有効な値

すべての sb4 値が有効ですが、さらに次のような定数が事前定義されています。

- OCI_MSG_NO_EXPIRATION — メッセージは期限切れになりません。

OCI_ATTR_MSG_STATE

モード

読み込み

説明

デキュー時のメッセージの状態を指定します。このパラメータを、エンキュー時刻に設定することはできません。

属性データ型

ub4

有効な値

戻される値は次の 4 つのみです。

- OCI_MSG_WAITING — メッセージの遅延時間がまだ経過していません。
- OCI_MSG_READY — メッセージは処理できる状態にあります。
- OCI_MSG_PROCESSED — メッセージは処理され、保存されています。
- OCI_MSG_EXPIRED — メッセージは例外キューに移動しました。

OCI_ATTR_PRIORITY

モード

読み込み / 書き込み

説明

メッセージの優先順位を指定します。数値が小さいほど高い優先順位を示します。優先順位は、負数も含めたあらゆる数値で指定できます。

デフォルト値は 0（ゼロ）です。

属性データ型

sb4

OCI_ATTR_RECIPIENT_LIST

モード

書き込み

説明

このパラメータは、複数のコンシューマが可能なキューに対してのみ有効です。デフォルトの受信者はキューのサブスクライバです。このパラメータはデキュー時にコンシューマに戻されません。

属性データ型

OCIAQAgent **

OCI_ATTR_SENDER_ID

モード

読み込み / 書き込み

説明

メッセージの最初の送信者を識別します。

属性データ型

OCIAgent *

OCI_ATTR_ORIGINAL_MSGID

モード

読み込み / 書き込み

説明

そのメッセージを生成した最後のキューの ID です。メッセージがあるキューから別のキューに伝播される際に、この属性はメッセージの最後の伝播元であるキューの ID を識別します。メッセージが複数のキューを経由して伝播されている場合、この属性は、最初のキューではなく最後のキューを識別します。

属性データ型

OCIRaw *

OCIAQAgent 記述子の属性

次に示す属性は、OCIAQAgent 記述子のプロパティです。

OCI_ATTR_AGENT_ADDRESS

モード

読み込み / 書き込み

説明

プロトコル特定の受信者アドレスです。プロトコルが 0（デフォルト）の場合、アドレスのフォームは `[schema.]queue[@dblink]` です。

属性データ型

text *

有効な値

128 バイト以下の任意の文字列を指定できます。

OCI_ATTR_AGENT_NAME

モード

読み込み / 書き込み

説明

メッセージのプロデューサ名またはコンシューマ名です。

属性データ型

text *

有効な値

30 バイト以下の任意の Oracle 識別子を指定できます。

OCI_ATTR_AGENT_PROTOCOL

モード

読み込み / 書き込み

説明

アドレスを解釈しメッセージを伝播させるプロトコルです。デフォルト値（現在はこの値のみサポートされています）は 0（ゼロ）です。

属性データ型

ub1

有効な値

有効な値は 0（ゼロ）のみです。これはデフォルトでもあります。

OCIServerDNs 記述子の属性

次に示す属性は、OCIServerDNs 記述子のプロパティです。

OCI_ATTR_DN_COUNT

モード

読み込み

説明

記述子に挿入されたデータベース・サーバーの数です。

属性データ型

ub2

OCI_ATTR_SERVER_DN

モード

読み込み / 書き込み

説明

読み取りモードの場合、この属性は、記述子にすでに挿入されているデータベース・サーバーの識別名のリストを戻します。

書き込みモードの場合、この属性は、データベース・サーバーの識別名を取得します。

属性データ型

text **/text *

サブスクリプション・ハンドル属性

関連項目： 9-52 ページ [「パブリッシュ・サブスクライブの通知」](#)

OCI_ATTR_SERVER_DNS

モード

読み込み / 書き込み

説明

クライアントが登録用に使用するデータベース・サーバーの識別名です。

属性データ型

OCIServerDNs *

OCI_ATTR_SUBSCR_CALLBACK

モード

読み込み / 書き込み

説明

サブスクリプション・コールバックです。属性 OCI_ATTR_SUBSCR_RECPTPROTO が OCI_SUBSCR_PROTO_OCI に設定されているか、または値が設定されていない場合、この属性は、サブスクリプション・ハンドルが登録コール OCISubscriptionRegister() に渡される前に設定する必要があります。

属性データ型

OCISubscriptionNotify *

OCI_ATTR_SUBSCR_CTX

モード

読み込み / 書き込み

説明

システムによって呼び出された際に、クライアントから OCI_ATTR_SUBSCR_CALLBACK が示すユーザー・コールバックに渡されるコンテキストです。属性 OCI_ATTR_SUBSCR_RECPTPROTO が OCI_SUBSCR_PROTO_OCI に設定されているか、または値が設定されていない場合、この属性は、サブスクリプション・ハンドルが登録コール OCISubscriptionRegister() に渡される前に設定する必要があります。

属性データ型

void *

OCI_ATTR_SUBSCR_NAME

モード

読み込み / 書き込み

説明

サブスクリプション名です。すべてのサブスクリプションは、サブスクリプション名によって識別されます。サブスクリプション名は、指定された長さの一連のバイト数から構成されます。サブスクリプション名のバイト長は、この名前がヌル文字で終了しないことを前提として指定する必要があります。名前にマルチバイト・キャラクタが含まれる可能性があるため、この前提は重要です。

クライアントでは、OCIAttrSet() コールを使用して、OCI_HTYPE_SUBSCR のハンドル・タイプと OCI_ATTR_SUBSCR_NAME の属性タイプを指定することによりサブスクリプション・ハンドルのサブスクリプション名属性を設定できます。

すべてのサブスクリプション・コールバックには、OCI_ATTR_SUBSCR_NAME 属性および OCI_ATTR_SUBSCR_NAMESPACE 属性が設定されたサブスクリプション・ハンドルが必要です。これらの属性が設定されていない場合は、エラーが戻されます。サブスクリプション・ハンドルに設定されるサブスクリプション名は、その名前空間と一貫性を持たせる必要があります。

属性データ型

text *

OCI_ATTR_SUBSCR_NAMESPACE

モード

読み込み / 書き込み

説明

サブスクリプション・ハンドルが使用される名前空間です。この属性の有効な値は、OCI_SUBSCR_NAMESPACE_AQ および OCI_SUBSCR_NAMESPACE_ANONYMOUS です。サブスクリプション・ハンドルに設定されるサブスクリプション名は、その名前空間と一貫性を持たせる必要があります。

属性データ型

ub4 *

OCI_ATTR_SUBSCR_PAYLOAD

モード

読み込み / 書き込み

説明

通知とともに送信する必要があるペイロードに対応するバッファです。バッファ長も同じ属性設定コールに指定できます。サブスクリプションで転送を実行するには、この属性を設定する必要があります。このリリースでは、タイプ化されていない (ub1 *) ペイロードのみをサポートします。

属性データ型

ub1 *

OCI_ATTR_SUBSCR_RECPT

モード

読み込み / 書き込み

説明

属性 OCI_ATTR_SUBSCR_RECPTPROTO が OCI_SUBSCR_PROTO_MAIL、OCI_SUBSCR_PROTO_HTTP または OCI_SUBSCR_PROTO_SERVER に設定されているときの通知の受信者名です。

OCI_SUBSCR_PROTO_HTTP の場合は、OCI_ATTR_SUBSCR_RECPT によって、通知の送信先 HTTP URL (http://www.oracle.com:80 など) が示されます。データベースで HTTP URL の妥当性がチェックされることはありません。

OCI_SUBSCR_PROTO_MAIL の場合は、OCI_ATTR_SUBSCR_RECPT によって、通知の送信先の電子メール・アドレス (xyz@oracle.com など) が示されます。データベース・システムで電子メール・アドレスの妥当性がチェックされることはありません。

OCI_SUBSCR_PROTO_SERVER の場合、OCI_ATTR_SUBSCR_RECPT は、通知イベントで呼び出されるデータベース・プロシージャ（たとえば、`schema.procedure`）を示します。サブスクライバには、実行するプロシージャに対する適切な権限が必要です。

関連項目： プロシージャ定義については、9-59 ページの「[通知プロシージャ](#)」を参照してください。

属性データ型

text *

OCI_ATTR_SUBSCR_RECPTPRES

モード

読み込み / 書き込み

説明

クライアントが通知を受信するための表現です。有効な値は、OCI_SUBSCR_PRES_DEFAULT および OCI_SUBSCR_PRES_XML です。

設定しない場合、この属性は、OCI_SUBSCR_PRES_DEFAULT にデフォルト設定されます。

イベント通知を XML 表示で受信する場合、この属性は OCI_SUBSCR_PRES_XML に設定する必要があります。それ以外の場合は、属性を設定しないか、OCI_SUBSCR_PRES_DEFAULT に設定します。

属性データ型

ub4

OCI_ATTR_SUBSCR_RECPTPROTO

モード

読み込み / 書き込み

説明

クライアントが通知を受信するためのプロトコルです。有効な値は次のとおりです。

- OCI_SUBSCR_PROTO_OCI
- OCI_SUBSCR_PROTO_MAIL
- OCI_SUBSCR_PROTO_SERVER
- OCI_SUBSCR_PROTO_HTTP

OCI クライアントでイベント通知を受信する場合は、OCI_SUBSCR_PROTO_OCI に設定します。

電子メールをイベント通知で送信する場合は、OCI_SUBSCR_PROTO_MAIL に設定します。イベント通知によって PL/SQL プロシージャをデータベースで呼び出す場合は、OCI_SUBSCR_PROTO_SERVER に設定します。HTTP URL をイベント通知に送信する場合は、OCI_SUBSCR_PROTO_HTTP に設定します。

設定しない場合、この属性は、OCI_SUBSCR_PROTO_OCI にデフォルト設定されます。

OCI_SUBSCR_PROTO_OCI の場合は、サブスクリプション・ハンドルを登録コール OCISubscriptionRegister() に渡す前に、OCI_ATTR_SUBSCR_CALLBACK 属性と OCI_ATTR_SUBSCR_CTX 属性を設定しておく必要があります。

OCI_SUBSCR_PROTO_MAIL の場合は、サブスクリプション・ハンドルを登録コール OCISubscriptionRegister() に渡す前に、OCI_SUBSCR_PROTO_SERVER、OCI_SUBSCR_PROTO_HTTP および属性 OCI_ATTR_SUBSCR_RECPT を設定しておく必要があります。

属性データ型

ub4

ダイレクト・パス・ロード・ハンドル属性

関連項目： [ダイレクト・パス・ロードおよびダイレクト・パス・ハンドルの割当ての詳細は、12-2 ページの「ダイレクト・パス・ロードの概要」](#) および [12-16 ページの「オブジェクト型のダイレクト・パス・ロード」](#) を参照してください。

ダイレクト・パス・コンテキスト・ハンドル（OCIDirPathCtx）の属性

OCI_ATTR_BUF_SIZE

モード

読み込み / 書き込み

説明

ストリーム転送バッファのサイズを設定します。デフォルト値は 64KB です。

属性データ型

ub4 */ub4 *

OCI_ATTR_CHARSET_ID

モード

読込み / 書込み

説明

文字データのデフォルトのキャラクタ・セット ID です。このキャラクタ・セット ID は、列レベルで上書きできることに注意してください。キャラクタ・セット ID が列レベルまたは表レベルで指定されていない場合は、グローバル・サポート環境設定が使用されます。

属性データ型

ub2 */ub2 *

OCI_ATTR_DATEFORMAT

モード

読込み / 書込み

説明

SQLT_CHAR から DTYDAT に変換する際の、デフォルトの日付書式文字列です。この日付書式文字列は、列レベルで上書きできることに注意してください。日付書式文字列が列レベルまたは表レベルで指定されていない場合は、グローバル・サポート環境設定が使用されます。

属性データ型

text */text *

OCI_ATTR_DIRPATH_DCACHE_DISABLE

モード

読込み / 書込み

説明

この属性を 1 に設定して、容量を超えた日付キャッシュは無効になることを示します。デフォルト値は 0 です。これは、キャッシュ内の参照項目によってキャッシュのオーバーフローが継続されることを意味します。

関連項目： この属性および次の 4 つの属性については、12-14 ページの「OCI のダイレクト・パス・ロードでの日付キャッシュの使用」を参照してください。

属性データ型

ub1 */ub1 *

OCI_ATTR_DIRPATH_DCACHE_HITS

モード
読み込み

説明
日付キャッシュのヒット数を問い合わせます。

属性データ型
ub4 *

OCI_ATTR_DIRPATH_DCACHE_MISSES

モード
読み込み

説明
日付キャッシュに関する現在のミスの数を問い合わせます。

属性データ型
ub4 *

OCI_ATTR_DIRPATH_DCACHE_NUM

モード
読み込み

説明
日付キャッシュ内の現在のエントリ数を問い合わせます。

属性データ型
ub4 *

OCI_ATTR_DIRPATH_DCACHE_SIZE

モード
読み込み / 書き込み

説明
表に日付キャッシュのサイズ（要素数単位）を設定します。日付キャッシュを使用禁止にするには、デフォルト値の 0 を設定します。

属性データ型
ub4 */ub4 *

OCI_ATTR_DIRPATH_MODE

モード

読み込み / 書き込み

説明

ダイレクト・パス・コンテキストのモードです。

- OCI_DIRPATH_LOAD – ロード操作（デフォルト）
- OCI_DIRPATH_CONVERT – 変換専用操作

属性データ型

ub1 */ub1 *

OCI_ATTR_DIRPATH_NOLOG

モード

読み込み / 書き込み

説明

各セグメントの NOLOG 属性は、イメージ REDO と無効化 REDO のどちらが生成されるかを次のように判断します。

- 0 – ロードされるセグメントの属性を使用します。
- 1 – ログ記録を行いません。必要に応じて DDL 文を上書きします。

属性データ型

ub1 */ub1 *

OCI_ATTR_DIRPATH_OBJ_CONSTR

モード

読み込み / 書き込み

説明

次のように、置換可能なオブジェクト表のオブジェクト型を示します。

```
text *obj_type; /* stores an object type name */
OCIAttrSet((dvoid *)dpctx,
            (ub4)OCI_HTYPE_DIRPATH_CTX,
            (dvoid *) obj_type,
            (ub4)strlen((const char *) obj_type),
            (ub4)OCI_ATTR_DIRPATH_OBJ_CONSTR, errhp);
```

属性データ型

text **/text *

OCI_ATTR_DIRPATH_PARALLEL

モード

読み込み / 書き込み

説明

この値を 1 に設定すると、複数ロード・セッションで、共通のセグメントを同時にロードできます。デフォルトは 0（ゼロ）です（非パラレル）。

属性データ型

ub1 */ub1 *

OCI_ATTR_LIST_COLUMNS

モード

読み込み

説明

ダイレクト・パス・コンテキストに対応付けられた列リストのパラメータ記述子に、ハンドルを戻します。OCI_ATTR_NUM_COLS 属性で列数を設定した後で、列リスト・パラメータ記述子を取り出すことができます。

関連項目： A-69 ページ「[列パラメータ属性へのアクセス](#)」

属性データ型

OCIParam**

OCI_ATTR_NAME

モード

読み込み / 書き込み

説明

ロードされる表の名前です。

属性データ型

text **/text *

OCI_ATTR_NUM_COLS

モード

読み込み / 書き込み

説明

表にロードされる列の数です。

属性データ型

ub2 */ub2 *

OCI_ATTR_NUM_ROWS

モード

読み込み / 書き込み

説明

読み込み時: これまでにロードした行の数です。

書き込み時: ダイレクト・パスおよびダイレクト・パス関数の列配列に対して割り当てられた行の数です。

属性データ型

ub2 */ub2 *

OCI_ATTR_SCHEMA_NAME

モード

読み込み / 書き込み

説明

ロードされる表が存在するスキーマの名前です。指定しない場合は、デフォルトで、接続ユーザーのスキーマが使用されます。

属性データ型

text **/text *

OCI_ATTR_SUB_NAME

モード

読み込み / 書き込み

説明

ロードされるパーティションまたはサブパーティションの名前です。指定しない場合は、表全体がロードされます。名前は、その表に属する有効なパーティションまたはサブパーティションの名前にする必要があります。

