多言語データベース/アプリケーションを目的としたUnicodeデータ型への移行

オラクル・ホワイト・ペーパー
2003年8月

ORACLE

多言語データベース/アプリケーションを目的とした Unicode データ型への移行

Oracle Corporation **発行**「Migration to Unicode Datatypes for Multilingual Databases and Applications」**の翻訳版です。**

多言語データベース/アプリケーションを目的とした Unicode データ型への移行

概要

Unicode データ型は Oracle9*i* で採用されました。Unicode データ型は、SQL NCHAR データ型（NCHAR、NVARCHAR2、および NCLOB）を介してサポートされています。（このホワイト・ペーパーでは、「Unicode データ型」は SQL NCHAR 型を意味します。）SQL NCHAR データ型は Oracle8 から存在しています。ただし Oracle9*i* 以降、顧客のグローバリゼーション要件を満たすために SQL NCHAR データ型が再定義され、長さセマンティクスを変更しています。SQL NCHAR データ型の列に保存されているデータは、データベース・キャラクタ・セットに関係なく排他的な Unicode エンコーディングで保存します。これらの Unicode 列を使用すれば、ユーザーはデータベース・キャラクタ・セットとして Unicode を使用しないデータベースに Unicode を保存できます。したがって開発者は、データベース・キャラクタ・セットに依存せず Unicode アプリケーションを構築できます。顧客にとっても Unicode データ型は、Unicode サポートに既存のアプリケーションとデータベースの段階的な移行が容易になります。

このホワイト・ペーパーは次に示す 3 つの部分から構成されます。最初には、Unicode データ型の特性を簡単に紹介します。次に、Unicode データ型の活用に既存データベースを移行する手順、機能および性能に関する事項を説明します。3 番目には、Unicode データ型サポートのためのアプリケーションの移行手順およびデータベース・スキーマの変更を処理する手順を強調します。最後に簡単なまとめがあります。

Unicode データ型の特性

Oracle9*i* に採用された Unicode データ型のコンセプトにより、顧客は Unicode 以外のデータベース内で Unicode 列をサポートできます。これは非常に強力な機能であり、SQL NCHAR データ型とその他のデータ型の間のインターオペラビィリティ（相互運用性）によりさらに強化されます。ユーザーは SQL CHAR データと同様に SQL NCHAR データを保存、処理、および検索できます。

新規 Unicode データ型機能に関しては、このセクションで説明するいくつかの主要なアスペクトがあります。

- キャラクタ・セットのエンコーディング
- 文字長セマンティクス
- 相互運用性
- データ消失の処理
- Unicode 文字列処理

Unicode キャラクタ・セットのエンコーディング

Oracle9*i* および Oracle Database 10g は、Unicode データ型に関する 2 つの Unicode エンコーディングをサポートしています。Oracle キャラクタ・セット名は、AL16UTF16 および UTF8 です。データベースを最初に作成するとき、「各国語キャラクタ・セット」パラメータとして AL16UTF16 または UTF8 を指定できます。各国語キャラクタ・セットが指定されない場合、デフォルトは AL16UTF16 です。Unicode エンコーディング・サポートの詳細については、Globalization Support Guide を参照してください。

Supplementary Character（補充文字）は、U+10000～U+10FFFF の範囲のコードポイントを持つ Unicode エンコード文字です。UTF-16 エンコーディングの場合は、2 つの Unicode 値シーケンスからなるサロゲート・ペアでエンコードされます。最初の値は U+D800～U+DBFF の範囲の高いサロゲートで 2 つ目の値は U+DC00～U+DFFF の範囲の低いサロゲートです。

NCHAR キャラクタ・セットとして AL16UTF16 を持つデータベースの作成例を次に示します。

CREATE DATABASE mydb
MAXINSTANCES 1
MAXLOGHISTORY 1
MAXLOGFILES 5
MAXLOGMEMBERS 5
MAXDATAFILES 100
DATAFILE '/vobs/oracle/oradata/mynewdb/system01.dbf' SIZE 325M REUSE
CHARACTER SET al32utf8
NATIONAL CHARACTER SET AL16UTF16
LOGFILE GROUP 1 ('/vobs/oracle/oradata/mynewdb/redo01.log') SIZE 100M,
GROUP 2 ('/vobs/oracle/oradata/mynewdb/redo02.log') SIZE 100M;

文字長セマンティクス

文字長セマンティクスも Oracle9*i* で採用されました。これは、バイト数ではなく Unicode のコード単位数に基づいた文字の長さを測ります。バイト数に基づいて測定するときは、バイト長セマンティクスと呼ばれます。Unicode のコード単位は、その他のキャラクタ・セット内での UCS2 コードポイントまたはそれに相当する表現に使用される 16 ビット単位です。それは、エンコードされたテキストの単位を表現できる最小ビット組合せです。Java 文字列の場合の長さセマンティクスに相当します。バイト・セマンティックと比較した場合の文字長セマンティクスの利点としては、長さが視覚的な長さに近いことおよび異なるキャラクタ・セット間での移植性です。データベースのデフォルトの長さセマンティクスは BYTE ですが、SQL NCHAR 型の場合は文字長セマンティクスが正当な長さセマンティクスです。SQL NCHAR 列定義には'CHAR'または'BYTE'数量詞は使用できません。

次に例を示します。

```
CREATE TABLE emp (
    empno NUMBER(4),
    ename NVARCHAR2(10),
    job NVARCHAR2(9),
    mgr NUMBER(4),
```

hiredate DATE,
SAL NUMBER(7,2),
deptno NUMBER(2));

ename 列には最大 10 の Unicode コード単位を収容できます。NCHAR キャラクタ・セットが AL16UTF16 のとき、最大バイト長は 20 バイトです。それ以外のとき、NCHAR キャラクタ・セットが UTF8 なら、*ename* 列にはまた最大 10 の Unicode コード単位を収容できますが、最大バイト長は 30 バイトです。

相互運用性

SQL NCHAR データ型とその他のデータ型の間で操作が必要なとき、ユーザーは明示的な変換関数を適用するか、システムに頼る暗黙的な変換ができます。

- **明示的変換関数**

  明示的な変換は、わかりやすく、制御可能です。ユーザーは変換の方向を制御できます。Oracle は顧客の要件を満たすために、広範な変換関数を提供しています。明示的変換関数の例には、TO_NCHAR と TO_DATE があります（詳細な説明付きの完全なリストについては、*SQL Reference* を参照してください）。

- **暗黙的変換**

  Oracle は、SQL NCHAR とその他のデータ型（DATE、NUMBER、ROWID、RAW および TIMESTAMP など）の間だけでなく、SQL NCHAR と SQL CHAR データ型の間の暗黙的変換をサポートしています。異なるデータ型の間で操作されたとき、または引数の種類が公式の定義と異なるとき、暗黙的変換が発生します。このような操作には、SQL INSERT、UPDATE、および SELECT 文、PL/SQL 内の割当て、比較、SQL または PL/SQL 文中での連結、および SQL 関数などがあります。暗黙的変換は、SQL NCHAR への移行を大幅に簡素化します。暗黙的変換の例を次に示します。

  *INSERT INTO emp VALUES (100, **'Scott'**, **'Engineer'**, 10, '06-05-2001', 50000, 5);*

  emp 表の 2 番目と 3 番目の列（*ename* と *job*）は両方とも NVARCHAR2 として定義されています。表の列にデータが挿入される前に、*'Scott'* および *'Engineer'* リテラルが NCHAR キャラクタ・セット・エンコーディング（つまり AL16UTF16）に変換されます。

  注意: Oracle Database 10g では、CLOB と NCLOB 間で暗黙的変換がサポートされています。

データ消失に関する例外処理

データベース・キャラクタ・セットと NCHAR キャラクタ・セットが異なる場合、SQL CHAR 型と SQL NCHAR 型の間での暗黙的または明示的変換にはキャラクタ・セットの変換が伴います。キャラクタ・セットの変換を伴わない唯一の構成は、データベース・キャラクタ・セットと NCHAR キャラクタ・セットがともに UTF-8 のときです。これは一般的な構成ではなく、その他すべての場合にはキャラクタ・セットの変換が伴います。対応するマッピング文字が宛先キャラクタ・セットにない場合は、宛先バッファ内でデフォルトの置換文字が使用され、データ消失が発生します。ユーザーは、NLS パラメータ、NLS_NCHAR_CONV_EXCP

