IBM solidDB
IBM solidDB Universal Cache

バージョン 6.5

プログラマー・ガイド

Information Management software

SC88-8167-00
(英文原典：SC23-9870-00)

IBM solidDB
IBM solidDB Universal Cache

バージョン 6.5

プログラマー・ガイド

Information Management software

SC88-8167-00
(英文原典：SC23-9870-00)

注記

本書および本書で紹介する製品をご使用になる前に、 365 ページの『特記事項』に記載されている情報をお読みください。

本書は、バージョン 6 リリース 5 の IBM solidDB (製品番号 5724-V17) および IBM solidDB Universal Cache (製品番号 5724-W91)、および新しい版で明記されていない限り、以降のすべてのリリースおよびモディフィケーションに適用されます。

お客様の環境によっては、資料中の円記号がバックスラッシュと表示されたり、バックスラッシュが円記号と表示されたりする場合があります。

原典： SC23-9870-00
IBM solidDB
IBM solidDB Universal Cache
Version 6.5
Programmer Guide

発行： 日本アイ・ビー・エム株式会社

担当： トランスレーション・サービス・センター

第1刷 2009.10

目次

図

表

本書について

本書には、共有メモリー・アクセス (SMA)、リンク・ライブラリー・アクセス (LLA)、または HotStandby を使用する場合、または使用しない場合の、さまざまなアプリケーション・プログラミング・インターフェースを介した IBM® solidDB® の使用に関する情報が記載されています。

solidDB ODBC ドライバー、solidDB Light Client、および solidDB JDBC ドライバーは、クライアント・アプリケーションから solidDB へのアクセスに役立ちます。

- solidDB ODBC ドライバーは、Microsoft® ODBC 3.51 API 規格に準拠しています。
- solidDB Light Client は、solidDB ODBC API の軽量バージョンであり、クライアント・アプリケーションのフットプリントを非常に小さくする必要がある環境を対象としています。
- solidDB JDBC ドライバーは、JDBC 2.0 標準の solidDB インプリメンテーションです。

本書は、リレーショナル・データベースと SQL に関する一般的な知識があることを前提としています。また、solidDB をよく理解していることも前提としています。ODBC ドライバーを使用する場合、本書は C プログラミング言語の実用上の知識を前提としています。JDBC ドライバーを使用する場合、本書は Java™ プログラミング言語の実用上の知識を前提としています。

書体の規則

solidDB の資料では、以下の書体の規則を使用します。

表 1. 書体の規則

フォーマット	用途
データベース表	このフォントは、すべての通常テキストに使用します。
NOT NULL	このフォントの大文字は、SQL キーワードおよびマクロ名を示しています。
`solid.ini`	これらのフォントは、ファイル名とパス式を表しています。
`SET SYNC MASTER YES;` `COMMIT WORK;`	このフォントは、プログラム・コードとプログラム出力に使用します。SQL ステートメントの例にも、このフォントを使用します。
run.sh	このフォントは、サンプル・コマンド行に使用します。
TRIG_COUNT()	このフォントは、関数名に使用します。
java.sql.Connection	このフォントは、インターフェース名に使用します。

表 1. 書体の規則 (続き)

フォーマット	用途
LockHashSize	このフォントは、パラメーター名、関数引数、および Windows® レジストリー項目に使用します。
argument	このように強調されたワードは、ユーザーまたはアプリケーションが指定すべき情報を示しています。
管理者ガイド	このスタイルは、他の資料、または同じ資料内の他の章の参照に使用します。新しい用語や強調事項もこのように記述します。
ファイル・パス表示	特に明記していない場合、ファイル・パスは UNIX® フォーマットで示します。スラッシュ (/) 文字は、インストール・ルート・ディレクトリーを表します。
オペレーティング・システム	資料にオペレーティング・システムによる違いがある場合は、最初に UNIX フォーマットで記載します。UNIX フォーマットに続いて、小括弧内に Microsoft Windows フォーマットで記載します。その他のオペレーティング・システムについては、別途記載します。異なるオペレーティング・システムに対して、別の章を設ける場合があります。