属性データ型

text **/text *

ダイレクト・パス関数のコンテキスト・ハンドル（OCI DirPathFuncCtx）属性

属性の詳細な説明は、次を参照してください。

関連項目： 12-33 ページ [「ダイレクト・パス関数コンテキストと属性」](#)

OCI_ATTR_DIRPATH_EXPR_TYPE

モード

読み込み / 書き込み

説明

スカラー列以外の関数コンテキストの OCI_ATTR_NAME で指定された式の型を示します。

次の値が有効です。

- OCI_DIRPATH_EXPR_OBJ_CONSTR（列オブジェクトのオブジェクト型名）
- OCI_DIRPATH_EXPR_REF_TBLNAME（参照オブジェクトの表名）
- OCI_DIRPATH_EXPR_SQL（列値を導出するための SQL 文字列）

属性データ型

ub1 */ub1 *

OCI_ATTR_LIST_COLUMNS

モード

読み込み

説明

ダイレクト・パス関数コンテキストに対応付けられた列リストのパラメータ記述子に、ハンドルを戻します。列リストのパラメータ記述子は、列数（スカラー列以外の列に対応付けられた属性または引数の数）を設定した後に、OCI_ATTR_NUM_COLS 属性を使用して取り出すことができます。

関連項目： A-69 ページ「[列パラメータ属性へのアクセス](#)」

属性データ型

OCIParam**

OCI_ATTR_NAME

モード

読み込み / 書き込み

説明

関数コンテキストが列オブジェクトを記述している場合はオブジェクト型名、SQL 文字列を記述している場合は SQL 関数、REF 列を記述している場合は参照表名です。

属性データ型

text **/text *

OCI_ATTR_NUM_COLS

モード

読み込み / 書き込み

説明

列が列オブジェクトの場合は、ロードするオブジェクト属性の数です。列が SQL 文字列または REF 列の場合は、処理する引数の数です。このパラメータは、列リストが取り出される前に設定する必要があります。

属性データ型

ub2 */ub2 *

OCI_ATTR_NUM_ROWS

モード

読み込み

説明

これまでにロードした行の数です。

属性データ型

ub4 *

ダイレクト・パスおよびダイレクト・パス関数の列配列ハンドル (OCIDirPathColArray) 属性

OCI_ATTR_COL_COUNT

モード

読み込み

説明

最後に処理された行の最終列です。

属性データ型

ub2 *

OCI_ATTR_NUM_COLS

モード

読み込み

説明

列配列の列ディメンションです。

属性データ型

ub2 *

OCI_ATTR_NUM_ROWS

モード
読み込み

説明
列配列の行ディメンションです。

属性データ型
ub4 *

OCI_ATTR_ROW_COUNT

モード
読み込み

説明
処理される行数です。

属性データ型
ub4 *

ダイレクト・パス・ストリーム・ハンドル（OCIDirPathStream）属性

OCI_ATTR_BUF_ADDR

モード
読み込み

説明
ストリーム・データの先頭のバッファ・アドレスです。

属性データ型
ub1 **

OCI_ATTR_BUF_SIZE

モード
読み込み

説明
ストリーム・データのバイト単位のサイズです。

属性データ型
ub4 *

OCI_ATTR_ROW_COUNT

モード
読み込み

説明
前の OCIDirPathLoadStream() コールで処理される行数です。

属性データ型
ub4 *

OCI_ATTR_STREAM_OFFSET

モード
読み込み

説明
最後に処理された行の、ストリーム・バッファへのオフセットです。

属性データ型
ub4 *

ダイレクト・パス列パラメータ属性

アプリケーションでは、各列パラメータ記述子に対して属性を設定することにより、ロードする列、およびデータの外部書式を指定します。列パラメータ記述子は、OCIParamGet() によって、列パラメータ・リストのパラメータとして取得されます。表の列パラメータ・リストは、ダイレクト・パス・コンテキストの OCI_ATTR_LIST_COLUMNS 属性から取得されます。スカラー列以外の列の場合、列パラメータ・リストは、そのダイレクト・パス関数コンテキストの OCI_ATTR_LIST_COLUMNS 属性から取得されます。

すべてのパラメータは 1 を基準としていることに注意してください。

列パラメータ属性へのアクセス

次のコード例は、スカラー列のダイレクト・パス列パラメータ属性の使用方法を示しています。この属性にアクセスするには、最初に、ロードする列の数を設定して、OCI_ATTR_LIST_COLUMNS 属性から列パラメータ・リストを取得する必要があります。

関連項目： 12-9 ページの [スカラー列に対するダイレクト・パス・ロードの例](#) リストで定義されているデータ構造を参照してください。

```
...
/* set number of columns to be loaded */
OCI_CHECK(ctlp->errhp_ctl, OCI_HTYPE_ERROR, ociret, ctlp,
          OCIAAttrSet((dvoid *)dpctx, (ub4)OCI_HTYPE_DIRPATH_CTX,
                      (dvoid *)&tblp->ncol_tbl,
                      (ub4)0, (ub4)OCI_ATTR_NUM_COLS, ctlp->errhp_ctl));

/* get the column parameter list */
OCI_CHECK(ctlp->errhp_ctl, OCI_HTYPE_ERROR, ociret, ctlp,
          OCIAAttrGet((dvoid *)dpctx,
                     OCI_HTYPE_DIRPATH_CTX,
                     (dvoid *)&ctlp->colLstDesc_ctl, (ub4 *)0,
                     OCI_ATTR_LIST_COLUMNS, ctlp->errhp_ctl));
```

これで、パラメータ属性を設定できます。

```
/* set the attributes of each column by getting a parameter handle on each
 * column, then setting attributes on the parameter handle for the column.
 * Note that positions within a column list descriptor are 1-based. */

for (i = 0, pos = 1, colp = tblp->col_tbl, fldp = tblp->fld_tbl;
     i < tblp->ncol_tbl;
     i++, pos++, colp++, fldp++)
{
    /* get parameter handle on the column */
    OCI_CHECK(ctlp->errhp_ctl, OCI_HTYPE_ERROR, ociret, ctlp,
              OCIParmGet((CONST dvoid *)ctlp->colLstDesc_ctl,
                        (ub4)OCI_DTYPE_PARAM, ctlp->errhp_ctl,
                        (dvoid **)&colDesc, pos));

    colp->id_col = i;                               /* position in column array */
}
```

```
/* set external attributes on the column */
/* column name */
OCI_CHECK(ctlp->errhp_ctl, OCI_HTYPE_ERROR, ociret, ctlp,
          OCIAttrSet((dvoid *)colDesc, (ub4)OCI_DTYPE_PARAM,
                     (dvoid *)colp->name_col,
                     (ub4)strlen((const char *)colp->name_col),
                     (ub4)OCI_ATTR_NAME, ctlp->errhp_ctl));

/* column type */
OCI_CHECK(ctlp->errhp_ctl, OCI_HTYPE_ERROR, ociret, ctlp,
          OCIAttrSet((dvoid *)colDesc, (ub4)OCI_DTYPE_PARAM,
                     (dvoid *)&colp->exttyp_col, (ub4)0,
                     (ub4)OCI_ATTR_DATA_TYPE, ctlp->errhp_ctl));

/* max data size */
OCI_CHECK(ctlp->errhp_ctl, OCI_HTYPE_ERROR, ociret, ctlp,
          OCIAttrSet((dvoid *)colDesc, (ub4)OCI_DTYPE_PARAM,
                     (dvoid *)&fldp->maxlen_fld, (ub4)0,
                     (ub4)OCI_ATTR_DATA_SIZE, ctlp->errhp_ctl));

if (colp->datemask_col) /* set column (input field) date mask */
{
    OCI_CHECK(ctlp->errhp_ctl, OCI_HTYPE_ERROR, ociret, ctlp,
              OCIAttrSet((dvoid *)colDesc, (ub4)OCI_DTYPE_PARAM,
                         (dvoid *)colp->datemask_col,
                         (ub4)strlen((const char *)colp->datemask_col),
                         (ub4)OCI_ATTR_DATEFORMAT, ctlp->errhp_ctl));
}
if (colp->prec_col)
{
    OCI_CHECK(ctlp->errhp_ctl, OCI_HTYPE_ERROR, ociret, ctlp,
              OCIAttrSet((dvoid *)colDesc, (ub4)OCI_DTYPE_PARAM,
                         (dvoid *)&colp->prec_col, (ub4)0,
                         (ub4)OCI_ATTR_PRECISION, ctlp->errhp_ctl));
}
if (colp->scale_col)
{
    OCI_CHECK(ctlp->errhp_ctl, OCI_HTYPE_ERROR, ociret, ctlp,
              OCIAttrSet((dvoid *)colDesc, (ub4)OCI_DTYPE_PARAM,
                         (dvoid *)&colp->scale_col, (ub4)0,
                         (ub4)OCI_ATTR_SCALE, ctlp->errhp_ctl));
}
```



```
    }  
    if (colp->csid_col)  
    {  
        OCI_CHECK(ctlp->errhp_ctl, OCI_HTYPE_ERROR, ociret, ctlp,  
                  OCIAttrSet((dvoid *)colDesc, (ub4)OCI_DTYPE_PARAM,  
                              (dvoid *)&colp->csid_col, (ub4)0,  
                              (ub4)OCI_ATTR_CHARSET_ID, ctlp->errhp_ctl));  
    }  
    /* free the parameter handle to the column descriptor */  
    OCI_CHECK((dvoid *)0, 0, ociret, ctlp,  
              OCIDescriptorFree((dvoid *)colDesc, OCI_DTYPE_PARAM));  
}  
...
```

OCI_ATTR_CHARSET_ID

モード

読み込み / 書き込み

説明

キャラクタ列のキャラクタ・セット ID です。設定しない場合は、ダイレクト・パス・コンテキスト内に設定されたキャラクタ・セット ID がデフォルトとして使用されます。

属性データ型

ub2 */ub2 *

OCI_ATTR_DATA_SIZE

モード

読み込み / 書き込み

説明

列の外部データのバイト単位の最大サイズです。これは、変換バッファ・サイズに影響する可能性があります。

属性データ型

ub2 */ub2 *

OCI_ATTR_DATA_TYPE

モード

読み込み / 書き込み

説明

列の外部データ型を戻すか、または設定します。有効なデータ型は次のとおりです。

- SQLT_CHR
- SQLT_DATE
- SQLT_TIMESTAMP
- SQLT_TIMESTAMP_TZ
- SQLT_TIMESTAMP_LTZ
- SQLT_INTERVAL_YM
- SQLT_INTERVAL_DS
- SQLT_INT
- SQLT_UIN
- SQLT_FLT
- SQLT_PDN
- SQLT_BIN
- SQLT_NUM
- SQLT_NTY
- SQLT_REF
- SQLT_VST
- SQLT_VNU

属性データ型

ub2 */ub2 *

OCI_ATTR_DATEFORMAT

モード

読み込み / 書き込み

説明

列の日付変換マスクです。設定しない場合、日付書式は、デフォルトでダイレクト・パス・コンテキスト内に設定された日付変換マスクになります。

属性データ型

text **/text *

OCI_ATTR_DIRPATH_OID

モード

読み込み / 書き込み

説明

ロードする列は、オブジェクト表のオブジェクト ID 列であることを示します。

属性データ型

ub1 */ub1 *

OCI_ATTR_DIRPATH_SID

モード

読み込み / 書き込み

説明

ロードする列は、NESTED TABLE の設定 ID 列であることを示します。

属性データ型

ub1 */ub1 *

OCI_ATTR_NAME

モード

読み込み / 書き込み

説明

ロードされる列の名前を戻すか、または設定します。

属性データ型

text **/text *

OCI_ATTR_PRECISION

モード

読み込み / 書き込み

説明

精度を戻すか、または設定します。

属性データ型

明示的な記述の場合は、ub1 */ub1 *

暗黙的な記述の場合は、sb2 */sb2 *

OCI_ATTR_SCALE

モード

読み込み / 書き込み

説明

バック 10 進数およびゾーン 10 進数の入力データ型からの変換のスケール（小数点以下の桁数）を戻すか、または設定します。

属性データ型

sb1 */sb1 *

プロセス・ハンドル属性

共有システムのパラメータは、OCIAttrSet() コールおよび OCIAttrGet() コールを使用して設定および読み込みができます。使用するハンドル・タイプはプロセス・ハンドル OCI_HTYPE_PROC です。

関連項目： A-9 ページ「[OCI_ATTR_SHARED_HEAPALLOC](#)」

OCI_ATTR_MEMPOOL_APPNAME 属性、OCI_ATTR_MEMPOOL_HOMENAME 属性および OCI_ATTR_MEMPOOL_INSTNAME 属性は、アプリケーション名、ホーム名およびインスタンス名を指定します。これらの名前を一緒に使用して、プロセスを適切な共有プール領域にマップすることができます。これらの属性を設定しない場合は、内部デフォルト値が使用されます。特定の動作のための有効な属性設定を次に示します。

- インスタンス名、アプリケーション名（非修飾）：この設定では、特定の名前の実行可能ファイルのみが、共通の共有サブシステムに接続できます。たとえば、*Office* という名前の OCI アプリケーションが、*Office* が存在するディレクトリに関係なく、共通の共有サブシステムに接続できます。
- インスタンス名、ホーム名：この設定では、特定のホーム・ディレクトリ内の一連の実行可能ファイルが、共有サブシステムの同じインスタンスに接続できます。たとえば、ORACLE_HOME ディレクトリに存在するすべての OCI アプリケーションが、共通の共有サブシステムを使用できます。
- インスタンス名、ホーム名、アプリケーション名（非修飾）：この設定では、特定の実行可能ファイルのみが、共有サブシステムに接続できます。たとえば、ORACLE_HOME ディレクトリ内の *Office* という名前の 1 つのアプリケーションが、指定の共有サブシステムに接続できます。

OCI_ATTR_MEMPOOL_APPNAME

モード

読み込み / 書き込み

説明

実行可能ファイルのファイル名または完全修飾パス名です。

属性データ型

text *

OCI_ATTR_MEMPOOL_HOMENAME

モード

読み込み / 書き込み

説明

共通の共有サブシステム・インスタンスを使用する実行可能ファイルが存在するディレクトリ名です。

属性データ型

text *

OCI_ATTR_MEMPOOL_INSTNAME

モード

読み込み / 書き込み

説明

共有サブシステムのインスタンスを識別する、任意のユーザー定義名です。

属性データ型

text *

OCI_ATTR_MEMPOOL_SIZE

モード

読み込み / 書き込み

説明

共有プールのバイト単位のサイズです。この属性は次のように設定されます。

```
ub4 plsz = 1000000;  
OCIAttrSet((dvoid *)0, (ub4) OCI_HTYPE_PROC,  
            (dvoid *)&plsz, (ub4) 0, (ub4) OCI_ATTR_POOL_SIZE, 0);
```

属性データ型

ub4 *

OCI_ATTR_PROC_MODE

モード

読み込み

説明

現在設定されているすべてのプロセス・モードを戻します。読み込まれる値には、現在設定されているすべての OCI プロセス・モードの論理和をとった値が含まれます。特定のモードが設定されているかどうかを判断するには、そのモードで値の論理和をとります。次に例を示します。

```
ub4 mode;  
boolean is_shared;  
  
OCIAttrGet((dvoid *)0, (ub4)OCI_HTYPE_PROC,  
            (dvoid *) &mode, (ub4 *) 0,  
            (ub4)OCI_ATTR_PROC_MODE, 0);  
  
is_shared = mode | OCI_SHARED;
```