をセットして、このデータ消失の発生に警告を受け取るかを選択できます。このパラメータが TRUE にセットされていると、データ消失の場合に ORA-12713 エラーが戻されます。デフォルト値は FALSE です。デフォルトによるデータ消失は報告されません。このセッション・パラメータは、init.ora パラメータ・ファイル内の RDBMS インスタンス全体、あるいは ALTER SESSION 文を使用したカレント・ユーザー・セッション内の RDBMS インスタンス全体にも指定できます。セッション中は動的に変更できます。

SQL Unicode 文字列処理

NCHAR に対する SQL の明示的変換関数サポートのほか、Oracle は NCHAR データを処理できるすべての SQL 文字列関数を強化しました。SQL NCHAR データは SQL CHAR データと同じ方法で、任意の SQL 関数に引き渡せます。複数の文字列パラメータを持つ SQL 関数は、SQL CHAR と SQL NCHAR の混合パラメータを受け取り、処理前に正しく変換できます。

次に例を示します。
SELECT LENGTH(ename) from emp;
SELECT INSTR(ename, 'SCOTT', 1) from emp;

Unicode データ入力

UNISTR および NCHR 関数を使用すると、Unicode に対応しない環境またはデータベースでもユーザーは Unicode 文字列を入力できます。UNISTR 関数により、Unicode データは'\' と 16 進形式の文字の UTF-16 コード単位値により表現できます。NCHR は、各国語キャラクタ・セット内の数値とバイナリ等価の Unicode 文字を返します。

次に例を示します。
INSERT INTO emp (empno, ename) VALUES (10, UNISTR('abc \1E05'));

Oracle のすべてのメタデータは、データベース・キャラクタ・セット・エンコーディングで表現し処理されます。文字列リテラルがメタデータまたは問合せ文での指定が必要な場合、リテラル・データは SQL CHAR または SQL NCHAR データ型に属している可能性があります。UNISTR 関数は、文字列リテラルをデータベース・キャラクタ・セット内で表現できない SQL NCHAR データ型に指定するとき使用できます。次に例を示します。

CREATE TABLE Dept_tab (
Deptno NUMBER(3),
Dname NVARCHAR2(15),
Loc NVARCHAR2(15),
CONSTRAINT Loc_check1 CHECK (loc IN (UNISTR(' \7533'),N'NEW YORK')));

ASCIISTR 関数は UNISTR の逆の関数です。引数として任意キャラクタ・セット内の文字列をとり、その文字列の ASCII バージョンを返します。非 ASCII 文字は \xxxx の形に変換されます。ここで xxxx は UTF-16 コード単位を表します。

制限

SQL NCHAR 型は、SQL CHAR 型とほとんど同様に使用できますが、次に示す例外があります。

- Oracle Text は SQL NCHAR 型をサポートしません。

データベースの移行

Oracle は、移行処理の簡素化および作業の削減に豊富な機能セットを提供していますが、移行時には Unicode キャラクタ・セットの選択からスキーマの移行まで、いくつかの処理を必要とする問題があります。この章では、データベース・スキーマを移行するための詳細な手順のほか、各手順での移行処理に役立ついくつかのデータベース機能についても説明します。

Oracle9*i* および Oracle Database 10g の SQL NCHAR 型は、Oracle8*i* の SQL NCHAR 型とは著しく異なります。Oracle8*i* に SQL NCHAR 列がある場合、Oracle9*i* にデータベースをアップグレードするときには、付属の移行スクリプトを実行して Oracle9*i* NCHAR セマンティクスへの移行が必要になります。旧 SQL NCHAR 列を移行しておかないと、Oracle9*i* または Oracle Database 10g データベース・サーバーではアクセスできません。データベースの一貫性を維持するために、この移行手順は必須です。ここでは特に重要な問題は予想されないため、このような移行については説明しません。もっと一般的なシナリオは、次に詳しく説明する SQL CHAR 型から SQL NCHAR 型への移行です。

手順 1: Unicode キャラクタ・セットを選択する

UTF-16 と UTF-8 の両方を Unicode データ型のエンコーディング・フォーマットとして選択できます。これによって、データベース・サーバーは UTF-16 エンコーディングをネイティブにサポートできます。UTF-16 を内部エンコーディングとして使用するこれらのアプリケーションは、データベース・サーバーとさらにシームレスかつ効率的に連携できます。アジアおよび西側諸国の顧客はそれぞれのデータに、もっとも効率的でコンパクトなエンコーディングを選択できるため、UTF-16 と UTF-8 エンコーディング間での選択がデータをディスク上でさらに効率よく保存できます。Unicode 用にキャラクタ・セットを選択する方法は、アプリケーションが扱うデータおよびアプリケーション自体の内容に依存します。次に示す要因を考慮してください。

- **領域の効率性:** 英語～欧州言語からのデータには UTF-8 がもっともコンパクトですが、アジア系言語からのデータにとってはそれほどコンパクトではありません。アジア系言語からのデータには AL16UTF16 がもっともコンパクトですが、英語および欧州言語からのデータにとってはそれほどコンパクトではありません。データ記憶域がさらにコンパクトになれば、メモリー使用量が減り、ディスク領域が節約され、ディスク I/O も減るため、処理速度が改善されます。
- **文字列処理の速度:** AL16UTF16 文字列処理の方が一般的に高速です。
- **補充文字のサポート:** UTF-8 は補充文字に対応していませんが、AL16UTF16 は補充文字に対応しています。

- **Java または Windows アプリケーション:** UTF-16 は、Java 文字列および Windows 環境用のエンコーディング形式です。RDBMS と Java 文字列間で NCHAR データが交換されるとき、変換はありません。

開発者は Unicode データ型用にキャラクタ・セット・エンコーディングを選択するとき、前述の要因を考慮する必要があります。AL16UTF16 エンコーディングはデフォルトのエンコーディングです。データのほとんどが ASCII 以外のとき、または UTF8 の使用で領域要件上の効果がそれほど得られない場合は、UTF8 よりも AL16UTF16 の方が推奨されます。UTF8 はデータベース・キャラクタ・セットとして使用できるため、SQL NCHAR データ型に使用する要件は AL16UTF16 ほど厳密ではありません。

手順 2: 移行対象の列を識別する

Unicode データ型を使用すると、ユーザーは顧客ニーズに応じて、Unicode をサポートする部分的または全体的な移行ができます。Unicode のサポートに必要な列の数が限定されている場合、または Unicode への段階的な移行には、Unicode データ型の使用が必要です。全データベースの移行には、SYS およびシステム・スキーマを除くすべての文字列を SQL NCHAR データ型に変更する必要があります。この場合は Unicode データ型の使用よりもデータベース・キャラクタ・セットを移行した UTF-8 のサポートをお薦めします。

ユーザーがデータベース・スキーマを部分的に Unicode データ型に移行したいと考える様々な理由があります。1 つのシナリオは、全面的移行をすぐに必要としない場合です。今後その必要性が生じる可能性はあります。アプリケーション・スキーマ全体の移行では、さらに作業が増え、システムの停止時間が長くなることは確実です。この場合は、段階的移行が推奨され、Unicode データ型がこの移行を達成する特性を備えています。特定のスキーマが特定のアプリケーションに対応することがよくあります。これらのアプリケーションのいくつかは、多言語データのサポートに必要です。このような要件は、その他のいくつかのアプリケーションには（今後このような要件が生じる可能性はありますが）まだ該当しません。したがって、対応するスキーマを即座に移行する必要性はありません。ユーザーは最初に必要なスキーマを移行し、このような要件が生じたときに残りのスキーマを移行する選択ができます。

手順 3: 移行前の考慮事項

ここでは、実際に移行前の考慮事項について説明します。

新しい定義が制約のルールに準拠していない場合、列の移行に定義されている**制約**が違反の可能性があります。たとえば、修飾された列に主キー制約が定義されているとします。移行後、異なる複数のキー値が同じキー値に変換される可能性があります。ネイティブ・キャラクタ・セットと Unicode 間のマッピングが 1 対 1 のマッピングとは限らず、複数対 1 のマッピングでも構わないため、これは可能です。データ例外または縮小が原因で、修正済み列のバイト長も移行時に変更されます。制約事項に列のバイト長が含まれている場合、同様に違反があります。解決策は、移行前に制約を削除することです。移行後に制約をアクティブにするかは各ユーザーの要件に依存するため、ケースバイケースの判断が必要になります。