構文表記法の規則

solidDB の資料では、以下の構文表記法の規則を使用します。

表 2. 構文表記法の規則

フォーマット	用途
`INSERT INTO table_name`	構文の記述には、このフォントを使用します。置き換え可能セクションには、この フォントを使用します。
`solid.ini`	このフォントは、ファイル名とパス式を表しています。
[ ]	大括弧は、オプション項目を示します。太字テキストの場合には、大括弧は構文に組み込む必要があります。
|	垂直バーは、構文行で、互いに排他的な選択項目を分離します。
{ }	中括弧は、構文行で互いに排他的な選択項目を区切ります。太字テキストの場合には、中括弧は構文に組み込む必要があります。
...	省略符号は、引数が複数回繰り返し可能なことを示します。
. . .	3 つのドットの列は、直前のコード行が継続することを示します。

1 IBM solidDB API 入門

このセクションでは、IBM solidDB データベースへのアクセスに使用するアプリケーション・プログラミング・インターフェースの概要を説明します。

アプリケーションは、これらのインターフェースを使用して、同時に複数のデータベース接続を確立し、同時に複数の SQL ステートメントを処理できます。

solidDB ODBC ドライバー

solidDB に組み込まれている 32 ビット・ネイティブ ODBC ドライバーは、Microsoft ODBC 3.51 API 標準に準拠しています。

ODBC 標準と solidDB インプリメンテーションの相異点については、このマニュアルの該当トピックを参照してください。

大部分のプラットフォームで、solidDB ODBC ドライバーは、solidDB Development Kit (SDK) に含まれています。

solidDB は、Unicode バージョンと ASCII バージョンの ODBC ドライバーを提供しています。Unicode バージョンについて詳しくは、207 ページの『6 章 Unicode の使用』を参照してください。

solidDB ODBC ドライバー関数の使用

すべてのプラットフォームのユーザーは、ODBC ドライバーがサポートする関数に、solidDB ODBC API でアクセスすることもできます。

solidDB ODBC API は、solidDB データベースのネイティブ・コール・レベル・インターフェース (CLI) です。ライブラリー (Microsoft Windows 上のダイナミック・リンク・ライブラリー (DLL)) の形式で配布されます。 solidDB ODBC API は、ANSI X3H2 SQL CLI 標準に準拠しています。

solidDB の ODBC API のインプリメンテーションは、堅固なデータベース・アプリケーションの作成に十分な、以下のような豊富な一連のデータベース・アクセス操作をサポートします。

- ハンドルの割り振りと割り振り解除
- 属性の取得と設定
- データベース接続のオープンとクローズ
- 記述子へのアクセス
- SQL ステートメントの実行
- スキーマ・メタデータへのアクセス
- トランザクションの制御
- 診断情報へのアクセス

アプリケーションの要求に応じて、solidDB ODBC ドライバーは、各 SQL ステートメントを自動的にコミットすることも、また明示的なコミットまたはロールバック要求を待ち合わせることもできます。ドライバーは、コミット操作またはロールバック操作を実行するときには、接続に関連付けられたすべてのステートメント要求をリセットします。

ODBC API での基本的なアプリケーション手順

クライアント・データベース・アプリケーションは、solidDB ODBC API を直接 (または ODBC ドライバー・マネージャーを通して) 呼び出し、データベースとのすべての対話を実行します。例えば、レコードの挿入、削除、更新、または選択を行うには、ODBC API で一連の関数呼び出しを行います。

ODBC API を使用するアプリケーションは、以下のタスクを実行します。

1. アプリケーションは、メモリーを割り振り、ハンドルを作成し、データベースへの接続を確立します。
   a. アプリケーションは、環境ハンドル (henv) および接続ハンドル (hdbc) 用にメモリーを割り振ります。どちらも、データベース接続の確立に必要です。