属性データ型

ub4 *

OCI デモ・プログラム

Oracle は、OCI コールの使用方法を示すコード例を提供しています。これらのプログラムはあくまでも例であり、すべてのプラットフォームで実行できるとは限りません。

デモ・プログラムは、Oracle インストレーションによって使用可能になります。デモ・プログラムの位置、名前および可用性は、プラットフォームによって異なります。UNIX ワークステーションでは、デモ・プログラムは \$ORACLE_HOME/rdbms/demo ディレクトリにインストールされています。Windows NT システムでは、デモ・プログラムは %ORACLE_HOME%\Oci\Samples ディレクトリにあります。

\$ORACLE_HOME/rdbms/demo ディレクトリには、デモ・プログラム以外に、Makefile という名前のファイルも含まれています。このファイルは、独自の OCI アプリケーションまたは外部プロシージャを作成するときのテンプレートとして使用してください。OCI アプリケーションまたは外部プロシージャを作成するために新規の Makefiles を開発する場合は、独自のマクロをリンク・ラインに追加して、提供されている Makefile をカスタマイズする必要があります。ただし、デモ・ファイル Makefile で提供されているマクロを保持する必要があります。これは、このファイルによって、新規に開発した Makefiles のメンテナンスが容易になるためです。Windows システムでの類似ファイルは、%ORACLE_HOME%\Oci\Samples ディレクトリ内の make.bat です。

特定のヘッダー・ファイルまたは SQL ファイルがアプリケーションで必要になった場合には、それらのファイルも用意されています。設定やプログラム実行のヒントは、デモ・プログラムの最初に記載されたコメントの情報を参照してください。

表 B-1「OCI デモ・プログラム」に、重要なデモ・プログラムおよびそのプログラムで示される OCI 機能を示します。

表 B-1 OCI デモ・プログラム

プログラム名	示される機能
cdemo81.c	Oracle8i OCI 以上の機能性を利用した基本 SQL 処理の使用
cdemo82.c	ユーザー定義オブジェクトの基本処理の実行
cdemocor.c	複合オブジェクト検索 (COR) を使用したパフォーマンス向上

表 B-1 OCI デモ・プログラム（続き）

プログラム名	示される機能
cdemodr1.c、 cdemodr2.c、 cdemodr3.c	基本データ型の LOB および REF で使用される RETURNING 句を伴う INSERT/UPDATE/DELETE 文の使用
cdemodsa.c	表の情報の記述
cdemodsc.c	オブジェクト型の情報の記述
cdemofo.c	アプリケーション・フェイルオーバー・コールバックの登録および操作
cdemolb.c	LOB データの作成と挿入およびその後のデータの読み込み、書き込み、コピー、追加および切捨て
cdemolb2.c	ストリーム・モードおよびコールバック関数を使用した CLOB/BLOB 列の書き込みおよび読み込み
cdemolbs.c	LOB バッファリング・システムを使用した LOB の書き込みおよび読み込み
cdemobj.c	REF オブジェクトの確保およびナビゲーション
cdemorid.c	INSERT/UPDATE/DELETE 文およびフェッチを使用した 1 ラウンドトリップでの複数 ROWID の取得
cdemoses.c	セッション切替えおよび移行の使用
cdemothr.c	OCIThread パッケージの使用
cdemosyev.c	事前定義済みサブスクリプションの登録およびクライアント通知用に呼び出すコールバック関数の指定（アドバンスト・キューイングの詳細は、『Oracle9i アプリケーション開発者ガイド - アドバンスト・キューイング』を参照してください）。
cdemodp.c、cdemdplp.c	ダイレクト・パス・ロード関数を使用したデータのロード
cdemdpco.c	ダイレクト・パス・ロード関数を使用した列オブジェクトのロード
cdemdpno.c	ダイレクト・パス・ロード関数を使用した、ネストされた列オブジェクトのロード
cdemdpin.c	ダイレクト・パス・ロード関数を使用した導出タイプ（継承）のロード
cdemdpit.c	継承があるオブジェクト表のロード
cdemdprou.c	ダイレクト・パス・ロード関数を使用した参照のロード
cdemdpss.c	ダイレクト・パス・ロード関数を使用した SQL 文字列のロード
cdemoucbl.c、 cdemoucbl.c	静的および動的ユーザー・コールバックの使用

表 B-1 OCI デモ・プログラム（続き）

プログラム名	示される機能
cdemoupk.c、 cdemoup1.c、 cdemoup2.c	複数パッケージでの動的ユーザー・コールバックの使用
cdemodt.c	日時および時間隔の例
cdemosc.c	スクロール・カーソル
cdemol2l.c	LONG API を使用した LOB へのアクセス
cdemoin1.c	表にある継承されたタイプを変更し、その表からレコードを表示する継承デモ
cdemoin2.c	属性の代入性を提示する継承デモ
cdemoin3.c	オブジェクト、継承されたタイプ、オブジェクト表およびサブ表を記述した継承デモ
cdemoanydata1.c	任意データ表との間で行の挿入と選択を行う任意データ・デモ
cdemoanydata2.c	OCITypeBeginCreate() を使用してタイプをピース単位に作成し、作成した新しいタイプを記述する任意データ・デモ
cdemosp.c	セッション・プーリング
cdemocp.c	接続プーリング
cdemoproxy.c	プロキシ機能を使用した接続プーリング
cdemostc.c	文キャッシュ

OCI 関数のサーバー・ラウンドトリップ

この付録では、様々な OCI コール中に発生するサーバー・ラウンドトリップについて説明します。この情報は、プログラマがアプリケーションで特定の作業を実行するために最も効率的な方法を判断するのに役立ちます。

この付録は、次の項目で構成されています。

- [サーバー・ラウンドトリップの概要](#)
- [リレーショナル関数のラウンドトリップ](#)
- [LOB 関数のラウンドトリップ](#)
- [オブジェクト関数およびキャッシュ関数のラウンドトリップ](#)
- [記述操作のラウンドトリップ](#)
- [データ型マッピング関数および操作関数によるラウンドトリップ](#)
- [任意型関数および任意データ関数のラウンドトリップ回数](#)
- [その他のローカル関数](#)

サーバー・ラウンドトリップの概要

この付録では、様々な OCI コール中に発生するサーバー・ラウンドトリップについて説明します。この情報は、アプリケーションで特定の作業を実行するための最も効率的な方法を判断するのに役立ちます。

リレーショナル関数のラウンドトリップ

OCI リレーショナル関数に必要なサーバー・ラウンドトリップ回数のリストを表 C-1 に示します。

表 C-1 リレーショナル関数用のサーバー・ラウンドトリップ

関数	サーバー・ラウンドトリップ回数
OCIBreak()	1
OCIEnvCreate()	0
OCIEnvInit()	0
OCIErrorGet()	0
OCIInitialize()	0
OCILdaToSvcCtx()	0
OCILogoff()	1
OCILogon()	1
OCIPasswordChange()	
OCIReset()	0
OCIServerAttach()	1
OCIServerDetach()	1
OCIServerVersion()	1
OCISessionBegin()	1
OCISessionEnd()	1
OCISvcCtxToLda()	0
OCITerminate()	1
OCIStmtExecute()	1
OCIStmtFetch()	0 または 1
OCIStmtGetPieceInfo()	1

表 C-1 リレーショナル関数用のサーバー・ラウンドトリップ (続き)

関数	サーバー・ラウンドトリップ回数
OCIStmtPrepare()	0
OCIStmtSetPieceInfo()	0
OCITransCommit()	1
OCITransDetach()	1
OCITransForget()	1
OCITransPrepare()	1
OCItransRollback()	1
OCITransStart()	1
OCIUserCallbackGet()	0
OCIUserCallbackregister()	0

LOB 関数のラウンドトリップ

OCILob*() コールで発生するサーバー・ラウンドトリップのリストを表 C-2 に示します。表の後に、読み込みおよび書き込みコールに関する情報を記載します。

表 C-2 OCILob*() コール用のサーバー・ラウンドトリップ

関数	サーバー・ラウンドトリップ回数
OCILobAppend()	1
OCILobAssign()	0
OCILobCharSetForm()	0
OCILobCharSetId()	0
OCILobCopy()	1
OCILobCreateTemporary()	1
OCILobDisableBuffering()	0
OCILobEnableBuffering()	0
OCILobErase()	1
OCILobFileClose()	1
OCILobFileCloseAll()	1
OCILobFileExists()	1

表 C-2 OCILob*() コール用のサーバー・ラウンドトリップ (続き)

関数	サーバー・ラウンドトリップ回数
OCILobFileName()	0
OCILobFileIsOpen()	1
OCILobFileOpen()	1
OCILobFileNameSet()	0
OCILobFlushBuffer()	この LOB のバッファにある変更ページごとに 1
OCILobFreeTemporary()	1
OCILobGetLength()	1
OCILobIsEqual()	0
OCILobIsTemporary()	0
OCILobLoadFromFile()	1
OCILobLocatorAssign()	ソース・ロケータまたは宛先ロケータ (あるいはその両方) が一時 LOB を参照している場合に 1
OCILobLocatorIsInit()	0
OCILobTrim()	1
OCILobOpen()	1
OCILobClose()	1
OCILobIsOpen()	1
OCILobGetChunkSize()	1

オブジェクト関数およびキャッシュ関数のラウンドトリップ

オブジェクト関数およびキャッシュ関数に必要なサーバー・ラウンドトリップ回数のリストを表 C-3 に示します。これらは、キャッシュがウォーム状態にあるときの値です。つまり、アプリケーションに必要な型記述子オブジェクトがロードされていることを想定しています。

表 C-3 オブジェクト関数およびキャッシュ関数用のサーバー・ラウンドトリップ

関数	サーバー・ラウンドトリップ回数
OCIObjectNew()	0
OCIObjectPin()	1。必要なオブジェクトがすでにキャッシュにある場合は 0。
OCIObjectUnpin()	0
OCIObjectPinCountReset()	0
OCIObjectLock()	1
OCIObjectMarkUpdate()	0
OCIObjectUnmark()	0
OCIObjectUnmarkByRef()	0
OCIObjectFree()	0
OCIObjectMarkDelete()	0
OCIObjectMarkDeleteByRef()	0
OCIObjectFlush()	1
OCIObjectRefresh()	1
OCIObjectCopy()	0
OCIObjectGetTypeRef()	0
OCIObjectGetObjectRef()	0
OCIObjectGetInd()	0
OCIObjectExists()	0
OCIObjectIsLocked()	0
OCIObjectIsDirty()	0
OCIObjectPinTable()	1
OCIObjectArrayPin()	1
OCICacheFlush()	1

表 C-3 オブジェクト関数およびキャッシュ関数用のサーバー・ラウンドトリップ（続き）

関数	サーバー・ラウンドトリップ回数
OCICacheRefresh()	1
OCICacheUnpin()	0
OCICacheFree()	0
OCICacheUnmark()	0

記述操作のラウンドトリップ

OCIDecribeAny()、OCIAttrGet() および OCIParamGet() に必要なサーバー・ラウンドトリップの回数のリストを表 C-4 に示します。

表 C-4 記述操作用のサーバー・ラウンドトリップ

関数	サーバー・ラウンドトリップ回数
OCIDecribeAny()	型記述子オブジェクトの REF 取得のために 1 ラウンドトリップ
OCIAttrGet()	型オブジェクトがオブジェクト・キャッシュにない場合は、型を記述するために 2 ラウンドトリップ 各コレクション要素、型属性、メソッドまたはメソッド引数記述子につき 1 ラウンドトリップ。コレクション要素、型属性またはメソッド引数に対して OCI_ATTR_TYPE_NAME または OCI_ATTR_SCHEMA_NAME を使用している場合は、さらに 1 ラウンドトリップ。 最初の OCIAttrGet() コールの後で、オブジェクト・キャッシュに記述されるすべての型オブジェクトがすでにある場合は 0
OCIParamGet()	0

データ型マッピング関数および操作関数によるラウンドトリップ

データ型マッピング関数および操作関数によるラウンドトリップ回数のリストを[表 C-5](#)に示します。表内のアスタリスクは、特定の接頭辞を持つすべての関数のサーバー・ラウンドトリップ回数が同じになることを示します。たとえば、`OCINumberAdd()`、`OCINumberPower()` および `OCINumberFromText()` では、サーバー・ラウンドトリップはすべて 0 となります。

表 C-5 データ型の操作関数用のサーバー・ラウンドトリップ

関数	サーバー・ラウンドトリップ回数
<code>OCINumber*()</code>	0
<code>OCIDate*()</code>	0
<code>OCIString*()</code>	0
<code>OCIRaw*()</code>	0
<code>OCIRef*()</code>	0
<code>OCIColl*()</code>	0。コレクションがキャッシュにロードされていない場合は 1。
<code>OCITable*()</code>	0。NESTED TABLE がキャッシュにロードされていない場合は 1。
<code>OCIIter*()</code>	0。コレクションがキャッシュにロードされていない場合は 1。

任意型関数および任意データ関数のラウンドトリップ回数

任意型関数および任意データ関数に必要なサーバー・ラウンドトリップ回数のリストを[表 C-6](#)に示します。リストされていない関数は、ラウンドトリップ回数を生成しません。

表 C-6 任意型関数および任意データ関数のサーバー・ラウンドトリップ回数

関数	サーバー・ラウンドトリップ回数
<code>OCIAnyDataAttrGet()</code>	0。型情報がキャッシュにロードされていない場合は 1。
<code>OCIAnyDataAttrSet()</code>	0。型情報がキャッシュにロードされていない場合は 1。
<code>OCIAnyDataCollGetElem()</code>	0。型情報がキャッシュにロードされていない場合は 1。

その他のローカル関数

表 C-7 に示す関数はローカルであるため、サーバー・ラウンドトリップは不要です。

表 C-7 ローカルに処理される関数

ローカル関数名	注意
OCIAttrGet()	オブジェクト型を記述すると、このコールは、型記述子オブジェクトをフェッチするために 1 回ラウンドトリップを行います。
OCIAttrSet()	
OCIBindByName()	
OCIBindByPos()	
OCIBindDynamic()	
OCIBindObject()	
OCIBindArrayOfStruct()	
OCIDefineByPos()	
OCIDefineDynamic()	
OCIDefineArrayOfStruct()	
OCIDefineObject()	
OCIDescriptorAlloc()	
OCIDescriptorFree()	
OCIEnvInit()	
OCIEnvCreate()	
OCIErrorGet()	
OCIHandleAlloc()	
OCIHandleFree()	
OCILdaToSvcCtx()	
OCISvcCtxToLda()	
OCIStmtGetBindInfo()	
OCIStmtPrepare()	
OCIStmtGetBindInfo()	

表 C-7 ローカルに処理される関数（続き）

ローカル関数名	注意
OCIStmtPrepare()	
OCIStmtFetch()	プリフェッチされた行を取り出す場合はローカル

索引

数字

- 2 次メモリー
 - オブジェクト, 13-17
- 3 層アーキテクチャ
 - スレッド・セーフティ, 9-2

A

- ADO, 「属性記述子オブジェクト」を参照
- ADT, 「オブジェクト型」を参照
- AQ, 「アドバンスト・キューイング」を参照

B

- BFILE
 - データ型, 3-20
- BLOB
 - データ型, 3-21
- BLOB (バイナリ・ラージ・オブジェクト)
 - データ型, 3-21

C

- C++ 言語
 - OCI のサポート, xxxi
- CASE OTT パラメータ, 14-30
- CHAR
 - 外部データ型, 3-16
- CHARZ
 - 外部データ型, 3-17
- checkerr() 関数
 - コード・リスト, 2-32
- CLOB
 - データ型, 3-21

- CODE OTT パラメータ, 14-29
- CONFIG OTT パラメータ, 14-30
- COR, 「複合オブジェクト検索」を参照
- C 言語
 - OCI のサポート, xxxi
- C データ型
 - OCI での操作, 11-4

D

- DATE
 - 外部データ型, 3-14
- DATE、ANSI
 - データ型, 3-22
- DDL, 「データ定義言語」を参照
- DML, 「データ操作言語」を参照