別の表への参照を制約されている列の場合、元の表と参照する表の両方を同時に移行する必要があります。従属表および参照する表内に対応する列は同じデータ型の必要があるため、SQL NCHAR 列は SQL CHAR 列を参照できません。またその逆の参照もできません。

最大列サイズ: CHAR/NCHAR または VARCHAR2/NVARCHAR2 列に許可される最大サイズは、2,000 および 4,000 バイトです。データベース・キャラクタ・セット内のシングルバイト文字は、AL16UTF16 内の 2 バイト文字です。さらに UTF8 キャラクタ・セット内では 3 バイト文字です。データ拡張の可能性があるため、すべての文字データが Unicode に変換後、最大サイズの制限が破られることがあります。ユーザーはサイズ制限が破られることの認識が必要です。実際の要件に応じて、ユーザーは代わりのデータ型として NCLOB の使用が必要な場合があります。

Unicode データ型に変更した列上に定義されている**トリガー**は、固有なトリガー条件が発生した場合にアクティベートされることがあります。トリガーの定義方法に応じて、条件には UPDATE、ALTER、CREATE などを含みます。開発者はこのような副作用を認識して予期しない影響を避けるため、トリガーの無効化が必要です。

区分化（パーティション化）: 列の文字エンコーディング（したがってバイナリ・ストレージ形式）、および長さセマンティクスは移行後に変わる可能性が高いため、区分（パーティション）化された列の Unicode データ型への移行がさらに複雑になります。旧バイナリ・ストレージ形式に基づいている区分（パーティション）は無効になります。区分表は、新規キャラクタ・セット・エンコーディングおよび長さセマンティクスに基づいた再区分化が必要になります。区分表の移行には、次の手順に従いエクスポートおよびインポート・ユーティリティの使用が推奨されます。エクスポートおよびインポート・ユーティリティの詳細については、『Oracle Database ユーティリティ』マニュアルを参照してください。

1. 移行対象の表をエクスポートします。
2. 列定義を Unicode データ型に変更します。
3. 表データを変更済み表にインポートします。

Character Set Scanner (CSSCAN)は、データベース・キャラクタ・セットの移行を考慮した特別な設計であり、移行に要する時間と作業の見積もりに役立つレポートが作成できます。ほとんどの場合、どの移行方法でも常に CSSCAN が生成したレポートの分析から開始する必要があります。具体的には、変換元データベース・キャラクタ・セットをネイティブに、ターゲット・データベース・キャラクタ・セットを UTF8 または AL16UTF16 に設定して、CHAR から NCHAR への移行をシミュレートできます。Character Set Scanner の詳細については、『Oracle 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。

手順 4: スキーマの移行

アプリケーションを Unicode データ型へ移行前に、ユーザーはスキーマの移行の検討が必要です。スキーマの移行にはいくつかのアプローチがあります。次にスキーマの移行のための異なるアプローチを示し、比較検討します。

'Alter Table' SQL コマンドの適用

既存の SQL CHAR 列を SQL NCHAR 列に変更、または既存の表に SQL NCHAR 列を追加に ALTER TABLE 文を使用できます。ユーザーは必要に応じて SQL NCHAR 列のまったく新しい表も定義できます。それぞれのケースに適したシナリオとプロシージャを説明します。

- 追加対象の新規 Unicode データは既存 SQL CHAR 表の列と一致している。たとえばアドレス情報は既存の名前フィールドに入れないでください。それ以外は、新規 Unicode データ型が既存のデータベース・キャラクタ・セットが網羅できない多言語データのサポートならば、既存の SQL CHAR 列を SQL NCHAR への移行を考慮してください。

  次に例を示します。
  グローバル企業の A 社はもともと西欧と米国での業務に焦点を合わせていました。A 社は WE8ISO8859P1 キャラクタ・セット付きのデータベースを持っています。NCHAR キャラクタ・セットは AL16UTF16 です。従業員表は次のように定義されています。

  emp (
  empno NUMBER(4),
  ename VARCHAR2(10),
  job VARCHAR2(9),
  mgr NUMBER(4),
  hiredate DATE,
  SAL NUMBER(7,2),
  deptno NUMBER(2));

  A 社はアジアに進出し、新しいアジアの従業員情報をデータベースに保存する必要が生じてきました。現在のデータベース・キャラクタ・セット、WE8ISO8859P1 にはアジア系文字は含まれていません。アジアの従業員情報は既存の *emp.ename* 列には保存できません。従業員名データは新規カテゴリではないため、追加列を *emp* 表に追加する必要がありません。最適な解決策は、*emp.ename* 列を VARCHAR2(10)から NVARCHAR2(10)へ変更することです。これはキャラクタ・セット・エンコーディングの問題を解決するだけでなく、列の長さセマンティクスを 10 バイトから 10 文字へ変更します。そのため、1 文字あたりに複数のバイトを必要とするマルチバイト文字が原因で発生する可能性の切捨て問題が回避されます。

  次に示す SQL コマンドは、SQL CHAR 型から SQL NCHAR 型へ表の列定義の変更に使用できます。

  ALTER TABLE emp MODIFY (ename NVARCHAR2(10));

  この ALTER TABLE コマンドは列定義を SQL CHAR 型から SQL NCHAR 型への変更だけでなく、列内のすべてのデータをデータベース・キャラクタ・セットから NCHAR キャラクタ・セットへ変換します。なお CLOB 列は ALTER TABLE コマンドの使用では、NCLOB に変更できないことに注意してください。*オンライン表再定義機能*を使用すれば、列を CLOB から NCLOB へ変換できます。詳細については、「オンライン表再定義」のセクションを参照してください。

移行する列に索引が作成されている場合は、各行の更新時に索引が更新されるため、索引を削除すると ALTER TABLE コマンドをスピードアップできます。

注意: NCHAR および NVARCHAR2 の最大列長は、2,000 バイトおよび 4,000 バイトです。NCHAR キャラクタ・セットが AL16UTF16 のとき、NCHAR および NVARCHAR2 列の最大サイズは 1,000 および 2,000 文字、つまり 2,000 および 4,000 バイトです。移行中にこのサイズ制限が違反される場合は、代わりに列を NCLOB に変更することを考慮してください。

- 元の列が意図していた目的とは別の目的を持つ新規の Unicode データの場合、既存の SQL CHAR 列に新規の Unicode データを収容することは適切ではありません。Unicode データのためにまったく新しい表の作成は必要ありません。既存の表のどれかに適合するかを検討してください。たとえば、いくつかのアジアの国では郷里や出生地情報が重要ですが、このシナリオではこの情報を反映できる既存の列はありません。既存の *emp* 表に Unicode データ型の追加列を足すことが適切です。次に示す SQL コマンドを使用して、Unicode 列を *emp* 表に追加できます。

  ALTER TABLE emp ADD (org NVARCHAR2(10));

  新規表定義は次のとおりです。

  emp (
  empno NUMBER(4),
  ename VARCHAR2(10),
  job VARCHAR2(9),
  mgr NUMBER(4),
  hiredate DATE,
  SAL NUMBER(7,2),
  deptno NUMBER(2),
  org NVARCHAR2(10));

- Unicode 列はまったく新しいものであり、目的は既存の SQL CHAR 列とはかなり異なります。Unicode データにはその他の情報もいくつか対応付けられています。そのため Unicode 列を持つ新しい表が必要になります。たとえば、A 社は従業員の本籍を追跡する必要もあります。次に示すコマンドを使用して、Unicode 列を持つ表を定義できます。

  CREATE TABLE addr (
  empno NUMBER(4),
  street NVARCHAR2(50),
  city NVARCHAR2(10),
  state NVARCHAR2(10),
  country NVARCHAR2(10),
  zip NUMBER(6));

オンライン表再定義

Unicode 型に移行を必要とする大量の行を持つ大きい表の場合、列内の全データの変換にはかなりの時間がかかります。この処理中、すべてのデータ列の読取り、あるいは更新にも使用できません。Oracle のオンライン表再定義機能を使用すれ

ば、移行の実施に必要な停止時間をかなり削減できます。この機能を使用すれば、移行処理の間、DML から表にアクセス可能です。表のサイズや再定義の複雑さに関係なく、非常に小さな時間枠の間だけ排他モードでロックされます。オンライン表再定義機能を使用した Unicode 型への移行手順を次に示します。

1. `DBMS_REDEFINITION.CAN_REDEF_TABLE()` プロシージャを呼び出して、オンラインで表が再定義できることを確認してください。表がオンライン再定義の候補でない場合、このプロシージャはエラーを生じ、表がオンラインで再定義できない理由が示されます。たとえば、*scott.emp* 表が移行中です。