      アプリケーションは、1 つ以上のデータ・ソースに関して複数の接続を要求できます。各接続は、別個のトランザクション・スペースとみなされます。言い換えると、1 つの接続での COMMIT や ROLLBACK によって、他の接続を通して実行されているステートメントがコミットまたはロールバックされることはありません。
   b. SQLConnect() 呼び出しは、データベース接続を確立し、サーバー名 (接続ストリングまたはデータ・ソース名)、ユーザー ID、およびパスワードを指定します。
   c. 次に、アプリケーションは、ステートメント・ハンドル用にメモリーを割り振ります。
2. アプリケーションは、ステートメントを実行します。そのためには、一連の関数呼び出しを行う必要があります。
   a. アプリケーションは、SQLExecDirect() を呼び出して SQL ステートメントを準備および実行するか、または SQLPrepare() と SQLExecute() を呼び出してステートメントを複数回実行します。
   b. ステートメントが SELECT の場合には、アプリケーション内の変数に結果列をバインドして、戻されたデータをアプリケーションが参照できるようにする必要があります。SQLBindCol() 関数は、アプリケーションの変数を結果セット内の列にバインドします。行は、SQLFetch() を使用して繰り返しフェッチすることができます。 SELECT ステートメントは、結果セットの処理が終了するとすぐにコミットする必要があります。

      ステートメントが UPDATE、DELETE、または INSERT の場合には、アプリケーションは、実行が成功したか検査し、SQLEndTran() を呼び出してトランザクションをコミットする必要があります。
3. 最後に、アプリケーションは接続を閉じ、すべてのハンドルを解放します。
   a. アプリケーションは、ステートメント・ハンドルを解放します。
   b. アプリケーションは、接続を閉じます。

c. アプリケーションは、接続ハンドルと環境ハンドル (hdbc と henv) を解放します。

ステップ 2 (SQL ステートメントの実行) は、実行する必要のある SQL ステートメントの数に応じて、繰り返し実行できることに注意してください。

これらの API 呼び出しの使用方法について詳しくは、11 ページの『2 章 solidDB ODBC API の使用』を参照してください。

接続ストリングのフォーマット

ODBC アプリケーションおよび Light Client アプリケーションで使用する接続ストリングは、以下に示す共通フォーマットに従います。

構成ファイル `solid.ini` 内の listen パラメーターにも、同じフォーマットが適用されます。

`protocol_name [options] [server_name] [port_number]`

ここで、options には、以下の項目を任意の数だけ指定できます。

表 3. 接続ストリング・オプション

オプション	意味
-z	この接続では、データ圧縮が使用可能です。
-c *milliseconds*	ログイン・タイムアウトを指定します (デフォルトは、オペレーティング・システムに固有です)。指定された時間が経過すると、ログイン要求が失敗します。注: TCP プロトコルにのみ適用されます。
-r *milliseconds*	接続 (または読み取り) タイムアウトを指定します (デフォルトは 60 秒です)。指定された時間内に応答が受信されないと、ネットワーク要求が失敗します。0 の値を指定すると、タイムアウトは無限に設定されます。注: TCP プロトコルにのみ適用されます。

例

```
tcp localhost 1315
tcp 1315
tcp -z -c1000 1315
nmpipe host22 SOLIDDB
```

クライアント・サイド構成ファイル

solidDB は、クライアント構成情報をクライアント・サイドの `solid.ini` ファイルから取得します。クライアント・サイド構成ファイルは、ODBC ドライバーが使用され、アプリケーションの作業ディレクトリーにファイルを配置する必要がある場合に使用されます。

重要: ほとんどの場合、solidDB 用のプログラミングでは、solidDB サーバー・サイド・パラメーターのみが使用されます。しかし、クライアント・サイド・パラメーターを使用することが必要な場合があります。例えば、データ・ソースを定義しないが、クライアント・サイド構成ファイルの接続ストリングからデータ・ソースを取得するアプリケーションを作成したい場合などです。

注: solidDB の資料では、solid.ini ファイルについて言及するときは、通常、サーバー・サイドの solid.ini ファイルを指します。

solidDB は、始動時に構成ファイル solid.ini を開こうとします。このファイルが存在しない場合、solidDB は、パラメーターにファクトリー値を使用します。solid.ini ファイルが存在する場合でも、その中の特定のパラメーターに値が設定されていない場合、solidDB は、そのパラメーターにファクトリー値を使用します。ファクトリー値は、使用するオペレーティング・システムに依存する場合があります。

デフォルトでは、クライアントは現行作業ディレクトリーで solid.ini ファイルを検索しますが、通常、これはクライアントを始動したディレクトリーです。solidDB は、ファイルの検索時に以下の優先順位に従います (上から下)。