E

- ERRTYPE OTT パラメータ, 14-30

F

- FILE
 - OS ファイルとの関連付け, 7-3
 - データ型, 3-20
- FLOAT
 - 外部データ型, 3-11

G

- GTRID, 「トランザクション識別子」を参照

H

HFILE OTT パラメータ, 14-30

I

INITFILE OTT パラメータ, 14-29

INITFUNC OTT パラメータ, 14-29

INTEGER

外部データ型, 3-11

INTERVAL DAY TO SECOND データ型, 3-23

INTERVAL YEAR TO MONTH データ型, 3-23

INTYPE OTT パラメータ, 14-28

intype ファイル

OTT の実行時に提供, 14-8

構造, 14-33

L

LOB

amount パラメータおよび offset パラメータ, 16-23

OCI 関数, 7-5

一時オブジェクトの属性, 7-4

一時 LOB の継続時間, 7-18

外部データ型, 3-19

可変幅キャラクタ・セット, 16-23

キャラクタ・セット, 16-23

コールバック, 7-13

固定幅キャラクタ・セット, 16-23

作成, 7-2

定義, 5-22

データのフェッチ, 4-16

テンポラリ, 7-17

テンポラリの解放, 7-18

テンポラリの作成, 7-18

テンポラリの例, 7-19

バインド, 5-10

バッファリング, 7-11

変更, 7-2

ロケータ, 2-17

LOB 関数, 16-22

サーバー・ラウンドトリップ回数, C-3

LOB 操作のバッファリング, 7-11

LOB ロケータ, 2-17

属性, A-41

LONG

外部データ型, 3-13

LONG RAW

外部データ型, 3-15

LONG VARCHAR

外部データ型, 3-16

LONG VARRAW

外部データ型, 3-16

M

Makefile (UNIX), B-1

MDO, 「メソッド記述子オブジェクト」を参照

N

NCHAR

問題, 5-34

NCLOB

データ型, 3-21

NESTED TABLE

操作用関数, 11-25

ダイレクト・パス・ロード, 12-16

要素順序, 11-25

NLS_LANG, 2-48

NLS_NCHAR, 2-48

no-op

定義, 17-22

NOT FINAL オブジェクト表

ダイレクト・パス・ロード, 12-30

NULL

アトミック, 10-30

インジケータ変数を使用して挿入, 2-36

検出, 2-37

挿入, 2-37

データベースに挿入, 2-36

NULL インジケータ構造体, 10-30

OTT で生成された, 10-8

NULL かどうか

オブジェクト, 10-30

NUMBER

外部データ型, 3-10

O

Object Type Translator

OCI での使用方法, 10-8

「OTT」を参照

サンプル出力, 10-8

OCI

オブジェクトのアクセスと操作, 14-23

オブジェクトのサポート, 1-6

概要, 1-2

コールの終了, 2-38

分類, 1-5

利点, 1-3

OCI_ATTR_ALLOC_DURATION

環境ハンドル属性, A-8

OCI_ATTR_APPCTX_ATTR, 8-21, A-17

OCI_ATTR_APPCTX_LIST, 8-21, A-17

OCI_ATTR_APPCTX_NAME, 8-21

OCI_ATTR_APPCTX_SIZE, 8-21, A-18

OCI_ATTR_APPCTX_VALUE, 8-22, A-18

OCI_ATTR_AUTOCOMMIT_DDL

属性, 6-21

OCI_ATTR_BIND_DN, A-3

OCI_ATTR_BUF_ADDR, A-67

OCI_ATTR_BUF_SIZE, A-58, A-68

OCI_ATTR_CACHE

属性, 6-15

OCI_ATTR_CACHE_ARRAYFLUSH, 13-11

環境ハンドル属性, A-3

OCI_ATTR_CACHE_MAX_SIZE

環境ハンドル属性, A-4

OCI_ATTR_CACHE_OPT_SIZE

環境ハンドル属性, A-4

OCI_ATTR_CATALOG_LOCATION

属性, 6-20

OCI_ATTR_CERTIFICATE, A-18

OCI_ATTR_CERTIFICATE_TYPE, A-18

OCI_ATTR_CHAR_COUNT

定義ハンドル属性, A-37

バインド・ハンドル属性, A-34

OCI_ATTR_CHAR_SIZE, 6-15

属性, 6-21

OCI_ATTR_CHAR_USED, 6-15

属性, 6-21

OCI_ATTR_CHARSET_FORM, 5-34, 6-18

属性, 6-12, 6-14, 6-16

定義ハンドル属性, A-37

バインド・ハンドル属性, A-34

OCI_ATTR_CHARSET_ID, 5-34, A-59, A-71

属性, 6-11, 6-14, 6-16, 6-18, 6-20

定義ハンドル属性, A-38

バインド・ハンドル属性, A-35

OCI_ATTR_CLIENT_IDENTIFIER, 8-17, A-19

OCI_ATTR_CLUSTERED

属性, 6-7

OCI_ATTR_COL_COUNT, A-66

OCI_ATTR_COLLECTION_ELEMENT

属性, 6-10

OCI_ATTR_COLLECTION_TYPECODE

属性, 6-9

OCI_ATTR_COMPLEXOBJECT_COLL_OUTOFLINE

COR ハンドル属性, A-42

OCI_ATTR_COMPLEXOBJECT_LEVEL

COR ハンドル属性, A-41

OCI_ATTR_COMPLEXOBJECTCOMP_TYPE_LEVEL

COR 記述子の属性, A-42

OCI_ATTR_COMPLEXOBJECTCOMP_TYPE

COR 記述子の属性, A-42

OCI_ATTR_CONN_BUSY_COUNT, A-22

OCI_ATTR_CONN_INCR, A-23

OCI_ATTR_CONN_MAX, A-22

OCI_ATTR_CONN_MIN, A-22

OCI_ATTR_CONN_NOWAIT, A-21, A-22

OCI_ATTR_CONN_OPEN_COUNT, A-22

OCI_ATTR_CONN_TIMEOUT, A-21

OCI_ATTR_CURRENT_POSITION

属性, 4-18, A-27

OCI_ATTR_CURSOR_COMMIT_BEHAVIOR

属性, 6-20

OCI_ATTR_DATA_SIZE, 6-15, 6-21, A-71

属性, 6-11, 6-13, 6-15, 6-17

OCI_ATTR_DATA_TYPE, A-72

属性, 6-11, 6-13, 6-16, 6-17

OCI_ATTR_DATE_FORMAT, A-59

OCI_ATTR_DATEFORMAT, A-73

OCI_ATTR_DBA

属性, 6-7

OCI_ATTR_DESC_PUBLIC, 15-93

OCI_ATTR_DIRPATH_EXPR_TYPE 関数コンテキスト 属性, 12-35

OCI_ATTR_DIRPATH_EXPR_TYPE ダイレクト・パス 関数属性, A-64

OCI_ATTR_DIRPATH_NOLOG, A-61

OCI_ATTR_DIRPATH_OBJ_CONSTR, A-61

OCI_ATTR_DIRPATH_OBJ_CONSTR ダイレクト・パス コンテキスト属性, 12-33

OCI_ATTR_DIRPATH_OID, A-73

OCI_ATTR_DIRPATH_PARALLEL, A-62

OCI_ATTR_DIRPATH_SID 列配列属性, 12-40

OCI_ATTR_DISTINGUISHED_NAME, 8-15, 8-16,
A-19
OCI_ATTR_DML_ROW_OFFSET
エラー・ハンドル属性, A-10
OCI_ATTR_DN_COUNT, A-53
OCI_ATTR_DURATION
属性, 6-7
OCI_ATTR_ENCAPSULATION
属性, 6-12
OCI_ATTR_ENV, A-27
サーバー・ハンドル属性, A-13
サービス・コンテキスト・ハンドル属性, A-11
OCI_ATTR_ENV_CHARSET_ID, 2-50
環境ハンドル属性, A-4
OCI_ATTR_ENV_NCHARSET_ID, 2-50
環境ハンドル属性, A-5
OCI_ATTR_ENV_UTF16
環境ハンドル属性, A-5
OCI_ATTR_EXTERNAL_NAME, 8-7
サーバー・ハンドル属性, A-13
OCI_ATTR_FOCBK
サーバー・ハンドル属性, A-14
OCI_ATTR_FSPRECISION
属性, 6-12
OCI_ATTR_HAS_DEFAULT
属性, 6-17
OCI_ATTR_HAS_FILE
属性, 6-9
OCI_ATTR_HAS_LOB
属性, 6-9
OCI_ATTR_HAS_NESTED_TABLE
属性, 6-9
OCI_ATTR_HEAPALLOC
環境ハンドル属性, A-8
OCI_ATTR_HW_MARK
属性, 6-15
OCI_ATTR_IN_V8_MODE
サーバー・ハンドル属性, A-14
サービス・コンテキスト・ハンドル属性, A-11
OCI_ATTR_INCR
属性, 6-15
OCI_ATTR_INDEX_ONLY
属性, 6-7
OCI_ATTR_INITIAL_CLIENT_ROLES, 8-16, A-19
OCI_ATTR_INTERNAL_NAME, 8-7
サーバー・ハンドル属性, A-14

OCI_ATTR_IOMODE
属性, 6-18
OCI_ATTR_IS_CONSTRUCTOR
属性, 6-12
OCI_ATTR_IS_DESTRUCTOR
属性, 6-12
OCI_ATTR_IS_FINAL_METHOD
属性, 6-13
OCI_ATTR_IS_FINAL_TYPE
属性, 6-10
OCI_ATTR_IS_INCOMPLETE_TYPE
属性, 6-9
OCI_ATTR_IS_INSTANTIABLE_METHOD
属性, 6-13
OCI_ATTR_IS_INSTANTIABLE_TYPE
属性, 6-10
OCI_ATTR_IS_INVOKER_RIGHTS
属性, 6-8, 6-10
OCI_ATTR_IS_MAP
属性, 6-12
OCI_ATTR_IS_NULL
属性, 6-16, 6-18
OCI_ATTR_IS_OPERATOR
属性, 6-12
OCI_ATTR_IS_ORDER
属性, 6-12
OCI_ATTR_IS_OVERRIDING_METHOD
属性, 6-13
OCI_ATTR_IS_PREDEFINED_TYPE
属性, 6-9
OCI_ATTR_IS_RNDS
属性, 6-12
OCI_ATTR_IS_RNPS
属性, 6-12
OCI_ATTR_IS_SELFISH
属性, 6-12
OCI_ATTR_IS_SUBTYPE
属性, 6-10
OCI_ATTR_IS_SYSTEM_GENERATED_TYPE
属性, 6-9
OCI_ATTR_IS_SYSTEM_TYPE
属性, 6-9
OCI_ATTR_IS_TEMPORARY
属性, 6-7
OCI_ATTR_IS_TRANSIENT_TYPE
属性, 6-9