   ```
   EXECUTE DBMS_REDEFINITION.CAN_REDEF_TABLE('scott', 'emp');
   ```

2. 所望の属性として SQL NCHAR 型を持つ空白の仮表を（再定義対象の表と同じスキーマ内に）作成します。

   ```
   CREATE TABLE int_emp (
   empno NUMBER(4),
   ename NVARCHAR2(10),
   job NVARCHAR2(9),
   mgr NUMBER(4),
   hiredate DATE,
   SAL NUMBER(7,2),
   deptno NUMBER(2),
   org NVARCHAR2(10));
   ```

3. 呼出しにより、再定義処理を起動します。

   ```
   DBMS_REDEFINITION.START_REDEF_TABLE().

   EXECUTE DBMS_REDEFINITION.START_REDEF_TABLE('SCOTT',
   'emp ',
   'int_emp',
   'empno empno,
   to_nchar(ename) ename,
   to_nchar(job) job,
   mgr mgr,
   hiredate hiredate,
   sal sal,
   deptno deptno,
   to_nchar(org) org');
   ```

   再定義を利用して CLOB 列を NCLOB 列に変更する方法は非常に似ています。前述の再定義文での違いは、対応する明示的変換関数が TO_NCHAR の代わりに TO_NCLOB でなければならない点です。

4. 仮表上に任意のトリガー、索引、権限付与、および制約を作成します。仮表を伴う参照制約（つまり仮表が参照制約の親表または子表の場合）がどれも使用不可能としての作成が必要です。再定義処理が完了、または異常終了するまで仮表上に定義されているトリガーは実行されません。

5. オプションとして、仮表 int_emp を同期化します。FINISH_REDEF_TABLE が呼び出される前に、多数の DML 操作が元の表に適用されている場合、SYNC_INTERIM_TABLE プロシージャを実行することで仮表は元の表と定期的な同期化が必要となります。これにより、FINISH_REDEF_TABLE()の再定義処理が完了するまでにかかる時間が短縮されます。

   EXECUTE DBMS_REDEFINITION.SYNC_INTERIM_TABLE ('scott', 'emp', 'int_emp');

6. DBMS_REDEFINITION.FINISH_REDEF_TABLE()プロシージャを実行して、表の再定義を完了します。

   EXECUTE DBMS_REDEFINITION.FINISH_REDEF_TABLE ('scott', 'emp', 'int_emp');

   このプロシージャの結果、元の表には次の処理が適用されます。

   - 仮表のすべての属性、索引、制約、権限付与、およびトリガーを持つように元の表が再定義されます。
   - 仮表の参照制約が再定義後の表を伴うことで使用可能になります。

7. 仮表を削除します。

   DROP TABLE int_emp;

   前述のプロシージャの最終結果は次のとおりです。

   - 元の表は Unicode 列へ移行されます。
   - 仮表上の START_REDEF_TABLE()と FINISH_REDEF_TABLE()の間に定義していたトリガー、権限付与、索引、および制約は再定義後の表に定義されます。再定義処理前の仮表を伴う参照制約は終了し、再定義後の表を伴うことで使用可能な状態になります。
   - 元の表に（再定義前に）定義されていた索引、トリガー、権限付与、および制約は仮表に移され、ユーザーが仮表を削除するときに削除されます。再定義前の元の表を伴う参照制約は、仮表を伴うことで使用禁止になります。
   - 元の表に（再定義前に）定義されていたどの PL/SQL プロシージャおよびカーソルも無効になります。次に使用されるときに自動的に再び有効になります（再定義処理の結果として表の形が変更されている場合、この再有効化は失敗します）。

   注意: オンライン表再定義に適用されるいくつかの制限があります。詳細については、『Oracle Database 管理者ガイド』を参照してください。

CLOB スキーマ移行の例

次に、CLOB 列を NCLOB 列へ移行する例を示します。列マッピング仕様の違いに注意してください。元の CLOB 列を NCLOB 列へマップするには TO_NCLOB が使用されます。

*CREATE TABLE lb (anum NUMBER PRIMARY KEY, lb **CLOB**);*
*CREATE TABLE int_lob (n1 NUMBER PRIMARY KEY, nlb **NCLOB**);*

EXECUTE DBMS_REDEFINITION.CAN_REDEF_TABLE('Scott', 'lb');
EXECUTE DBMS_REDEFINITION.START_REDEF_TABLE('Scott', 'lb', 'int_lob', 'anum n1, ***TO_NCLOB****(lb) nlb');*
EXECUTE DBMS_REDEFINITION.SYNC_INTERIM_TABLE('Scott', 'lb', 'int_lob');
EXECUTE DBMS_REDEFINITION.FINISH_REDEF_TABLE('Scott', 'lb', 'int_lob');

DROP TABLE int_lob;

オンライン表再定義の使用で、表の停止時間が削減されます。これは、再定義が最終的な完了前に、元の表では最小限の DML 操作を想定しています。再定義の完了前、元の表の DML 操作は若干ゆっくりです。

"Alter table（表変更）"と"On-line Table Redefinition（オンライン表再定義）"の比較

ALTER TABLE コマンドとオンライン表再定義はともに列の定義が変更できます。特定のシナリオにおけるそれぞれの利点があります。

ALTER TABLE コマンド:

a. 使いやすい。
b. 制限が少ない。

オンライン表再定義:

a) よりよいパフォーマンス。オンライン表再定義は通常、コマンドよりも高速。
b) 一度に複数の列を移行できる。
c) 移行処理中もほとんどの時間 DDL 用に表が使用可能。
d) 表の断片化を回避できるため、領域節約とより高速なアクセスが可能になる。
e) CLOB から NCLOB への移行に役立つ。

手順 5: 移行後の作業

移行後のデータベース・スキームを元の状態に復元するためには、いくつかの操作が必要になります。またスキーム移行の結果、関連するアプリケーションにその他の間接的な影響を及ぼすこともあります。ユーザーはこれらの影響に注意を払い、防止するための適切な処置が必要です。

索引: ALTER TABLE MODIFY コマンドによって表の列が SQL CHAR 型から Unicode データ型に変更されたとき、それをベースに構築されている索引はデータベース・システムによって自動的に変更します。ただし、これはコマンドのパフォーマンスも低下します。ALTER TABLE コマンドの発行前に索引が削除される場合は、移行後の再作成が必要になります。

制約: 移行前に使用禁止にされた制約は、移行後に再度使用可能にする必要があります。

トリガー: 移行前に使用禁止にされたトリガーは移行後に再度使用可能にする必要があります。

レプリケーション: Unicode 型に移行された列がいくつかのサイトにわたってレプリケートされる場合、移行によるデータ変更はレプリケーション定義に応じて、同期または非同期にそれぞれのサイトに伝播されます。

バイナリ順序: データベースと NCHAR が異なるキャラクタ・セットを持つ場合、SQL CHAR 列から Unicode データ型への移行はキャラクタ・セットの変換を伴います。キャラクタ・セットが異なると同じデータでもバイナリ順序が異なるため、この順序に依存しているアプリケーションに影響を及ぼす可能性があります。

Unicode データ型へのアプリケーションの移行

Oracle は中間層またはクライアントにおいて、アプリケーションが使用できるデータベース・アクセス・インタフェースの総合的なセットを提供しています。すべての主要なアクセス・インタフェースは Unicode データ型を様々なレベルでサポートします。この章では、Unicode データ型対応にするため、既存のアプリケーションの移行方法について説明します。ポピュラーな 3 つのデータベース・アクセス・インタフェースと環境、PL/SQL、OCI、および JDBC に焦点を当てます。SQL CHAR 型を含むその他の Oracle データ型と SQL NCHAR 型の間の暗黙的変換は、Unicode データ型への移行の処理負荷を減らします。SQL CHAR 型と SQL NCHAR 型の間の操作方法に関して、アプリケーションが暗黙的変換に依存する場合は、新たに追加された SQL NCHAR 列または表にアプリケーションを適合させるためのわずかな変更が必要です。すべての Unicode 列が SQL CHAR 列から変換され既存スキーマに追加された新規 Unicode 列がない場合、既存データの代わりに既存アプリケーションを Unicode データ型で実行するための変更は、論理上必要としません。ただし、データベース・キャラクタ・セットで網羅されない新規 Unicode データの追加には、可能なデータ消失を回避するアプリケーションの移行が必要となります。また、Unicode データでアプリケーションをさらに効率よく実行させるには、コードの変更もパフォーマンスの改善に役立ちます。次に Unicode データ型へ移行するガイドラインを示します。移行を簡単にして、よりよいパフォーマンスの結果となります。

PL/SQL アプリケーション

ユーザーは PL/SQL アプリケーションの移行時、いくつかの事項に注意が必要です。注意事項は次のサブセクションで説明します。

PL/SQL 変数

PL/SQL 環境で処理されるいくつかのデータ型または PL/SQL 変数があります。データベース表の列とやりとりするデータの保持および処理するため、変数の 1 つの型が使用されます。このような変数は、次に示すカテゴリに分類されます。

a. データベースから受け取ったデータまたはデータベースの列に挿入されるデータを保持。

b. 比較、連結、および SQL 関数などの操作でデータベース表の列とのやりとり。

c. カテゴリ a および b の変数とのやりとり。

上記カテゴリの PL/SQL 変数は、やりとりするデータベースの列と同期化が必要です。全体的な Unicode データ型に移行されるデータベースの場合、これらの変数は SQL NCHAR 型として定義が必要です。部分的に Unicode に移行されるデータベースの場合、ユーザーは表の列のデータ型に注意が必要となります。%TYPE および%ROWTYPE 構文を使用して、このような PL/SQL 変数をデータベースの列と同期化ができます。この構文の使用例を次に示します。

```
DECLARE
my_ename NVARCHAR2(10);
my_job emp.job%TYPE;
BEGIN
SELECT ename, job INTO my_ename, my_job FROM emp where ename=’SCOTT’;
END;
```

CHAR と NCHAR 間の暗黙的変換は必要に応じて自動的に行われるため、PL/SQL 変数とデータベース列の間の同期化または PL/SQL 変数間の同期化は必須ではありません。ただし、次に示す結果を回避するため、ユーザーが常に上記のガイドラインに従うことをお薦めします。

1. データ消失: SQL NCHAR 列のデータが選択されて SQL CHAR PL/SQL 変数に挿入、SQL NCHAR 変数が SQL CHAR データベースの列に挿入、あるいは SQL NCHAR 変数が SQL CHAR 変数に割り当てられるとき、データベース・キャラクタ・セットが NCHAR キャラクタ・セットのサブセットにすぎない場合、データ消失の可能性があります。
2. パフォーマンス・オーバーヘッド: SQL CHAR と SQL NCHAR 間の変換は、パフォーマンス・オーバーヘッドの可能性があります。同期化はこのような種類のオーバーヘッドを除去します。

PL/SQL 変数の別のカテゴリは、メタデータに関係するもの（ユーザー名、パスワード、表、ビュー、プロシージャ、関数、および列など）および SQL 文です。これらは SQL CHAR 型としてデータベース内で保存し、処理されます。したがってこのカテゴリの PL/SQL 変数の移行は必要ありません。最終的にはネイティブのデータベース・エンコーディングを使用した SQL CHAR 型へ変換し戻されるため、SQL NCHAR 型への移行は不要な変換とパフォーマンス・オーバーヘッドを生じます。

次に例を示します。

```
DECLARE
sql_command VARCHAR2(100);
……
BEGIN
sql_command : = 'INSERT INTO ' || tabName || ' (col1, col2, col3 )
VALUES( :var1, :var2, :var3 )';
dbms_sql.parse( sqlCursor, sqlCommand, dbms_sql.v7 );
dbms_sql.bind_variable( sqlCursor, ':var1', varVal );
……
END;
```

前述の例は、***SQL command*** 変数を Unicode データ型への変更が必要ないことを示しています。

直接的、または間接的にデータベースとやりとりしない PL/SQL 変数もあります。これらの変数は、データベース・キャラクタ・セット内にデータが定義されているかにより、SQL NCHAR 型または SQL CHAR 型として定義できます。

```
DECLARE
var1 VARCHAR2(10);
var2 VARCHAR2(20);
……
BEGIN
…..
var2 : = var1;
……
END;
```

PL/SQL 文字列リテラル

文字列リテラルが SQL NCHAR 型かどうかを示すため、文字'N'を文字列リテラルの前に付けることができます。'N'プリフィックスは必須ではありません。必要に応じてリテラルを SQL NCHAR 型に変換するとき、ユーザーは'N'を省略し、暗黙的変換に依存できます。PL/SQL プログラムに埋め込まれている文字列リテラルは、SQL NCHAR 型とやりとりする場合としない場合があります。SQL NCHAR 変数とやりとりする文字列リテラルの場合、リテラルが NCHAR 型であることを示すため、文字列リテラルの単一引用符の前に'N'を付けることをお奨めします。結果として、リテラルは NCHAR キャラクタ・セット・エンコーディングに変換され、コンパイル時に SQL NCHAR データと同様に扱われます。それ以外の場合、リテラルはコンパイル時に CHAR 型として扱われ、実行時における文の実行ごと NCHAR に変換されます。リテラルがループ内にある、または包含プロシージャが複数回呼び出される場合、これは変換オーバーヘッドを削減します。例を示します。

```
DECLARE
name NVARCHAR(2000);
BEGIN
  FOR i IN 1 .. 2000 LOOP
    name : = name || N' ';
  END LOOP;
END;
```

PL/SQL 文字列の比較

SQL クエリー内の SQL CHAR 型と SQL NCHAR 型の比較演算の場合は、データ消失を回避するため、いつも暗黙的変換が SQL CHAR 型から SQL NCHAR 型へ変換します。2 つのデータベース列の間、データベース列とリテラル、SQL 関数、またはクエリーの結果との間で比較されます。データ消失が発生しない場合、ユーザーは明示的変換機能を適用して、変換数の少ない方を変換できます。このように PL/SQL、OCI、またはその他の開発関数に埋込み可能なすべての SQL クエリーに該当します。

次に例を示します。

tab_a (col_a VARCHA2(100), ……) has 1 million rows

table tab_b (col_b NVARCHAR2(100), ……) has 100 rows
select col_a from tab_a, tab_b where tab_a.col_a = TO_CHAR(tab_b.col_b);

明示的変換は、*col_b* を Unicode 型から VARCHAR2 型へ変換するため、結果として 100 の変換で済みます。暗黙的変換に依存すると、*col_a* が Unicode 型に変換されるため、結果として 1,000,000 の変換が生じます

PL/SQL プロシージャ/関数

SQL CHAR 型のみをサポートしている PL/SQL プロシージャ/関数に、SQL CHAR 型と SQL NCHAR 型の両方をサポートさせるには移行が必要です。Unicode 型対応の移行をしない場合、SQL NCHAR 型として PL/SQL プロシージャに引き渡された Unicode データは、暗黙的に SQL CHAR 型にまず変換されます。Unicode データの一部がデータベース・キャラクタ・セットにマッピングできない場合、データ消失が発生します。PL/SQL プロシージャ内で SQL NCHAR 型をサポートする方法は 3 通りあります。

- ローカル変数のほか、引数および PL/SQL プロシージャ/関数の戻り型を SQL CHAR 型から SQL NCHAR 型へ変換します。次に例を示します。

  FUNCTION get_str(cur IN NVARCHAR2,
  old IN NVARCHAR2,
  new IN NVARCHAR2)
  RETURN NVARCHAR2 IS
  local_var NVARCHAR2(10);
  BEGIN
  local_var : = cur;
  ……
  END;

  前述の PL/SQL プロシージャは SQL NCHAR 型を使用して定義されていますが、SQL CHAR データ型と SQL NCHAR データ型の両方で正しく機能します。実行時、SQL CHAR 型の引数が引き渡されると、暗黙的変換が自動的に SQL CHAR データを SQL NCHAR データに変換できます。結果として、多少のパフォーマンス・オーバーヘッドが生じます。

- ANY_CS を使用して、CHAR/VARCHAR2/CLOB 型のプロシージャ/関数へのパラメータのデータ型を宣言できます。これは、パラメータが実行時に実際の引数値からキャラクタ・セット（SQL CHAR 型か SQL NCHAR 型かを決定する）の継承を示します。%CHARSET は、キャラクタ・セット名 ANY_CS で宣言されているパラメータまたは変数のキャラクタ・セットをコピーできます。同じパラメータ・リスト上の前方のパラメータ名に適用し、前方のパラメータと同じキャラクタ・セットで、宣言対象のパラメータを引き渡す必要があることを示します。またローカル変数の宣言にも使用し、引き渡されたパラメータと同じキャラクタ・セット内にある、または別の文字型変数と同じキャラクタ・セット内にあることを示せます。ANY_CS と%CHARSET を使用して、PL/SQL プロシージャ/関数は SQL CHAR と SQL NCHAR の両方の型をサポートできます。同時に、変数が互いに同じキャラクタ・セットであることを保証できます。これは、SQL CHAR 型と SQL NCHAR 型の間の操作に関するキャラクタ・セット変換の数を減らします。