OCI_ATTR_IS_WNDS
属性, 6-12

OCI_ATTR_IS_WNPS
属性, 6-13

OCI_ATTR_LDAP_AUTH, A-5

OCI_ATTR_LDAP_CRED, A-6

OCI_ATTR_LDAP_CTX, A-6

OCI_ATTR_LDAP_HOST, A-6

OCI_ATTR_LDAP_PORT, A-7

OCI_ATTR_LEVEL
属性, 6-17

OCI_ATTR_LFPRECISION
属性, 6-12

OCI_ATTR_LINK
属性, 6-14, 6-18

OCI_ATTR_LIST_ARGUMENTS
属性, 6-8, 6-12

OCI_ATTR_LIST_COLUMNS, A-62
属性, 6-7

OCI_ATTR_LIST_COLUMNS ダイレクト・パス関数コ
ンテキスト属性, A-65

OCI_ATTR_LIST_OBJECTS
属性, 6-19

OCI_ATTR_LIST_SCHEMAS
属性, 6-20

OCI_ATTR_LIST_SUBPROGRAMS
属性, 6-8

OCI_ATTR_LIST_TYPE
属性, 6-19

OCI_ATTR_LIST_TYPE_ATTRS
属性, 6-10

OCI_ATTR_LIST_TYPE_METHODS
属性, 6-10

OCI_ATTR_LOBEMPTY
LOB ロケータ属性, A-41

OCI_ATTR_LOCKING_MODE
属性, 6-21

OCI_ATTR_MAP_METHOD
属性, 6-10

OCI_ATTR_MAX
属性, 6-15

OCI_ATTR_MAX_CATALOG_NAMELEN
属性, 6-20

OCI_ATTR_MAX_COLUMN_NAMELEN
属性, 6-20

OCI_ATTR_MAX_PROC_NAMELEN
属性, 6-20

OCI_ATTR_MAXCHAR_SIZE, A-35, A-38
属性, 5-37

OCI_ATTR_MAXCHAR_SIZE 属性, 5-37

OCI_ATTR_MAXDATA_SIZE
属性, 5-37
バインドで使用, 5-36
バインド・ハンドル属性, A-36

OCI_ATTR_MEMPOOL_APPNAME, A-75

OCI_ATTR_MEMPOOL_HOMENAME, A-76

OCI_ATTR_MEMPOOL_INSTNAME, A-76

OCI_ATTR_MEMPOOL_SIZE, A-76

OCI_ATTR_MIGSESSION
ユーザー・セッション・ハンドル属性, A-20

OCI_ATTR_MIN
属性, 6-15

OCI_ATTR_NAME, 12-34, A-62, A-74
属性, 6-8, 6-10, 6-11, 6-12, 6-13, 6-14, 6-16,
6-17

OCI_ATTR_NAME 関数コンテキスト属性, 12-34

OCI_ATTR_NAME ダイレクト・パス関数コンテキ
スト属性, A-65

OCI_ATTR_NAME 列配列属性, 12-38

OCI_ATTR_NCHARSET_ID
属性, 6-20

OCI_ATTR_NONBLOCKING_MODE
サーバー・ハンドル属性, 2-42, A-15

OCI_ATTR_NOWAIT_SUPPORT
属性, 6-21

OCI_ATTR_NUM_COLS, 12-36, A-63, A-66
属性, 6-7

OCI_ATTR_NUM_COLS ダイレクト・パス関数コン
テキスト属性, 12-36, A-65

OCI_ATTR_NUM_DML_ERRORS, A-27

OCI_ATTR_NUM_ELEMENTS
属性, 6-13

OCI_ATTR_NUM_HANDLES
属性, 6-19

OCI_ATTR_NUM_PARAMS
属性, 6-5

OCI_ATTR_NUM_ROWS, A-67

OCI_ATTR_NUM_ROWS 関数コンテキスト属性,
12-37

OCI_ATTR_NUM_ROWS 属性, 12-41

OCI_ATTR_NUM_ROWS ダイレクト・パス関数コ
ンテキスト属性, A-66

OCI_ATTR_NUM_ROWS ダイレクト・パス・コン
テキスト属性, A-63

OCI_ATTR_NUM_TYPE_ATTRS	OCI_ATTR_PRECISION, A-74
属性, 6-10	属性, 6-5, 6-11, 6-13, 6-16, 6-17
OCI_ATTR_NUM_TYPE_METHODS	OCI_ATTR_PREFETCH_MEMORY
属性, 6-10	文ハンドル属性, A-29
OCI_ATTR_OBJ_ID	OCI_ATTR_PREFETCH_MEMORY 文ハンドル属性,
属性, 6-5	A-29
OCI_ATTR_OBJ_NAME	OCI_ATTR_PREFETCH_ROWS
属性, 6-5	文ハンドル属性, A-29
OCI_ATTR_OBJ_SCHEMA	OCI_ATTR_PROC_MODE, A-77
属性, 6-5	OCI_ATTR_PROXY_CREDENTIALS, 8-15, A-20
OCI_ATTR_OBJECT	OCI_ATTR_PTYPE
環境ハンドル属性, A-7	属性, 6-6
OCI_ATTR_OBJECT_DETECTCHANGE, 13-14	OCI_ATTR_RADIX
環境ハンドル属性, 13-14, A-9	属性, 6-18
OCI_ATTR_OBJECT_NEWNOTNULL, 17-45	OCI_ATTR_REF_TDO
環境ハンドル属性, A-8	属性, 6-7, 6-9, 6-11, 6-14, 6-16, 6-18
OCI_ATTR_OBJID	OCI_ATTR_ROW_COUNT, 4-18, A-29, A-67, A-68
属性, 6-7, 6-14, 6-15	OCI_ATTR_ROWID
OCI_ATTR_ORDER	文ハンドル属性, A-30
属性, 6-15	OCI_ATTR_ROWS_FETCHED, 4-18, A-30
OCI_ATTR_ORDER_METHOD	OCI_ATTR_ROWS_RETURNED
属性, 6-10	コールバックに使用, 5-33
OCI_ATTR_OVERLOAD	バインド・ハンドル属性, A-37
属性, 6-8	OCI_ATTR_SAVEPOINT_SUPPORT
OCI_ATTR_PARAM	属性, 6-20
記述ハンドル属性, A-40	OCI_ATTR_SCALE, A-74
属性自体が記述子である場合に使用, 15-49	属性, 6-11, 6-13, 6-16, 6-17
OCI_ATTR_PARAM_COUNT	OCI_ATTR_SCHEMA_NAME, A-63
記述ハンドル属性, A-40	属性, 6-10, 6-11, 6-14, 6-16, 6-18
OCI_ATTR_PARAM_COUNT 文ハンドル属性, A-28	OCI_ATTR_SEQ
OCI_ATTR_PARSE_ERROR_OFFSET, A-34	属性, 6-15
OCI_ATTR_PARSE_ERROR_OFFSET 文ハンドル属性,	OCI_ATTR_SERVER
A-28	サービス・コンテキスト・ハンドル属性, A-12
OCI_ATTR_PARTITIONED	OCI_ATTR_SERVER_DN, A-54
属性, 6-7	OCI_ATTR_SERVER_DNS, A-54
OCI_ATTR_PASSWORD, 8-20	OCI_ATTR_SERVER_GROUP
ユーザー・セッション・ハンドル属性, A-20	サーバー・ハンドル属性, A-15
OCI_ATTR_PDPRC, A-36	OCI_ATTR_SERVER_STATUS
OCI_ATTR_PDSCL	サーバー・ハンドル属性, A-15
バインド・ハンドル属性, A-36, A-39	OCI_ATTR_SESSION
OCI_ATTR_PIN_DURATION	サービス・コンテキスト・ハンドル属性, A-12
環境ハンドル属性, A-8	OCI_ATTR_SHARED_HEAP_ALLOC
OCI_ATTR_PINOPTION	環境ハンドル属性, A-9
環境ハンドル属性, A-7	OCI_ATTR_SQLFNCODE
OCI_ATTR_POSITION	文ハンドル属性, A-30
属性, 6-17	OCI_ATTR_STATEMENT 文ハンドル属性, A-33
	OCI_ATTR_STMT_STATE, A-33

OCI_ATTR_STMT_TYPE
 文ハンドル属性, A-33
 OCI_ATTR_STMTCACHESIZE, 9-30, 15-36, A-12
 OCI_ATTR_STREAM_OFFSET, A-68
 OCI_ATTR_SUB_NAME, A-64
 属性, 6-18
 OCI_ATTR_SUBSCR_CALLBACK, A-54
 OCI_ATTR_SUBSCR_CTX, A-55
 OCI_ATTR_SUBSCR_NAME, A-55
 OCI_ATTR_SUBSCR_NAMESPACE, A-56
 OCI_ATTR_SUBSCR_PAYLOAD, A-56
 OCI_ATTR_SUBSCR_RECPT, A-56
 OCI_ATTR_SUBSCR_RECPTPRES, A-57
 OCI_ATTR_SUBSCR_RECPTPROTO, A-57
 OCI_ATTR_SUBSCR_SERVER_DN 記述子ハンドル,
 9-57
 OCI_ATTR_SUPERTYPE_NAME
 属性, 6-10
 OCI_ATTR_SUPERTYPE_SCHEMA_NAME
 属性, 6-10
 OCI_ATTR_TABLESPACE
 属性, 6-7
 OCI_ATTR_TIMESTAMP
 属性, 6-6
 OCI_ATTR_TRANS
 サービス・コンテキスト・ハンドル属性, A-13
 OCI_ATTR_TRANS_NAME, 8-4
 トランザクション・ハンドル属性, A-26
 OCI_ATTR_TRANS_TIMEOUT
 トランザクション・ハンドル属性, A-26
 OCI_ATTR_TYPE_NAME
 属性, 6-11, 6-14, 6-16, 6-18
 OCI_ATTR_TYPECODE
 属性, 6-9, 6-11, 6-13, 6-17
 OCI_ATTR_USERNAME
 ユーザー・セッション・ハンドル属性, A-21
 OCI_ATTR_VERSION
 属性, 6-20
 OCI_ATTR_WALL_LOC, A-10
 OCI_ATTR_XID, 8-4
 トランザクション・ハンドル属性, A-26
 OCI_CONTINUE, 2-31
 OCI_CPOOL_REINITIALIZE, 15-6
 OCI_CRED_PROXY, 8-15
 OCI_CRED_RDBMS, 8-15
 OCI_DEFAULT, 9-3, 15-6
 OCI_DIRPATH_CONVERT, A-61
 OCI_DIRPATH_DATASAVE_FINISH, 16-120
 OCI_DIRPATH_DATASAVE_SAVEONLY, 16-120
 OCI_DIRPATH_OID 列配列属性, 12-40
 OCI_DTYPE_AQAGENT, 2-15
 OCI_DTYPE_AQDEQ_OPTIONS, 2-15
 OCI_DTYPE_AQENQ_OPTIONS, 2-15
 OCI_DTYPE_AQMSG_PROPERTIES, 2-15
 OCI_DTYPE_AQNFY, 2-15
 OCI_DTYPE_COMPLEXOBJECTCOMP, 2-15
 OCI_DTYPE_DATE, 2-15
 OCI_DTYPE_FILE, 2-15
 OCI_DTYPE_INTERVAL_DS, 2-15
 OCI_DTYPE_INTERVAL_YM, 2-15
 OCI_DTYPE_LOB, 2-15
 OCI_DTYPE_PARAM, 2-15, 15-48, 15-61
 コード例での使用, 4-13
 使用される場合, 15-49
 OCI_DTYPE_ROWID, 2-15
 OCI_DTYPE_SNAP, 2-15
 OCI_DTYPE_SRVDN, 2-15
 OCI_DTYPE_TIMESTAMP, 2-15
 OCI_DTYPE_TIMESTAMP_LTZ, 2-15
 OCI_DTYPE_TIMESTAMP_TZ, 2-15
 OCI_DURATION_SESSION, 13-8, 16-24, 19-10,
 20-7, 20-21, 20-27, 20-37
 OCI_DURATION_STATEMENT, 16-24, 19-10, 20-7,
 20-21, 20-27, 20-37
 OCI_DURATION_TRANS, 13-8
 OCI_ERROR, 2-31, 8-8
 OCI_EVENTS
 通知受信のモード, 9-54
 OCI_EXT_CRED, 8-15
 OCI_HTYPE_AUTHINFO, 2-6, 9-25
 OCI_HTYPE_BIND, 2-6
 OCI_HTYPE_COMPLEXOBJECT, 2-6
 OCI_HTYPE_COR, 15-61
 OCI_HTYPE_CPOOL, 2-6, 9-15
 OCI_HTYPE_DEFINE, 2-6
 OCI_HTYPE_DESCRIBE, 2-6
 OCI_HTYPE_DIRPATH_COLUMN_ARRAY, 2-6
 OCI_HTYPE_DIRPATH_CTX, 2-6
 OCI_HTYPE_DIRPATH_FN_CTX, 2-6
 OCI_HTYPE_DIRPATH_STREAM, 2-6
 OCI_HTYPE_ENV, 2-6
 OCI_HTYPE_ERROR, 2-6
 OCI_HTYPE_PROC, 2-6
 OCI_HTYPE_SERVER, 2-6

OCI_HTYPE_SESSION, 2-6
 OCI_HTYPE_SPOOL, 2-6
 OCI_HTYPE_STMT, 2-6, 15-48, 15-61
 OCI_HTYPE_SUBSCRIPTION, 2-6
 OCI_HTYPE_SVCCTX, 2-6
 OCI_HTYPE_TRANS, 2-6
 OCI_INVALID_HANDLE, 2-31
 OCI_LOCK_NONE, 13-13
 OCI_LOCK_X, 13-13
 OCI_LOCK_X_NOWAIT, 13-13, 13-14
 パラメータの使用方法, 13-14
 OCI_LTYPE_ARG_FUNC リスト属性, 6-19
 OCI_LTYPE_ARG_PROC リスト属性, 6-19
 OCI_LTYPE_DB_SCH リスト属性, 6-19
 OCI_LTYPE_SCH_OBJ リスト属性, 6-19
 OCI_LTYPE_SUBPRG リスト属性, 6-19
 OCI_LTYPE_TYPE_ARG_FUNC リスト属性, 6-19
 OCI_LTYPE_TYPE_ARG_PROC リスト属性, 6-19
 OCI_LTYPE_TYPE_ATTR リスト属性, 6-19
 OCI_LTYPE_TYPE_METHOD リスト属性, 6-19
 OCI_MIGRATE, 8-11
 OCI_NEED_DATA, 2-31
 OCI_NEW_LENGTH_SEMANTICS, 15-10, 15-16
 OCI-NLS_MAXBUFSZ, 16-176
 OCI_NO_DATA, 2-31
 OCI_NO_Mutex, 9-3
 OCI_NUM_SHARED_PROCS, 2-25
 OCI_PIN_ANY, 13-7
 OCI_PIN_LATEST, 13-8
 OCI_PIN_RECENT, 13-8
 OCI_PTYPE_ARG
 属性, 6-17
 OCI_PTYPE_COL
 属性, 6-15
 OCI_PTYPE_COLL
 属性, 6-13
 OCI_PTYPE_DATABASE
 属性, 6-20
 OCI_PTYPE_FUNC
 属性, 6-8
 OCI_PTYPE_LIST
 属性, 6-19
 OCI_PTYPE_PKG
 属性, 6-8
 OCI_PTYPE_PROC
 属性, 6-8
 OCI_PTYPE_SCHEMA
 属性, 6-19
 OCI_PTYPE_SYN
 属性, 6-14
 OCI_PTYPE_TABLE
 属性, 6-7
 OCI_PTYPE_TYPE
 属性, 6-9
 OCI_PTYPE_TYPE_ATTR
 属性, 6-11
 OCI_PTYPE_TYPE_FUNC
 属性, 6-12
 OCI_PTYPE_TYPE_PROC
 属性, 6-12
 OCI_PTYPE_VIEW
 属性, 6-7
 OCI_SESSRLS_RETAG, 15-44, 15-45
 OCI_SHARED_MODE, 2-24
 OCI_STILL_EXECUTING, 2-31, 2-41
 OCI_STMT_SCROLLABLE_READONLY
 属性, 4-18
 OCI_SUCCESS, 2-31, 8-8
 OCI_SUCCESS_WITH_INFO, 2-31
 OCI_THREADED, 9-3
 OCI_TRANS_LOOSE, 8-5
 OCI_TRANS_READONLY, 8-3, 8-10
 OCI_TRANS_RESUME, 8-9
 OCI_TRANS_SERIALIZABLE, 8-3
 OCI_TRANS_TIGHT, 8-5
 OCI_TRANS_TWOPHASE, 8-9
 OCI_TYPECODE
 値, 3-29, 3-30, 3-31
 OCI_UTF16ID, 2-48
 OCIAnyDataAccess(), 20-12
 OCIAnyDataAttrGet(), 20-14
 OCIAnyDataAttrSet(), 20-17
 OCIAnyDataBeginCreate(), 20-20
 OCIAnyDataCollAddElem(), 20-22
 OCIAnyDataCollGetElem(), 20-24
 OCIAnyDataConvert(), 20-26
 OCIAnyDataDestroy(), 20-28
 OCIAnyDataEndCreate(), 20-29
 OCIAnyDataGetCurrAttrNum(), 20-30
 OCIAnyDataGetType(), 20-31
 OCIAnyDataIsNull(), 20-32
 OCIAnyDataSetAddInstance(), 20-35
 OCIAnyDataSetBeginCreate(), 20-37

OCIAnyDataSetDestroy(), 20-39
 OCIAnyDataSetEndCreate(), 20-40
 OCIAnyDataSetGetCount(), 20-41
 OCIAnyDataSetGetInstance(), 20-42
 OCIAnyDataSetGetType(), 20-43
 OCIAnyDataSetTypeCodeToSql(), 20-33
 OCIAQAgent
 記述子の属性, A-52
 OCIAQDeq(), 16-86
 OCIAQDeqOptions
 記述子の属性, A-44
 OCIAQEnq(), 16-88
 OCIAQEnqOptions
 記述子の属性, A-43
 OCIAQListen(), 16-100
 OCIAQMsgProperties
 記述子の属性, A-48
 OCIArray, 11-21
 バインドと定義, 11-21, 11-41
 OCIArray の操作
 コード例, 11-23
 OCIAttrGet(), 15-48
 記述用に使用, 4-13
 OCIAttrSet(), 15-50
 OCIAuthInfo の定義, 9-25
 OCIAuthInfo ハンドル属性, A-17
 OCIBindArrayOfStruct(), 15-65
 OCIBindByName(), 15-66
 OCIBindByPos(), 15-71
 OCIBindDynamic(), 15-75
 OCIBindObject(), 15-79
 OCIBreak(), 16-172
 使用, 2-38, 2-42
 OCICacheFlush(), 17-9
 OCICacheFree(), 17-50
 OCICacheRefresh(), 17-11
 OCICacheUnmark(), 17-17
 OCICacheUnpin(), 17-51
 OCICharSetConversionIsReplacementUsed(), 2-54
 OCICharSetToUnicode(), 2-54
 OCIColl, 11-21
 バインドと定義, 11-21
 OCICollAppend(), 18-6
 OCICollAssign(), 18-7
 OCICollAssignElem(), 18-9
 OCICollGetElem(), 18-11
 OCICollsLocator(), 18-14
 OCICollMax(), 18-15
 OCICollSize(), 18-16
 OCICollTrim(), 18-18
 OCIComplexObject
 使用, 10-23
 OCIComplexObjectComp
 使用, 10-23
 OCIConnectionPoolCreate(), 15-5
 OCIConnectionPoolDestroy(), 15-8
 OCIContextClearValue(), 19-20
 OCIContextGenerateKey(), 19-21
 OCIContextGetValue(), 19-19
 OCIContextSetValue(), 19-17
 OCIDate, 11-6
 バインドと定義, 11-6, 11-41
 OCIDateAddDays(), 18-30
 OCIDateAddMonths(), 18-31
 OCIDateAssign(), 18-32
 OCIDateCheck(), 18-33
 OCIDateCompare(), 18-35
 OCIDateDaysBetween(), 18-36
 OCIDateFromText(), 18-37
 OCIDateGetDate(), 18-39
 OCIDateGetTime(), 18-40
 OCIDateLastDay(), 18-41
 OCIDateNextDay(), 18-42
 OCIDateSetDate(), 18-43
 OCIDateSetTime(), 18-44
 OCIDateSysDate(), 18-45
 OCIDateTimeAssign(), 18-48
 OCIDateTimeCheck(), 18-49
 OCIDateTimeCompare(), 18-51
 OCIDateTimeConstruct(), 18-52
 OCIDateTimeConvert(), 18-54
 OCIDateTimeFromArray(), 18-55
 OCIDateTimeFromText(), 18-57
 OCIDateTimeGetDate(), 18-59
 OCIDateTimeGetTime, 18-60
 OCIDateTimeGetTime(), 18-60
 OCIDateTimeGetTimeZoneName(), 18-62
 OCIDateTimeGetTimeZoneOffset(), 18-63
 OCIDateTimeIntervalAdd(), 18-64
 OCIDateTimeIntervalSub(), 18-65
 OCIDateTimeSubtract(), 18-66
 OCIDateTimeSysTimeStamp(), 18-67
 OCIDateTimeToArray(), 18-68
 OCIDateToText(), 18-46