FUNCTION get_str(cur IN VARCHAR2 ***CHARACTER SET ANY_CS,***
old IN VARCHAR2 CHARACTER SET ***cur%CHARSET,***
new IN VARCHAR2 CHARACTER SET ***cur%CHARSET)***
RETURN VARCHAR2 CHARACTER SET ***cur%CHARSET*** *IS*
local_var VARCHAR2(10) CHARACTER SET ***cur%CHARSET;***
BEGIN
…..
END;

- SQL NCHAR 型用には別のプロシージャ/関数を定義してください。これは、再定義が必要なプロシージャ/関数が少数のときのみ使用できます。

長さセマンティクスの変更

文字長セマンティクスは、Unicode データ型の唯一の長さセマンティクスです。PL/SQL アプリケーションはこのセマンティックの変更に応じた調整が必要です。データベースの列と同期化される PL/SQL 変数の場合は、同時に長さセマンティクスも同期化されます。これはつまり、%TYPE または%ROWTYPE 構文に基づいた変数が SQL NCHAR 型だと判断された場合、長さセマンティクスが自動的に文字長セマンティクスになることを意味します。PL/SQL プロシージャ仕様内のパラメータ・リストには、長さの制約がなにもありません。そのため PL/SQL プロシージャの場合、長さセマンティクスは問題になりません。実行時、SQL NCHAR データがプロシージャに引き渡されるとき、パラメータの長さセマンティクスは暗黙的に文字長セマンティクスとなります。

文字列関数

PL/SQL アプリケーションでは、長さ関連の SQL または PL/SQL 内部関数が様々な目的で使用されます。これらの関数のいくつか、たとえば LENGTHB、SUBSTRB、および INSTRB などはバイト数に基づいています。その他のいくつかの関数、たとえば LENGTH、SUBSTR、INSTR または LENGTH2、SUBSTR2、INSTR2 などは文字数に基づいています（『Oracle Database SQL リファレンス』を参照してください）。PL/SQL 変数およびプロシージャが Unicode データ型に移行後、同時にそれらを操作する内部関数も移行が必要となります。これらの内部関数は、様々な目的に使用されます。調査して正しい移行が必要です。次にこれらの使用方法を簡単にまとめます。

I. ほとんどの場合、シングルバイト・データとマルチバイト・データの両方に文字ベースの関数が使用されます。文字長セマンティクスについては理解済みであるため、特別なことは必要ありません。

II. バイト・ベースの関数が使用されるのは次に示すシナリオです。

- バイト長が文字長と同じかをチェックすることで、マルチバイト・キャラクタの有無を調べます。

  DECLARE
  context NVARCHAR2(10);
  …..
  BEGIN

…..…
if length(context) != lengthb(context) then…
END;

*コンテキスト*は現在 Unicode 型であり、すべての文字が少なくとも 2 バイト長であることが必要なため、AL16UTF16 エンコーディングに対してこのチェックはいつも失敗します。

- 入力文字列が制限（バイト数）より短いかをチェックします。SQL 関数は文字ベースに変更し、相当する制限の単位も文字数に変更する必要があります。

  If lengthb(context) > 10 then raise context_too_long;
  次のように変更:
  If length(context) > 10 then raise context_too_long;

- 文字列内のすべてのバイトを処理するため、ループ境界としてバイト長を使用します。処理は文字単位の処理に変更できます。バイト・ベースの関数は文字ベースの関数に変更の必要があります。

  DECLARE
  buffer NVARCHAR2(200);
  val NUMBER;
  …..…
  BEGIN
  FOR i IN 1 .. LENGTHB(buffer) LOOP
  val : = ASCII(SUBSTRB(buffer, i, 1));
  …
  END LOOP;
  END;

  LOOP 文は次のような変更が必要です。
  FOR i IN 1 .. LENGTH(buffer) LOOP
  val : = ASCII(SUBSTR(buffer, i, 1));

  元の PL/SQL プログラムはシングルバイト・キャラクタ・セットに対してのみ有効です。アプリケーションは Unicode データ型対応に移行後、正しく機能するための変更が必要です。AL16UTF16 エンコーディングのシングルバイトは、意味を持ちません。意味を持つのは 2 バイト・コードのみです。関数は、変数と同じ長さセマンティクスになる変更が必要です。したがって LENGTHB と SUBSTRB は、LENGTH と SUBSTR に変更が必要です。

- 文字列のバイト位置には特別な意味があります。バイト位置は、文字位置に変更できます。

  DECLARE
  colon NUMBER;
  doc_info NVARCHAR2(200);
  node_id NUMBER;
  BEGIN
  colon : = INSTRB(doc_info, ':');
  node_id : = to_number(SUBSTRB(doc_info, 1, colon-1));
  END;

最後の 2 行は、次のような変更が必要です。
colon : = INSTR(doc_info, ':');
node_id : = to_number(SUBSTR(doc_info, 1, colon-1));

常にキャラクタ・セマンティックを持つ NVARCHAR2 としてオペランドが定義されているため、INSTRB と SUBSTRB は INSTR と SUBSTR に変更する必要があります。

- 文字列が別の変数に合うことを確認してください。関数の長さセマンティクスが宛先変数定義の一貫性を確認する必要があります。

  DECLARE
  MSG NVARCHAR2(2000);
  …..…
  BEGIN
  MSG : = substrb(MSG||' (`||TOK_NAM||'='||TOK_VAL||')',1,2000);
  …..…
  END;

  次のように変更:
  MSG : = substr(MSG||' (`||TOK_NAM||'='||TOK_VAL||')',1,2000);

  MSG が文字長セマンティクスとして定義されているため、SUBSTRB を SUBSTR に変更する必要があります。
  SUBSTRB は文字を途中で切り捨てる結果になります。

上記の使用法シナリオに基づいて、SQL NCHAR 型を扱うバイト・ベース関数は文字ベース関数に変更する必要があるという結論となります。

III. 長さセマンティクスの異なる SQL 関数

Oracle は各種のアプリケーション環境を満たすため、長さ関連の SQL 関数を提供しています。SUBSTR、SUBSTRB、SUBSTR2、SUBSTR4、および SUBSTRC のように、関数名に連結されている文字または数字によって区別されます。関数間の主な違いを次に説明します。それは LENGTH、LENGTHB、LENGTH2、LENGTH4、LENGTHC および INSTR、INSTRB、INSTR2、INSTR4、INSTRC にも該当します。

SUBSTR は、データ型のキャラクタ・セットにより定義された文字単位で長さを測定します。たとえば、AL32UTF8 は UCS4 コード単位で計算されます。UTF8 と AL16UTF16 は、UCS2 コード単位です。そのため、補充文字は AL32UTF8 では 1 文字、AL16UTF16 では 2 文字としてカウントされます。またこれは、SUBSTR が VARCHAR と NVARCHAR とで異なる文字単位を使う可能性も意味します。一貫性が重要な場合は、SUBSTR2 または SUBSTR4 を使用して、セマンティクスのすべての演算を UCS2 または UCS4 に強制することをお薦めします。

SUBSTRB は、バイト単位で長さを計算します。

SUBSTR2 は、Java 文字列および Windows クライアント環境に準拠している UCS2 コード単位で長さを計算します。補充文字は 2 コード単位としてカウントされます。

SUBSTR4 は UCS4 コード単位で長さを計算します。補充文字は 1 コード単位としてカウントされます。

SUBSTRC は、Unicode の完全文字単位で長さを計算します。補充文字と複合文字は、1 文字としてカウントされます。

CLOB から NCLOB への移行

前述の説明はすべて、CLOB から NCLOB への移行にも該当します。Oracle Database 10g では、CLOB と NCLOB 間で暗黙的変換がサポートされています。ただし Oracle9*i* および一般的なベスト・プラクティスでは、TO_NCLOB および TO_CLOB SQL 関数は明示的操作が必要です。暗黙的変換に依存する操作を試みると、Oracle9*i* ではコンパイル・エラーが発生します。次の例では、TO_NCLOB が使用されています。

```
DECLARE
alb CLOB;
nlb NCLOB;
BEGIN
  nlb : = CONCAT(nlb, TO_NCLOB(alb));
  ……
    END;
```