OCIDateZoneToZone(), 18-72
OCIDate の操作
 コード例, 11-8
OCIDefineArrayOfStruct(), 15-81
OCIDefineByPos(), 15-82
OCIDefineDynamic(), 15-86
OCIDefineObject(), 15-89
OCIDescribeAny(), 15-91
 使用方法, 6-2
 使用例, 6-23
OCIDescriptorAlloc(), 15-52
OCIDescriptorFree(), 15-54
OCIDirPathAbort(), 16-111
OCIDirPathColArrayEntryGet(), 16-112
OCIDirPathColArrayEntrySet(), 16-114
OCIDirPathColArrayReset(), 16-117
OCIDirPathColArrayRowGet(), 16-116
OCIDirPathColArrayToStream(), 16-118
OCIDirPathColArray コンテキスト, 12-5
OCIDirPathCtx コンテキスト, 12-5
OCIDirPathDataSave(), 16-120
OCIDirPathFinish(), 16-121
OCIDirPathFlushRow(), 16-122
OCIDirPathPrepare(), 16-124
OCIDirPathStreamLoad(), 16-123
OCIDirPathStreamReset(), 16-125
OCIDirPathStream コンテキスト, 12-5
OCIDuration
 使用, 13-8, 13-15
OCIDurationBegin(), 16-24, 19-10
OCIDurationEnd(), 16-25, 19-11
OCIEnvCreate(), 15-9
OCIEnvInit(), 15-12
OCIEnvNlsCreate(), 2-48, 15-14
OCIErrorGet(), 16-173
OCIExtProcAllocCallMemory(), 19-5
OCIExtProcGetEnv(), 19-8
OCIExtProcRaiseExcp(), 19-6
OCIExtProcRaiseExcpWithMsg(), 19-7
OCIExtractFromFile(), 19-29
OCIExtractFromList(), 19-36
OCIExtractFromStr(), 19-30
OCIExtractInit(), 19-23
OCIExtractReset(), 19-25
OCIExtractSetKey(), 19-27
OCIExtractSetNumKeys(), 19-26
OCIExtractTerm(), 19-24
OCIExtractToBool(), 19-32
OCIExtractToInt(), 19-31
OCIExtractToList(), 19-35
OCIExtractToOCINum(), 19-34
OCIFileClose(), 19-43
OCIFileExists(), 19-48
OCIFileInit(), 19-39
OCIFileRead(), 19-44
OCIFileSeek(), 19-46
OCIFileTerm(), 19-40
OCIFileWrite(), 19-45
OCIFormatInit(), 19-52
OCIFormatString(), 19-54
OCIFormatTerm(), 19-53
OCIHandleAlloc(), 15-56
OCIHandleFree(), 15-59
OCIInd
 使用, 10-30
OCIInitialize(), 15-18
 共有モード, 2-23
OCIIntervalAssign(), 18-75
OCIIntervalCheck(), 18-76
OCIIntervalCompare(), 18-78
OCIIntervalDivide(), 18-79
OCIIntervalFromNumber(), 18-80
OCIIntervalFromText(), 18-81
OCIIntervalFromTZ(), 18-83
OCIIntervalGetDaySecond(), 18-84
OCIIntervalGetYearMonth(), 18-86
OCIIntervalMultiply(), 18-87
OCIIntervalSetDaySecond(), 18-88
OCIIntervalSetYearMonth(), 18-90
OCIIntervalToText(), 18-93
OCIIter, 11-21
 使用例, 11-23
 バインドと定義, 11-21
OCIIterCreate(), 18-19
OCIIterDelete(), 18-20
OCIIterGetCurrent(), 18-21
OCIIterInit(), 18-22
OCIIterNext(), 18-23
OCIIterPrev(), 18-25
OCILdaToSvcCtx(), 16-175
OCILobAppend(), 16-26
OCILobAssign(), 16-28
OCILobCharSet(), 16-30, 16-31
OCILobClose(), 16-32

OCILobCopy(), 16-34
 OCILobCreateTemporary(), 16-36
 OCILobDisableBuffering(), 16-38
 OCILobEnableBuffering(), 16-39
 OCILobErase(), 16-40
 OCILobFileClose(), 16-42
 OCILobFileCloseAll(), 16-43
 OCILobFileExists(), 16-44
 OCILobFileGetName(), 16-45
 OCILobFileIsOpen(), 16-47
 OCILobFileOpen(), 16-48
 OCILobFileSetName(), 16-49
 OCILobFlushBuffer(), 16-51
 OCILobFreeTemporary(), 16-53
 OCILobGetChunkSize(), 16-54
 OCILobGetLength(), 16-56
 OCILobIsEqual(), 16-57
 OCILobIsOpen(), 16-58
 OCILobIsTemporary(), 16-60
 OCILobLoadFromFile(), 16-61
 OCILobLocatorAssign(), 16-63
 OCILobLocatorIsInit(), 16-65
 OCILobOpen(), 16-67
 OCILobRead(), 16-69
 OCILobTrim(), 16-74
 OCILobWrite(), 16-76
 OCILobWriteAppend(), 16-81
 OCILockOpt
 可能な値, 17-28, 17-57
 OCILogoff(), 15-21
 OCILogon(), 15-22
 使用方法, 2-26
 OCILogon2(), 15-24
 OCIMemoryAlloc(), 19-12
 OCIMemoryFree(), 19-15
 OCIMemoryResize(), 19-14
 OCIMessageClose(), 2-54
 OCIMessageGet(), 2-54
 OCIMessageOpen(), 2-54
 OCIMultiByteInSizeToWideChar(), 2-51
 OCIMultiByteStrCaseConversion(), 2-53
 OCIMultiByteStrcat(), 2-52
 OCIMultiByteStrcmp(), 2-52
 OCIMultiByteStrncpy(), 2-52
 OCIMultiByteStrlen(), 2-52
 OCIMultiByteStrncat(), 2-53
 OCIMultiByteStrncmp(), 2-52
 OCIMultiByteStrncpy(), 2-53
 OCIMultiByteStrnDisplayLength(), 2-53
 OCIMultiByteToWideChar(), 2-51
 OCIMultiTransPrepare(), 16-160
 OCINlsCharSetConvert(), 2-51
 OCINlsCharSetIdToName(), 2-51
 OCINlsCharSetNameToId(), 2-51
 OCINlsEnvironmentVariableGet(), 16-176
 OCINlsGetInfo(), 2-50, 2-51
 OCINlsNameMap(), 2-51
 OCINlsNumericInfoGet(), 2-51
 OCINumber, 11-14
 定義の例, 11-42
 バインドと定義, 11-14, 11-41
 バインドの例, 11-42
 OCINumberAbs(), 18-97
 OCINumberAdd(), 18-98
 OCINumberArcCos(), 18-99
 OCINumberArcSin(), 18-100
 OCINumberArcTan(), 18-101
 OCINumberArcTan2(), 18-102
 OCINumberAssign(), 18-103
 OCINumberCeil(), 18-104
 OCINumberCompare(), 18-105
 OCINumberCos(), 18-106
 OCINumberDec(), 18-107
 OCINumberDiv(), 18-108
 OCINumberExp(), 18-109
 OCINumberFloor(), 18-110
 OCINumberFromInt(), 18-111
 OCINumberFromReal(), 18-112
 OCINumberFromText(), 18-113
 OCINumberHypCos(), 18-115
 OCINumberHypSin(), 18-116
 OCINumberHypTan(), 18-117
 OCINumberInc(), 18-118
 OCINumberIntPower(), 18-119
 OCINumberIsInt(), 18-120
 OCINumberIsZero(), 18-121
 OCINumberLn(), 18-122
 OCINumberLog(), 18-123
 OCINumberMod(), 18-124
 OCINumberMul(), 18-125
 OCINumberNeg(), 18-126
 OCINumberPower(), 18-127
 OCINumberPrec(), 18-128
 OCINumberRound(), 18-129

- OCINumberSetPi(), 18-130
- OCINumberSetZero(), 18-131
- OCINumberShift(), 18-132
- OCINumberSign(), 18-133
- OCINumberSin(), 18-134
- OCINumberSqrt(), 18-135
- OCINumberSub(), 18-136
- OCINumberTan(), 18-137
- OCINumberToInt(), 18-138
- OCINumberToReal(), 18-139
- OCINumberToText(), 18-140
- OCINumberTrunc(), 18-142
- OCINumber の操作
 - コード例, 11-17
- OCIObjectArrayPin(), 17-52
- OCIObjectCopy(), 17-33
- OCIObjectExists(), 17-25
- OCIObjectFlush(), 17-13
- OCIObjectFree(), 17-54
- OCIObjectGetAttr(), 17-35
- OCIObjectGetInd(), 17-37
- OCIObjectGetObjectRef(), 17-38
- OCIObjectGetTypeRef(), 17-39
- OCIObjectIsDirty(), 17-30
- OCIObjectIsLocked(), 17-31
- OCIObjectLifetime
 - 可能な値, 17-27
- OCIObjectLock(), 17-40
- OCIObjectLockNoWait(), 17-41
- OCIObjectMarkDelete(), 17-18
- OCIObjectMarkDeleteByRef(), 17-19
- OCIObjectMarkStatus
 - 可能な値, 17-29
- OCIObjectMarkUpdate(), 17-20
- OCIObjectNew(), 17-43
- OCIObjectPin(), 17-56
- OCIObjectPinCountReset(), 17-59
- OCIObjectPinTable(), 17-60
- OCIObjectRefresh(), 17-14
- OCIObjectSetAttr(), 17-47
- OCIObjectUnmark(), 17-22
- OCIObjectUnmarkByRef(), 17-23
- OCIObjectUnpin(), 17-62
- OCIParmGet(), 15-61
 - 記述用に使用, 4-13
- OCIParmSet(), 15-63
- OCIPasswordChange(), 16-178
- OCIPinOpt
 - 使用, 13-7
- OCIRaw, 11-20
 - バインドと定義, 11-20, 11-41
- OCIRawAllocSize(), 18-144
- OCIRawAssignBytes(), 18-145
- OCIRawAssignRaw(), 18-146
- OCIRawPtr(), 18-147
- OCIRawResize(), 18-148
- OCIRawSize(), 18-149
- OCIRaw の操作
 - コード例, 11-21
- OCIRef, 11-28
 - 使用例, 11-28
 - バインドと定義, 11-28
- OCIRefAssign(), 18-151
- OCIRefClear(), 18-152
- OCIRefFromHex(), 18-153
- OCIRefHexSize(), 18-154
- OCIRefIsEqual(), 18-155
- OCIRefIsNull(), 18-156
- OCIRefToHex(), 18-157
- OCIReset(), 16-180
 - 使用, 2-42
- OCIRowid ROWID 記述子, 2-18
- OCIRowidToChar(), 16-181
- OCIServerAttach(), 15-27
 - シャドウ・プロセス, 15-28
- OCIServerDetach(), 15-29
- OCIServerDNs 記述子の属性, A-53
- OCIServerVersion(), 16-182
- OCISessionBegin(), 15-30
- OCISessionEnd(), 15-34
- OCISessionGet(), 15-35
- OCISessionPoolCreate(), 15-39
- OCISessionPoolDestroy(), 15-43
- OCISessionRelease(), 15-44
- OCISstmtExecute(), 16-5
 - iters パラメータの使用, 4-7
 - プリフェッチ, 4-7
- OCISstmtFetch(), 16-8
- OCISstmtFetch2(), 4-18, 16-10
- OCISstmtGetBindInfo(), 15-94
- OCISstmtGetPieceInfo(), 16-13
- OCISstmtPrepare()
 - SQL 文の準備, 4-4
 - 共有モード, 2-24

OCISmtPrepare2(), 16-17
 OCISmtRelease(), 16-19
 OCISmtSetPieceInfo(), 16-20
 OCISString, 11-19
 バインドと定義, 11-19, 11-41
 OCISStringAllocSize(), 18-159
 OCISStringAssign(), 18-160
 OCISStringAssignText(), 18-161
 OCISStringGetEncoding(), 18-162
 OCISStringPtr(), 18-162
 OCISStringResize(), 18-163
 OCISStringSize(), 18-164
 OCISString の操作
 コード例, 11-19
 OCISubscriptionDisable(), 16-102
 OCISubscriptionEnable(), 16-103
 OCISubscriptionPost(), 16-104
 OCISubscriptionRegister(), 16-106
 OCISubscriptionUnRegister(), 16-109
 OCISvcCtxToLda(), 16-183
 OCITable, 11-21
 バインドと定義, 11-21, 11-41
 OCITableDelete(), 18-166
 OCITableExists(), 18-167
 OCITableFirst(), 18-168
 OCITableLast(), 18-169
 OCITableNext(), 18-170
 OCITablePrev(), 18-171
 OCITableSize(), 18-172
 OCITerminate(), 15-46
 OCIThreadClose(), 16-128
 OCIThreadCreate(), 16-129
 OCIThreadHandleGet(), 16-131
 OCIThreadHndDestroy(), 16-132
 OCIThreadHndInit(), 16-133
 OCIThreadIdDestroy(), 16-134
 OCIThreadIdGet(), 16-135
 OCIThreadIdInit(), 16-136
 OCIThreadIdNull(), 16-137
 OCIThreadIdSame(), 16-138
 OCIThreadIdSet(), 16-139
 OCIThreadIdSetNull(), 16-140
 OCIThreadInit(), 16-141
 OCIThreadIsMulti(), 16-142
 OCIThreadJoin(), 16-143
 OCIThreadKeyDestroy(), 16-144
 OCIThreadKeyGet(), 16-145
 OCIThreadKeyInit(), 16-146
 OCIThreadKeySet(), 16-147
 OCIThreadMutexAcquire(), 16-148
 OCIThreadMutexDestroy(), 16-149
 OCIThreadMutexInit(), 16-150
 OCIThreadMutexRelease(), 16-151
 OCIThreadProcessInit(), 16-152
 OCIThreadTerm(), 16-153
 OCIThread パッケージ, 9-5
 OCITransCommit(), 16-155
 OCITransDetach(), 16-158
 OCITransForget(), 16-159
 OCITransMultiPrepare(), 8-9
 OCITransPrepare(), 16-161
 OCITransRollback(), 16-162
 OCITransStart(), 16-163
 OCIType
 説明, 11-29
 OCITypeAddAttr(), 20-5
 OCITypeArrayByName(), 17-65
 OCITypeArrayByRef(), 17-68
 OCITypeBeginCreate(), 20-6
 OCITypeByName(), 17-70
 OCITypeByRef(), 17-73
 OCITypeElem
 説明, 11-29
 OCITypeEndCreate(), 20-8
 OCITypeMethod
 説明, 11-29
 OCITypeSetBuiltin(), 20-9
 OCITypeSetCollection(), 20-10
 OCIUnicodeToCharset(), 2-54
 OCIUserCallbackGet(), 16-184
 OCIUserCallbackRegister(), 16-186
 OCIWideCharDisplayLength(), 2-52
 OCIWideCharInSizeToMultiByte(), 2-51
 OCIWideCharIsAlnum(), 2-53
 OCIWideCharIsAlpha(), 2-53
 OCIWideCharIsCntrl(), 2-53
 OCIWideCharIsDigit(), 2-53
 OCIWideCharIsGraph(), 2-53
 OCIWideCharIsLower(), 2-53
 OCIWideCharIsPrint(), 2-53
 OCIWideCharIsPunct(), 2-53
 OCIWideCharIsSingleByte(), 2-53
 OCIWideCharIsSpace(), 2-53
 OCIWideCharIsUpper(), 2-53