OCI アプリケーション

SQL NCHAR 型用として別の外部 OCI 型はありません。SQL NCHAR 型用の外部 OCI 型は、SQL CHAR 型のものと同じです。OCI はバインド/定義バッファ用に OCI 属性、OCI_ATTR_CHARSET_FORM 経由で Unicode データ型をサポートします。このパラメータの値は、SQLCS_IMPLICIT（データベース・キャラクタ・セット ID を示している）および SQLCS_NCHAR（各国語キャラクタ・セット ID を示している）です。OCI_ATTR_CHARSET_FORM が SQLCS_NCHAR にセットされているとき、データはクライアント上の NCHAR キャラクタ・セット設定からサーバー上の NCHAR キャラクタ・セットに変換されます。またはその逆方向に変換されます。OCI_ATTR_CHARSET_FORM が SQLCS_CHAR にセットされているとき、定義/バインド・バッファ内のデータがクライアント側のデータベース・キャラクタ・セットから変換されます。またはその逆方向に変換されます。次に OCI 使用法シナリオを一覧し、各ケースでの OCI アプリケーションの移行方法について説明します。

- 完全な Unicode アプリケーションでの OCI: UTF-16 として SQL CHAR および SQL NCHAR キャラクタ・セットの指定には OCIEnvNlsCreate の使用をお薦めします。SQL CHAR および SQL NCHAR の両方に *OCI_UTF16ID* が指定されている場合、NLS_LANG 設定に関係なくすべての文字データは UTF-16 エンコーディングです。これにはユーザー・データのほか、すべてのメタデータが含まれます。OCI アプリケーションが SQL NCHAR 列からデータを検索して定義

バッファに入れるとき、またはバインド・バッファからデータを SQL NCHAR 列に挿入するとき、OCIAttrSet を呼び出して定義/バインド・バッファに対応付けられている OCI_ATTR_CHARSET_FORM 属性をセットする必要があります。OCI_ATTR_CHARSET_FORM 属性を SQLCS_NCHAR にセットしない場合、OCI によるデータベース・サーバーへ送信前にバインド・バッファ内のすべてのデータがデータベース・キャラクタ・セットに変換されます。データベース・キャラクタ・セットが NCHAR キャラクタ・セットのサブセットにすぎない場合、データの最終的な宛先が SQL NCHAR データ型の列なら、データが消失する可能性があります。このようなデータ消失は、OCI_ATTR_CHARSET_FORM 属性を SQLCS_NCHAR にセットすることで回避できます。

Unicode プログラミング用の OCIEnvNlsCreate:

OCIEnvNlsCreate(envhpp, OCI_DEFAULT, …, OCI_UTF16ID ,OCI_UTF16ID);

OCI_ATTR_CHARSET_FORM 属性は次の関数によってセットできます。

ub1 charsetfm = SQLCS_NCHAR;
*OCIAttrSet((dvoid *)bindp, (ub4) OCI_HTYPE_BIND, (dvoid *)&charsetfm, (ub4) 0, (ub4) OCI_ATTR_CHARSET_FORM, errhp));.*

- その他のアプリケーションでの OCI: SQL NCHAR 列からのデータを扱う OCI アプリケーションには各種あります。そのようなアプリケーションには、ユーザーとの相互作用のエンド・ユーザー・アプリケーション、またはデータを通すだけでデータ処理をしないゲートウェイのような中間層アプリケーションがあります。データ処理を行うアプリケーションでもかまいません。各ケースにおいて、クライアント・バッファが対応付けられている列が SQL NCHAR 型なら OCI_ATTR_CHARSET_FORM 属性を SQLCS_NCHAR にセットし、列が SQL CHAR 型なら OCI_ATTR_CHARSET_FORM 属性を SQLCS_IMPLICIT にセットする（これがデフォルトのケースなため、ユーザーはセットする必要がありません）ことをお薦めします。理由は 2 つあります。
  - データベース・キャラクタ・セットが NCHAR キャラクタ・セットのサブセットにすぎない場合、これはデータ消失を回避します。
  - この属性がデータベースの列と同じキャラクタ・セット構成になるようなセット後、サーバー側で変換する代わりに、定義/バインド・バッファ内のデータが適切なキャラクタ・セットに変換されます。これは、サーバーの処理負荷を削減できます。

- OCI での CLOB から NCLOB への移行: CLOB から NCLOB への移行は、その他の SQL CHAR から SQL NCHAR データ型への移行と似ています。Oracle9*i* では、CLOB と NCLOB 間の暗黙的変換はサポートされていません。ユーザーは、ターゲット表の列の型に応じた OCI_ATTR_CHARSET_FORM 属性を正しくセットする必要があります。そうしないと、エラーが戻されます。なお Oracle Database 10g 以降は、CLOB と NCLOB 間の暗黙的変換がサポートされているため、エラーは戻されません。NCLOB 型の処理は NCHAR/NVARCHAR2 とは相違点が多く、NCLOB アプリケーション向けのコーディング方法を理解する例を次に示します。

次の例は、*FOO (A INT, C NCLOB)*表に基づいています。

```
char *insstmt = (char *)"INSERT INTO FOO (A, C) VALUES (1, :1);"
char *selstmt = (char *)"SELECT C FROM FOO WHERE A = 1";
ub1 buf[MAXBUFLEN];
ub1 *rbuf;
ub4 blen = 0;
ub4 loblen = 0;
OCILobLocator *clob;

if (OCIDescriptorAlloc((dvoid *) envhp, (dvoid **) &clob,
                (ub4)OCI_DTYPE_LOB, (size_t) 0, (dvoid **) 0))
{
  return OCI_ERROR;
}

if (OCIStmtPrepare(stmthp, errhp, insstmt, (ub4)strlen((char *)insstmt),
             (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT))
{
  return OCI_ERROR;
}

memset((void *) buf, (int) 'A', (size_t) MAXBUFLEN);

if (OCIBindByPos(stmthp, &bndhp[0], errhp, (ub4) 1,
                (dvoid *) buf, (sb4) inputlen,
                SQLT_CHR,
                (dvoid *) 0, (ub2 *)0, (ub2 *)0,
                (ub4) 0, (ub4 *) 0, (ub4) OCI_DATA_AT_EXEC))
{
  return OCI_ERROR;
}

if (OCIAttrSet((dvoid *) bindp, (ub4) OCI_HTYPE_BIND,
                  (dvoid *) SQLCS_NCHAR, (ub4) 0,
                  (ub4) OCI_ATTR_CHARSET_FORM, errhp))
{
  return OCI_ERROR;
}

retval = OCIStmtExecute(svchp, stmthp, errhp, (ub4) 1, (ub4) 0,
                  (OCISnapshot*) 0, (OCISnapshot*) 0,
                  (ub4) OCI_DEFAULT);

/** The following statements may not be needed for this example.
   But just in case the character set and id are needed in other scenarios. **/

if (OCILobCharSetId(envhp, errhp, clob, &csid))
{
  return OCI_ERROR;
}

if (OCILobCharSetForm(envhp, errhp, clob, &csform))
```

```
{
 DISCARD printf("FAILED: OCILobCharSetForm() \n");
 report_error(errhp);
}

if (OCIStmtPrepare(stmthp, errhp, selstmt, (ub4)strlen((char *)selstmt),
            (ub4) OCI_NTV_SYNTAX, (ub4) OCI_DEFAULT))
{
 return OCI_ERROR;
}

if (OCIStmtExecute(svchp, stmthp, errhp, (ub4) 0, (ub4) 0,
            (OCISnapshot*) 0, (OCISnapshot*) 0,
            (ub4) OCI_DEFAULT))
{
 return OCI_ERROR;
}

if (OCIDefineByPos(stmthp, &dfnhp, errhp, (ub4) 1,
            (dvoid *) &clob, (sb4) -1,
            SQLT_CLOB, (dvoid *) 0,
            (ub2 *) 0,(ub2 *) 0, (ub4) OCI_DEFAULT))
{
 return OCI_ERROR;
}

if (OCIAttrSet((dvoid *) bindp, (ub4) OCI_HTYPE_BIND,
                (dvoid *) &csform, (ub4) 0,
                (ub4) OCI_ATTR_CHARSET_FORM, errhp))
{
 return OCI_ERROR;
}

if (OCIStmtFetch(stmthp, errhp, (ub4) 1, (ub2) OCI_FETCH_NEXT,
            (ub4) OCI_DEFAULT))
{
 return OCI_ERROR;
}

/* OCILobGetLength returns the lob length in number of characters */
if (OCILobGetLength(svchp, errhp, locator, &loblen))
{
 return OCI_ERROR;
}

/** Maximum number of bytes per character is 4. The character length multiplied
   by the maximum number of bytes per character can guarantee the retrieved data
   fits in the buffer **/

#define MAX_BYTE_LEN 4
blen = loblen*MAX_BYTE_LEN;
```

```
rbuf = (ub1 *)malloc(blen);
memset((void *) rbuf, (int) ' \0', (size_t) blen);

if (OCILobRead(svchp, errhp, locator, &amtp, (ub4) pos, (dvoid *) rbuf,
        (ub4) loblen, (dvoid *)0,
        (sb4 (*)(dvoid *, CONST dvoid *, ub4, ub1)) 0,
        (ub2) 0, (ub1) SQLCS_NCHAR ))
{
 report_error(errhp);
}
else
{
 if (memcmp((const void *) rbuf, (const void *) rbuf2, (size_t)MAXBUFLEN))
   DISCARD printf("FAILED: OCILobRead(); buffers differ \n");
 else
DISCARD printf("PASSED: OCILobRead(); buffers equal \n");
}
```