- OCIWideCharIsXdigit(), 2-53
- OCIWideCharMultibyteLength(), 2-52
- OCIWideCharStrCaseConversion(), 2-52
- OCIWideCharStrcat(), 2-52
- OCIWideCharStrchr(), 2-52
- OCIWideCharStrcmp(), 2-51
- OCIWideCharStrcpy(), 2-52
- OCIWideCharStrlen(), 2-52
- OCIWideCharStrncat(), 2-52
- OCIWideCharStrncmp(), 2-52
- OCIWideCharStrncpy(), 2-52
- OCIWideCharStrrchr(), 2-52
- OCIWideCharToLower(), 2-51
- OCIWideCharToMultiByte(), 2-51
- OCIWideCharToUpper(), 2-51
- OCI アプリケーション
 - OTT の使用, 14-21
 - 一般的構造, 2-3
 - オブジェクト付き
 - 初期化, 10-9
 - オブジェクトを使用した構造体, 10-3
 - 構造, 2-3
 - コンパイル, 1-4
 - 終了, 2-30
 - 初期化例, 2-27
 - ステップ, 2-20
 - リンク, 1-4
- OCI 環境
 - オブジェクトの初期化, 10-9
- OCI 関数
 - アドバンスト・キューイング, xxxvii
 - グローバリゼーション, 2-2
 - グローバリゼーション・サポート, xxxvii
 - コールの取消し, 2-38
 - 使用されなくなった, 1-18
 - その他のマニュアル, xxxvii
 - データ・カートリッジ, xxxvii, 2-2
 - 未サポート, 1-20
 - リターン・コード, 2-31, 2-34
- OCI コールの取消し, 2-38
- OCI での UTF-16 Unicode サポート, 2-45, 2-49, 2-51
- OCI ドキュメント、その他, xxxvi
- OCI ナビゲーション関数, 13-20
 - 確保 / 確保解除 / 解放関数, 13-21
 - その他の関数, 13-23
 - ネーミング計画, 13-20
 - フラッシュ関数, 13-21
 - マーク関数, 13-22
 - メタ属性アクセッサ関数, 13-22
- OCI プログラム, 「OCI アプリケーション」を参照
- OCI プロセス
 - オブジェクトの初期化, 10-9
- OCI リレーショナル関数
 - アドバンスト・キューイングおよびパブリッシュ・サブスクライブ, 16-85
 - 参照項目の概要, 19-2
 - 接続、認証および初期化, 15-4
- OID, 「オブジェクト識別子」を参照
- Oracle Call Interface, 「OCI」を参照
- Oracle データ型, 3-2
 - C へのマッピング, 11-3
- oratypes.h
 - OCI にパラメータを渡す手段としてサポートされている, 3-33
 - 内容, 3-33
- ORE, 「オブジェクト・ランタイム環境」を参照
- OTT
 - intype ファイル, 14-33
 - intype ファイルの提供, 14-8
 - outtype ファイル, 14-20
 - 概要, 14-2
 - コマンドライン, 14-6
 - コマンドライン構文, 14-26
 - 使用方法, 14-1, 14-2
 - 制限, 14-40
 - データ型マッピング, 14-10
 - データベースでの型の作成, 14-5
 - パラメータ, 14-27
 - リファレンス, 14-25
- OTT, 「Object Type Translator」を参照
- OTT 初期化関数
 - コール, 14-23
 - 作業, 14-25
- OTT パラメータ
 - CASE, 14-30
 - CODE, 14-29
 - CONFIG, 14-30
 - ERRTYPE, 14-30
 - HFILE, 14-30
 - INITFILE, 14-29
 - INITFUNC, 14-29
 - INTYPE, 14-28
 - OUTTYPE, 14-28
 - SCHEMA_NAMES, 14-31

USERID, 14-28
表示される場所, 14-32
OTT パラメータ TRANSITIVE, 14-31
OTT パラメータ URL, 14-32
OUTTYPE OTT パラメータ, 14-28
outtype ファイル, 14-33
OTT の実行時, 14-20

P

PDO, 「パラメータ記述子オブジェクト」を参照
PL/SQL, 1-10
NESTED TABLE のバインドと定義, 5-43
OCI アプリケーションでの使用, 2-44
OCI プログラムでの使用, 5-7
REF カーソルのバインドと定義, 5-43
出力変数の定義, 5-24
ピース単位操作, 5-50
プレースホルダのバインド, 2-44

R

RAW
外部データ型, 3-15
REF
インジケータ変数, 2-36, 2-37
カーソル変数、バインド, 5-17
外部データ型, 3-18
サーバーからの取出し, 10-10
定義, 5-22, 11-39
バインド, 5-10, 11-37
REF カーソル変数
バインドと定義, 5-43
REF 列
ダイレクト・パス・ロード, 12-23
RETURNING 句
OCI の使用, 5-29
REF で, 5-31
エラー処理, 5-31
バインド, 5-30
変数の初期化, 5-30
RETURNING 句を使用する DML
「RETURNING 句」を参照
ROWID
位置指定の更新および削除に使用, 2-39
外部データ型, 3-19

ユニバーサル ROWID, 3-6
論理, 3-6
ROWID 記述子, 2-18

S

sb1
定義, 3-33
sb2
定義, 3-33
sb4
定義, 3-33
SCHEMA_NAMES OTT パラメータ, 14-31
使用方法, 14-37
SQLCS_IMPLICIT, 5-34, 16-30, 16-72, 16-79, 16-83
SQLCS_NCHAR, 5-34, 16-30, 16-72, 16-79, 16-83
SQLT_NTY
オブジェクト・メモリーの事前割当て, 11-40
説明, 3-18
定義の例, 11-48
バインドの例, 11-47
SQLT_REF
説明, 3-18
定義, 3-18
SQLT 型コード, 3-31
SQL 問合せ
結果のフェッチ, 4-16
出力変数の定義, 4-15, 5-18, 11-38
出力変数の定義, 「定義操作」を参照
プレースホルダのバインド, 「バインド操作」を参照
参照
文の種類, 1-9
SQL 文, 1-7
実行, 4-7
実行の準備, 4-4
種類
PL/SQL, 1-10
埋込み SQL, 1-11
制御文, 1-8
データ操作言語, 1-8
データ定義言語, 1-7
問合せ, 1-9
準備するタイプの決定, 4-4
処理, 4-2
バインド・プレースホルダ, 4-6, 5-2, 11-36
SQL 文の実行, 4-7

STRING

外部データ型, 3-12

sword

定義, 3-33

T

TDO

型記述子オブジェクト, 「TDO」を参照

取得, 11-29

説明, 11-29

定義, 11-36

TDO, 「型記述子オブジェクト」を参照

TIMESTAMP WITH LOCAL TIME ZONE データ型,
3-23

TIMESTAMP WITH TIME ZONE データ型, 3-22

TIMESTAMP データ型, 3-22

TRANSITIVE OTT パラメータ, 14-9, 14-15, 14-31

U

ub1

定義, 3-33

ub2

定義, 3-33

ub4

定義, 3-33

Unicode

OCILobRead(), 16-73

OCILobWrite(), 16-80

キャラクタ・セット ID, A-35, A-38

UNIX での OCI アプリケーションの作成, B-1

UNSIGNED

外部データ型, 3-15

URL OTT パラメータ, 14-32

UROWID

ユニバーサル ROWID, 3-6

USERID OTT パラメータ, 14-28

utext

Unicode データ型, 5-41

UTF-16 データ、コード例, 5-41

V

VARCHAR

外部データ型, 3-13

VARCHAR2

外部データ型, 3-9

VARNUM

外部データ型, 3-13

VARRAW

外部データ型, 3-15

W

with_context

外部プロシージャ関数に対する引数, 19-3

X

XA 仕様, 8-4

XID, 「トランザクション識別子」を参照

xtramem_sz パラメータ

使用方法, 2-19

あ

値, 10-5

オブジェクト・アプリケーション, 10-6

アップグレード

リリース 7.x からリリース 8.0, 1-21

リリース 7.x からリリース 8.0 OCI, 1-22

アドバンスト・キューイング

OCI および, 9-48

OCI 関数, 9-49

OCI と PL/SQL, 9-49

OCI の説明, 9-49

エンキュー関数, 16-88

関数, 16-85

説明, 9-48

デキュー関数, 16-86

例, 16-89

アトミック NULL, 10-30

アプリケーション・フェイルオーバー

OCI コールバック, 9-41

コールバックの登録, 9-43

コールバックの例, 9-43

い

- 移行
 - セッション, 8-11, 15-31
 - リリース 7.x からリリース 8.0, 1-21
- 一時オブジェクト, 10-6
 - LOB
 - 属性, 7-4
 - メタ属性, 10-20
- 位置指定, 2-39
 - 削除, 2-39
- 一貫性
 - オブジェクト・キャッシュ, 13-5
- インジケータ変数, 2-36
 - PL/SQL OUT バインド, 11-39
 - REF 定義, 11-39
 - REF バインド, 11-37
 - REF 用, 2-36, 2-37
 - 構造体配列, 5-28
 - 名前付きデータ型の定義, 11-39
 - 名前付きデータ型のバインド, 11-37
 - 名前付きデータ型用, 2-36, 2-37

う

- 埋込み SQL, 1-11
 - OCI コールとの混在, 1-11
- 埋込みオブジェクト
 - フェッチ, 10-15

え

- 永続オブジェクト, 10-5
 - メタ属性, 10-17
- エラー
 - オブジェクト・アプリケーションでの処理, 10-36
 - 処理, 2-31
 - 処理例, 2-32
- エラー・コード
 - 定義コール, 2-33
 - ナビゲーション関数, 17-5
- エラー・ハンドル
 - 説明, 2-9
 - 属性, A-10

お

- オブジェクト, 12-16, 13-6
 - 2 次メモリー, 13-17
 - LOB 属性, 7-3
 - NCHAR および NVARCHAR2 属性, 11-3
 - NULL, 10-30
 - OCI オブジェクト・アプリケーション構造体, 10-3
 - OCI でのアクセス, 14-23
 - OCI での使用方法, 10-2
 - OCI での操作, 14-23
 - 一時, 10-5, 10-6
 - 一時オブジェクトの LOB 属性, 7-4
 - インスタンスのメモリー・レイアウト, 13-17
 - 永続, 10-5
 - 解放, 10-32, 13-9
 - 確保, 10-11, 13-7
 - 確保解除, 10-29, 13-9
 - 確保カウント, 10-29
 - 確保継続時間, 13-15
 - 型, 17-2
 - クライアント側キャッシュ, 13-2
 - 継続時間, 13-15
 - コピー, 10-32
 - 作成, 10-32
 - 種類, 10-5
 - 属性, 10-16
 - 操作, 10-13
 - 存続期間, 17-2
 - トップレベル・メモリー, 13-17
 - ナビゲーション, 13-18
 - 単純, 13-18
 - 配列確保, 10-12
 - フラッシュ, 13-11
 - 変更のフラッシュ, 10-14
 - マーク, 10-14, 13-10
 - マーク解除, 13-10
 - メタ属性, 10-16
 - メモリー管理, 13-2
 - 用語, 10-5, 17-2
 - リフレッシュ, 13-12
 - ロック, 13-13
 - 割当て時間, 13-15
- オブジェクト・アプリケーション
 - コミット, 13-15
 - データベース接続, 10-10
 - ロールバック, 13-15

- オブジェクト型
 - C アプリケーションでの表現, 10-8
- オブジェクト関数
 - サーバー・ラウンドトリップ回数, C-5
 - 「ナビゲーション関数」を参照
- オブジェクト・キャッシュ, 13-2
 - 一貫性, 13-5
 - オブジェクトの削除, 13-7
 - オブジェクトのロード, 13-7
 - コヒーレンス, 13-5
 - サイズの設定, 13-5
 - 初期化, 10-9
 - 操作, 13-6
 - メモリー・パラメータ, 13-5
- オブジェクト参照, 10-35
- オブジェクト参照, 「REF」を参照
- オブジェクト識別子
 - 永続オブジェクト用, 10-5
- オブジェクトの確保, 13-7
- オブジェクト表
 - ダイレクト・パス・ロード, 12-29
- オブジェクト・ランタイム環境
 - 初期化, 10-9
- UNSigned, 3-15
- VARCHAR, 3-13
- VARCHAR2, 3-9
- VARNUM, 3-13
- VARRAW, 3-15
 - 名前付きデータ型, 3-18
 - 変換, 3-24
- 外部で初期化されたコンテキスト, 8-20
- 外部プロシージャ
 - OCI コールバック, 9-41
- 外部プロシージャ関数
 - with_context 型, 19-3
 - リターン・コード, 19-3
- 解放
 - オブジェクト, 10-32, 13-9
- 拡張 DML 配列, 4-9
- 拡張 DML 配列機能, 4-9
- 拡張子
 - OTT のデフォルトのファイル名, 14-39
- 確保, 13-7
- 確保解除, 10-29, 13-9
 - オブジェクト, 13-9
- 確保カウント, 10-29
- 確保継続時間
 - オブジェクト, 13-15
 - 例, 13-16
- 型
 - 記述, 6-2
 - 属性, 6-9
- 型関数
 - 属性, 6-12
- 型記述子オブジェクト, 10-8, 11-29
- 型コード, 3-29
- 型参照, 10-35
- 型属性
 - 属性, 6-11
- 型の継承
 - OTT のサポート, 14-16
- 型の変更, 10-41
 - オブジェクト・キャッシュ, 13-23
- 型プロシージャ
 - 属性, 6-12
- 環境ハンドル
 - 説明, 2-9
 - 属性, A-3
- 関数名
 - コーディング・ガイドライン, 2-41

か

- カートリッジ関数, 19-1
- 外部データ型, 3-4, 3-7
 - CHAR, 3-16
 - CHARZ, 3-17
 - DATE, 3-14
 - FLOAT, 3-11
 - INTEGER, 3-11
 - LOB, 3-19
 - LONG, 3-13
 - LONG RAW, 3-15
 - LONG VARCHAR, 3-16
 - LONG VARRAW, 3-16
 - NUMBER, 3-10
 - RAW, 3-15
 - REF, 3-18
 - ROWID, 3-19
 - SQLT_BLOB, 3-19
 - SQLT_CLOB, 3-19
 - SQLT_NCLOB, 3-19
 - SQLT_NTY, 3-18
 - SQLT_REF, 3-18
 - STRING, 3-12

き

キーワード, xxxviii, 2-39

記述

暗黙的, 4-13

型, 6-2

コレクション, 6-2

シノニム, 6-2

順序, 6-2

スキーマ, 6-2

ストアド・ファンクション, 6-2

ストアド・プロシージャ, 6-2

選択リスト, 4-12

データベース, 6-2

パッケージ, 6-2

ビュー, 6-2

表, 6-2

明示的, 4-14

明示のおよび暗黙的, 6-5

記述関数, 15-64

記述子, 2-15

ROWID, 2-18

オブジェクト, 11-29

スナップショット, 2-16

パラメータ, 2-18

複合オブジェクト検索, 2-19

割当て, 2-25

記述子オブジェクト, 11-29

記述子関数, 15-47

記述操作

サーバー・ラウンドトリップ回数, C-6

記述ハンドル

説明, 2-11

属性, A-40

キャッシュ関数

サーバー・ラウンドトリップ回数, C-5

キャラクタ・セット ID, 5-34

Unicode, A-35, A-38

キャラクタ・セット・フォーム, 5-34

共有データ構造モード, 2-22

共有モード, 2-22

OCIInitialize(), 2-23

OCIStmtPrepare(), 2-24

環境変数の使用方法, 2-24

く

グローバルゼーション・サポート, xxxvii, 2-45

OCI 関数, xxxvii, 2-2

グローバル・トランザクション, 8-4

け

継続時間

オブジェクト, 13-15

例, 13-16

結果セット, 4-17

こ

更新, 2-39

位置指定, 2-39

ピース単位, 5-44, 5-47

構造体

配列, 5-25

構造体配列, 5-25

インジケータ変数, 5-28

使用される OCI コール, 5-28

スキップ・パラメータ, 5-26

コーディング・ガイドライン

関数名, 2-41

予約語, 2-39

コード

サンプル・プログラム, B-1

デモ・プログラムのリスト, B-1

コールバック

LOB ストリーム・インタフェース, 7-14

LOB 操作用, 7-13

LOB の書込み用, 7-16

LOB の読込み用, 7-14

アプリケーション・フェイルオーバー, 9-41

アプリケーション・フェイルオーバーのための

登録, 9-43

外部プロシージャから, 9-41

制限, 9-38

動的な登録, 9-35

パラメータ・モード, 15-87

ユーザー定義関数, 9-30

コピー

オブジェクト, 10-32

コヒーレンス

オブジェクト・キャッシュ, 13-5

- コミット, 2-29
 - オブジェクト・アプリケーション, 13-15
 - グローバル・トランザクションの1フェーズ, 8-8
 - グローバル・トランザクションの2フェーズ, 8-8
- コミット時ロック
 - 実現, 13-14
- コレクション
 - 記述, 6-2
 - スキャン関数, 11-23
 - 説明, 11-21
 - 操作用関数, 11-22
 - 属性, 6-13
 - データ操作関数, 11-22
 - マルチレベル, 11-26

さ

- サーバー・ハンドル
 - サービス・コンテキストでの設定, 2-10
 - 説明, 2-10
 - 属性, A-13
- サーバー・ラウンドトリップ回数
 - LOB 関数, C-3
 - オブジェクト関数, C-5
 - 記述操作, C-6
 - キャッシュ関数, C-5
 - データ型マッピング関数および操作関数, C-7
 - リレーショナル関数, C-8
- サービス・コンテキスト・ハンドル
 - 説明, 2-9
 - 属性, A-11
 - 要素, 2-9
- 削除
 - 位置指定, 2-39
- 作成
 - オブジェクト, 10-32
- サブスクリプション・ハンドル, 2-12
 - 属性, A-54
- 参照, 「REF」を参照
- サンプル・プログラム, B-1

し

- 実行
 - 複数サーバーに対する, 4-5
 - モード, 4-8

- シノニム
 - 記述, 6-2
 - 属性, 6-14
- 順序
 - 記述, 6-2
 - 属性, 6-15
- 初期化関数, 15-4

す

- スキーマ
 - 記述, 6-2
 - 属性, 6-19
- スキップ・パラメータ
 - 構造体配列, 5-26
 - 標準配列, 5-27
- スクロール・カーソル, 4-17
 - 例, 4-19
- ストアド・ファンクション
 - 記述, 6-2
- ストアド・プロシージャ
 - 記述, 6-2
- スナップショット
 - 実行, 4-7
- スナップショット記述子, 2-16
- スナップショットの実行, 4-7
- スレッド管理関数, 16-126
- スレッド・セーフティ, 9-2
 - 3層アーキテクチャ, 9-2
 - OCIでの実装, 9-3
 - 基本概念, 9-3
 - 必要なOCIコール, 9-3
 - 利点, 9-2
 - リリース7.xとリリース8.0コールの混在, 9-4
- スレッド・パッケージ, 9-5

せ

- セッション
 - 移行, 8-11, 15-31
- セッション管理, 8-11, 8-13
- セッション・プーリング, 9-23
 - タグ付け, 9-23
- セッション・プーリング、機能, 9-23
- セッション・プーリングの例, 9-28
- セッション・プール・ハンドル
 - 属性, A-23

接続関数, 15-4
接続プーリング, 9-13
 コード例, 9-19
接続モード
 非ブロック化, 2-41
選択リスト
 記述, 4-12

そ

操作の定義, 4-15, 5-18, 11-38
 LOB, 5-22
 PL/SQL 出力変数, 5-24
 REF, 5-22, 11-39
 使用するステップ, 5-19
 名前付きデータ型, 5-21, 11-38
 ピース単位フェッチ, 5-24
 例, 5-20
挿入
 ピース単位, 5-44, 5-47
属性
 オブジェクト, 10-16
 パラメータ, 6-5
 パラメータ記述子, 6-5
 ハンドル, 2-13
属性記述子オブジェクト, 11-29
その他の関数, 16-171

た

ダイレクト・パス
 日付列, 12-14
ダイレクト・パス関数コンテキスト, 12-5
ダイレクト・パス・ハンドル, 2-12
ダイレクト・パス・ロード, 12-2
 関数, 12-8, 16-110
 コンテキスト・ハンドル属性, A-58
 ストリーム・ハンドル属性, A-67
 制限, 12-9
 ダイレクト・パス・コンテキスト・ハンドル, 12-5
 ダイレクト・パス・ストリーム・ハンドル, 12-7
 ダイレクト・パス列配列ハンドル, 12-6
 ハンドル, 12-5
 ハンドル属性, A-58
 ピース単位, 12-31
 例, 12-9
 列のデータ型, 12-4, A-72

 列配列ハンドル属性, A-66
 列パラメータ属性, A-68
タグ付け
 セッション・プーリング, 9-23, 15-38, 15-44
単一のメッセージで複数のブランチを準備, 8-9

て

定義
 OCINumber, 11-42
 配列, 11-41
 リターン・コードとエラー・コード, 2-33
定義関数, 15-64
定義ハンドル
 説明, 2-11
 属性, A-37
データ・カートリッジ
 OCI 関数, xxxvii, 2-2, 19-1
データ型
 ANSI DATE, 3-22
 BFILE, 3-20
 BLOB (バイナリ・ラージ・オブジェクト), 3-21
 CLOB, 3-21
 FILE, 3-20
 INTERVAL DAY TO SECOND, 3-23
 INTERVAL YEAR TO MONTH, 3-23
 NCLOB, 3-21
 OCI での操作, 11-4
 Oracle, 3-2
 Oracle から C へのマッピング, 11-3
 TIMESTAMP, 3-22
 TIMESTAMP WITH LOCAL TIME ZONE, 3-23
 TIMESTAMP WITH TIME ZONE, 3-22
 外部, 3-4, 3-7
 ダイレクト・パス・ロード, 12-4, A-72
 内部, 3-4
 内部コード, 3-4
 バインドと定義, 11-41
 ピース単位操作作用, 5-44
 変換, 3-24
 マッピング、Oracle 方法論, 11-4
 マッピング、OTT, 14-10
 マッピング関数および操作関数, C-7
データ型の内部コード
 データ型コード, 3-4
データ構造
 リリース 8.0 用の新規, 2-5

- データ操作言語
 - SQL 文, 1-8
- データ定義言語
 - SQL 文, 1-7
- データベース
 - 記述, 6-2
 - 属性, 6-20
- データベース接続
 - オブジェクト・アプリケーション用, 10-10
- デフォルトのファイル名拡張子
 - OTT, 14-39
- デフォルト名のマッピング
 - OTT, 14-39
- デモ・プログラム, B-1
 - リスト, B-1

と

- 問合せ
 - 明示的記述, 4-14
- 問合せ, 「SQL 問合せ」を参照
- 登録
 - ユーザー・コールバック, 9-31
- ドキュメント、その他の Oracle, xxxii
- トップレベル・メモリー
 - オブジェクト, 13-17
- トランザクション, A-26
 - OCI 関数
 - トランザクション, 8-2
 - 関数, 16-154
 - グローバル, 8-4
 - 1 フェーズ・コミット, 8-8
 - 2 フェーズ・コミット, 8-8
 - トランザクション識別子, 8-4
 - ブランチ, 8-5
 - ブランチの状態, 8-7
 - グローバルな例, 8-9
 - コミット, 2-29
 - 初期化パラメータ, 8-10
 - シリアル化可能, 8-3
 - 読取り専用, 8-3
 - ローカル, 8-3
 - ロールバック, 2-29
- トランザクション識別子, 8-4
- トランザクションの複雑度
 - OCI でのレベル, 8-3

- トランザクション・ハンドル
 - 説明, 2-10
 - 属性, A-26

な

- 内部データ型, 3-4
 - 変換, 3-24
- ナビゲーション関数
 - エラー・コード, 17-5
 - 戻り値, 17-5
 - 用語, 17-3
- ナビゲーション, 13-18
- 名前付き, 11-36
- 名前付きデータ型
 - インジケータ変数, 2-36, 2-37
 - 外部データ型, 3-18
 - 定義, 3-18, 5-21, 11-38
 - バインド, 5-10, 11-36
 - バインドと定義, 11-41

に

- 日時
 - 予期しない結果の回避, 3-24
- 日時および日付
 - アップグレード規則, 3-29
- 認証
 - X.509 証明書による, 8-16
 - 管理, 8-11
 - 識別名による, 8-15
- 認証関数, 15-4
- 認証情報ハンドル属性, A-17

は

- パーティション化されたファイングレイン・アクセス・コントロール, 8-16
- 配列
 - スキップ・パラメータ, 5-27
 - 定義, 11-41
 - バインド, 11-37
- バインド
 - OCINumber, 11-42
 - PL/SQL プレースホルダ, 2-44
 - 概要, 5-17

- 配列, 11-37
 - バッファの拡張, 5-37
- バインド関数, 15-64
- バインド時のバッファの拡張, 5-37
- バインド操作, 4-6, 5-2, 11-36
 - LOB, 5-10
 - OCI_DATA_AT_EXEC モード, 5-16
 - OCI 配列インタフェース, 5-5
 - PL/SQL, 5-5
 - REF, 5-10, 11-37
 - REF カーソル変数, 5-17
 - 行われた関連付け, 5-3
 - 使用するステップ, 5-6
 - 定位置対名前付き, 5-4
 - 名前付き対定位置, 5-4
 - 名前付きデータ型, 5-10, 11-36
 - 変数の初期化, 5-3
 - 例, 5-6
- バインド・ハンドル
 - 説明, 2-11
 - 属性, A-34
- パスワード管理, 8-11, 8-12
- パッケージ
 - 記述, 6-2
 - 属性, 6-8
- バッチ・エラー・モード, 4-9
- パブリッシュ・サブスクライブ
 - _SYSTEM_TRIG_ENABLED パラメータ, 9-60
 - COMPATIBLE パラメータ, 9-54
 - LDAP 登録, 9-56
 - 関数, 9-53
 - サブスクリプション・ハンドル, 9-54
 - 通知機能, 9-52
 - 通知コールバック, 9-58
 - ハンドル属性, 9-54, A-54
 - 例, 9-60
- パブリッシュ・サブスクライブ関数, 16-85
- パブリッシュ・サブスクライブ通知の LDAP 登録, 9-56
- パラメータ
 - 属性, 6-5
 - バッファ長, 15-3, 16-3
 - モード, 15-2, 16-2, 19-2
 - 文字列長, 15-3, 16-3
 - 文字列を渡す, 2-35
- パラメータ記述子, 2-18
 - 属性, 6-5, A-40

- パラメータ記述子オブジェクト, 11-29
- ハンドル, 2-5
 - C データ型, 2-5
 - エラー・ハンドル, 2-9
 - 親が解放される際に解放される子, 2-7
 - 解放, 2-7
 - 環境ハンドル, 2-9
 - 記述ハンドル, 2-11
 - サーバー・ハンドル, 2-10
 - サービス・コンテキスト・ハンドル, 2-9
 - サブスクリプション, 2-12, 9-54
 - タイプ, 2-5
 - ダイレクト・パス, 2-12
 - 定義ハンドル, 2-11
 - トランザクション・ハンドル, 2-10
 - バインド・ハンドル, 2-11
 - プロセス, 2-13
 - プロセスの属性, A-75
 - 文ハンドル, 2-11
 - ユーザー・セッション・ハンドル, 2-10
 - 利点, 2-8
 - 割当て, 2-7, 2-25
- ハンドル関数, 15-47
- ハンドル属性, 2-13
 - 設定, 2-13
 - 読み込み, 2-13

ひ

- ピース単位操作, 5-47
 - PL/SQL, 5-50
 - 更新, 5-44
 - 挿入, 5-44
 - フェッチ, 5-44, 5-50
 - 有効なデータ型, 5-44
- ピース単位フェッチ, 5-50
- 引数の属性, 6-17
- 日付キャッシュ, 12-14
- 非ブロック化モード, 2-41
 - 例, 2-42
- ビュー
 - 記述, 6-2
 - 属性, 6-7
- 表
 - 記述, 6-2
 - 属性, 6-7

ふ

- ファイングレイン・アクセス・コントロール
 - パーティション化, 8-16
- ファンクション
 - 属性, 6-8
- フェッチ
 - ピース単位, 5-44, 5-50
- フェッチ操作, 4-16
 - LOB データ, 4-16
 - プリフェッチ・カウンタの設定, 4-17
- 複合オブジェクト検索, 10-20
 - 実現, 10-23
 - ナビゲーション・プリフェッチ, 10-25
- 複合オブジェクト検索 (COR) 記述子, 2-19
 - 属性, A-42
- 複合オブジェクト検索 (COR) ハンドル, 2-12
 - 属性, A-41
- 複数サーバー
 - 文の実行, 4-5
- 不透明、定義, 1-2
- フラッシュ, 13-11
 - オブジェクト, 13-11
 - オブジェクトの変更, 10-14
- ブランチ
 - 再開, 8-7
 - 連結解除, 8-7
- ブランチの再開, 8-7
- ブランチの連結解除, 8-7
- プリフェッチ
 - OCIStmtExecute() 中に, 4-7
 - プリフェッチ・メモリー・サイズの設定, 4-17
 - 行カウンタの設定, 4-17
- プロキシ認証, 8-20
- プロシージャ
 - 属性, 6-8
- プロセス
 - ハンドル属性, A-75
- プロセス・ハンドル, 2-13
- ブロック化モード, 2-41
- 文キャッシュ, 9-28
 - コード例, 9-30
- 文ハンドル
 - 説明, 2-11
 - 属性, A-27

ま

- マーク
 - オブジェクト, 13-10
- マーク解除, 13-10
 - オブジェクト, 13-10
- マルチスレッド開発
 - 基本概念, 9-3

め

- メソッド記述子オブジェクト, 11-29
- メタ属性
 - 一時オブジェクト, 10-20
 - 永続オブジェクト, 10-17
 - オブジェクト, 10-16

も

- 文字長セマンティクス, 2-45, 5-38, 5-39, 6-21
- 文字列
 - パラメータとして渡す, 2-35
- 戻り値
 - ナビゲーション関数, 17-5

ゆ

- ユーザー・セッション・ハンドル
 - サービス・コンテキストでの設定, 2-10
 - 説明, 2-10
 - 属性, A-17
- ユーザー定義コールバック関数, 9-30
 - 登録, 9-31
- ユーザー・メモリー
 - 割当て, 2-19
- ユニバーサル ROWID, 3-6

よ

- 用語
 - このマニュアルで使用される, 1-11
 - ナビゲーション関数, 17-3
- 予約語, xxxviii, 2-39
- 予約済みネームスペース, 2-40

ら

ラウンドトリップ回数,
「サーバー・ラウンドトリップ回数」を参照

り

リスト
属性, 6-19
利点
OCI, 1-3
リフレッシュ, 13-12
オブジェクト, 13-12
リレーショナル関数, C-8
サーバー・ラウンドトリップ回数, C-2

れ

例
OCIThread の使用方法, 9-11
デモ・プログラム, B-1
非ブロック化モード, 2-42
列
属性, 6-5, 6-15
列オブジェクト
ダイレクト・パス・ロード, 12-17

ろ

ロールバック, 2-29
オブジェクト・アプリケーション, 13-15
ロケータ, 2-15
LOB データ型用, 2-17
ロック, 13-13
オブジェクト, 13-13
最適モデル, 13-14

わ

割当て時間
オブジェクト, 13-15
例, 13-16