JDBC アプリケーション

JDBC では、SQL NCHAR 型用に対応する別のデータ型またはクラスが定義されていません。SQL CHAR データ型と同じクラスとメソッドを使用して SQL NCHAR データ型にアクセスします。そのため SQL NCHAR データ型の使用方法は、SQL CHAR データ型と類似しています。SQL CHAR 型の扱いと SQL NCHAR 型の扱いの主な相違点を次に示します。

a. JDBC プログラムがデータをバインドするとき、setFormOfUse()メソッドを呼び出して、データが SQL NCHAR データ型用にバインドされることの指定が必要です。これはキャラクタ・セット構成属性をセットできる OCIAttrSet 関数に似ています。キャラクタ・セット構成用には、FORM_CHAR と FORM_NCHAR の 2 つの有効な値があります。FORM_CHAR はデフォルト値です。使用構成として FORM_NCHAR がセットされている場合、JDBC ドライバはサーバーの各国語キャラクタ・セットでデータを内部的に表現します。サーバー上の SQL NCHAR データ型に対応するすべてのバッファに対して FORM_NCHAR の使用が必要となります。次のコードは、SQL NCHAR データ型のアクセス方法です。

```
int empno = 12345;
String ename = " \uFF2A \uFF4F \uFF45";
String job = “Engineer”;
oracle.jdbc.OraclePreparedStatement pstmt =
(oracle.jdbc.OraclePreparedStatement)
conn.prepareStatement("INSERT INTO emp (empno, ename, job) VALUES(?, ?, ?)");

pstmt.setFormOfUse(2, FORM_NCHAR);
pstmt.setFormOfUse(3, FORM_NCHAR);

pstmt.setInt(1, 1);
pstmt.setString(2, ename);
pstmt.setString(3, job);
```

pstmt.execute();
pstmt.close();

注意: 正しい結果を得るため、*setFormOfUse* を *setString* の前に呼び出す必要があります。

b. oracle.sql.CHAR クラスは、Oracle オブジェクト・タイプに埋め込まれている SQL CHAR データ型と SQL NCHAR データ型の両方を目的として設計されています。Oracle JDBC によって文字データの処理および変換に使用されます。CHAR オブジェクトの主要属性の 1 つは、オブジェクト内の文字データのエンコーディングを定義するキャラクタ・セットです。CHAR オブジェクトの作成時に指定が必要です。SQL NCHAR データ型用に CHAR オブジェクトを作成するとき、サーバーのデータベース各国語キャラクタ・セットを使用する必要があります。SQL CHAR データ型用の場合は、データベース・キャラクタ・セットに応じて US7ASCII、WE8ISO8859P1、または UTF8 のいずれか 1 つが必要です。CHAR オブジェクトの詳細については、JDBC マニュアルを参照してください。SQL NCHAR データ型用に CHAR オブジェクトを作成する例を次に示します。

```
int oracleId = CharacterSet.AL16UTF16_CHARSET; // Character set ID for
AL16UTF16
…
CharacterSet mycharset = CharacterSet.make(oracleId);
String mystring = "\uFFA0";
...
CHAR mychar = new CHAR(mystring, mycharset)
;
```

c. JDBC アプリケーション用の CLOB から NCLOB への移行

同様に、ユーザーは NCLOB 列がターゲットであるデータ用にキャラクタ・セット構成属性を FORM_NCHAR にセットする必要があります。Oracle9*i* では、CLOB と NCLOB 間の暗黙的変換はサポートされていません。この属性をターゲット CLOB/NCLOB 列と同じキャラクタ・セット構成にセットしない場合、Oracle9*i* ではエラーが戻されます。なお Oracle Database 10g 以降は、CLOB と NCLOB 間の暗黙的変換がサポートされているため、エラーは戻されませんが、暗黙的変換はパフォーマンスに影響する可能性があります。JDBC アプリケーション内で、NCLOB 列に対するコーディングの例を次に示します。

次の例は、表定義、*clob_table (v2 VARCHAR2 (30), ncb NCLOB)*に基づいています。

```
Connection conn;
try {
    OraclePreparedStatement pstmt =
     (oracle.jdbc.OraclePreparedStatement) conn.prepareStatement
     ("insert into clob_table values (?, ?)");

    pstmt.setFormOfUse(2, OraclePreparedStatement.FORM_NCHAR);
```

```
 pstmt.setString (1, "one");
 pstmt.setString (2, "\uFF10 \uFF11 \uFF12 \uFF13 \uFF14");
 pstmt.execute ();

}

ResultSet rset = stmt.executeQuery ("select * from clob_table where v2
                                    = 'one' for update");
if (rset.next ())
{
 oracle.jdbc2.Clob clob = ((OracleResultSet)rset).getClob (2);
 show("getLength() = "+clob.length());

 String str = clob.getSubString(1,5);
 String data = "\uFF41 \uFF42 \uFF43 \uFF44 \uFF45";
 ((CLOB)clob).putString(1, data);
}
```

JDBC アプリケーションの移行を簡単にまとめると、JAVA 内部文字列エンコーディングは UTF-16 であるための前述の 2 つのシナリオのほか、Unicode 対応に必要なことはそれほどありません。変数が SQL NCHAR データ型にバインドされるとき、使用構成属性は FORM_NCHAR にセットする必要があります。CHAR オブジェクトの作成時も、データベース NCHAR キャラクタ・セットの調達が必要となります。

まとめ

このペーパーでは、データベースとアプリケーションの両方に関して、Unicode データ型をサポートする移行手順について説明しました。Unicode データ型を使用すれば、ユーザーは Unicode に段階的な移行ができます。Unicode キャラクタ・セットの選択基準、Unicode スキーマへの移行方法、およびユーザーが注意を必要とする重要事項などについては、データベースの移行で説明します。

SQL CHAR 型と SQL NCHAR 型の間の暗黙的変換は、アプリケーションを Unicode データ型に移行するための処理負荷を大幅に削減します。ただし、データ消失を回避し、移行済みアプリケーションの最高のパフォーマンスを達成するには、ユーザーはガイドラインに従う必要もあります。アプリケーション移行の説明に基づいて、SQL NCHAR 列がクライアントまたは中間層アプリケーションからアクセスされるとき、SQL NCHAR 型として SQL NCHAR 列をアクセスするには、文字構成属性のセット、あるいは%TYPE または%ROWTYPE 構文を使用したデータベース列の同期化により特別な設定が必要になります。これによってデータ消失が回避されます。最高のパフォーマンスを達成するためのキーは、SQL NCHAR 型と SQL CHAR 型の間の操作を可能な限り最小化することです。アプリケーション開発者は混合データ型の間の操作が必要なシナリオを回避する努力を必要とされます。操作が回避不可能なとき、ユーザーは前述のガイドラインに従い変換の数を削減できます。

多言語データベース/アプリケーションを目的とした Unicode **データ型への移行**
2003 **年** 8 **月**
著書: Gary Chen
寄稿者: Barry Trute

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

海外からのお問合せ窓口:
電話: +1.650.506.7000
ファックス: +1.650.506.7200
www.oracle.com

オラクル社は、インターネット上での活動を強化するソフトウェアを提供します。