- SOLIDDIR 環境変数によって指定された場所 (この環境変数が設定されている場合)
- 現行作業ディレクトリー

クライアント・サイド・パラメーター

このセクションでは、最も重要な solidDB クライアント・サイド・パラメーターについて説明します。

- Com.Connect

  [Com] セクション内の Connect パラメーターは、クライアントがサーバーと通信する際に、クライアントが接続するデフォルトのネットワーク名 (接続ストリング) を定義します。当然のことながら、クライアントは、サーバーが listen するのと同じネットワーク名で対話しなければならないため、クライアント上の Connect パラメーターの値は、サーバー上の Listen パラメーターの値と一致しなければなりません。

  同じフォーマットの接続ストリングが、すべての listen 構成パラメーターおよび ODBC アプリケーションと Light Client アプリケーションで使用される接続ストリングに適用されます。

- Com.Trace

  Trace パラメーターをデフォルト設定の No から Yes に変更すると、solidDB は、確立したネットワーク接続に関してネットワーク・メッセージのトレース情報のロギングを開始し、デフォルトのトレース・ファイルまたは TraceFile パラメーターで指定されたファイルに記録します。

- Com.TraceFile

  Trace パラメーターを Yes に設定すると、ネットワーク・メッセージに関するトレース情報が TraceFile パラメーターで指定されたファイルに書き込まれます。ファイル名の指定がない場合、サーバーはデフォルト値 soltrace.out を使用しますが、このファイルは、トレースがどちら側で開始されたかにより、サーバーまたはクライアントの現行作業ディレクトリーに書き込まれます。

ODBC の非標準動作

このセクションでは、solidDB ODBC ドライバーの非標準動作と制限事項について説明します。

エラー情報

クライアントによって設定されたバージョンに関係なく、ドライバーは ODBC 3.0 仕様に基づいたエラー情報を返します。

SQL_NULL_DATA をパラメーター長として使用した SQLPutData でのエラー

1 つ以上のデータ項目を挿入または更新しようとする場合、いずれかの項目が SQL_NULL_DATA を長さの指定子として持っていると、データは挿入されません。列値は NULL になります。

SQLAllocHandle が不完全なエラー情報を返す場合

SQLAllocHandle を呼び出すとき、ハンドル・タイプが無効である場合、例えば、以下のような場合、

```
SQLAllocHandle(-5, hdbc, &hstmt);
```

この関数は SQL_ERROR を返しますが、エラー状態「HY092」やメッセージ「Invalid Attribute/Option Identifier」を返しません。

MSAccess - 表と特定の列型とのリンク

表をデータ型 WCHAR、WVARCHAR、および LONG WVARCHAR とリンクした後、ユーザーが特定のレコードを挿入し、別のレコードを挿入/更新/削除すると、ドライバーは前の新規に追加/更新されたレコードについて、「#deleted」を表示します。

ADO - OpenSchema メソッド

以下の OpenSchema メソッドは、ADO によってサポートされません。

- adSchemaCatalogs
- adSchemaColumnPrivileges
- adSchemaConstraintColumnUsage
- adSchemaConstraintTableUsage
- adSchemaTableConstraint
- adSchemaForeignKeys
- adSchemaTablePrivileges
- adSchemaViews
- adSchemaViewTableUsage

上記の OpenSchema メソッドは、どの ODBC ドライバーを使用しても、ADO によってサポートされません。これは Microsoft OLE DB Provider for ODBC の制限事項です。これは、solidDB ODBC ドライバーに限ったことではありません。

solidDB Light Client

solidDB Light Client では、C、または C 関数呼び出し規約に準拠する任意のツールを使用してフットプリントの小さいアプリケーションを開発できます。

solidDB Light Client は、20 個の関数から成る ODBC API のサブセットであり、solidDB データベースのデータにアクセスするアプリケーション開発者に完全な SQL 機能を提供します。データベース接続の制御、SQL ステートメントの実行、結果セットのリトリーブ、トランザクションのコミット、およびその他のデータ管理機能のための関数を提供します。

`/samples/lightclient` サブディレクトリーに、solidDB Light Client API を使用する C プログラムのサンプルがあります。

詳しくは、 57 ページの『3 章 solidDB Light Client の使用』を参照してください。

注: solidDB Light Client は、すべてのプラットフォームで使用可能なわけではありません。

solidDB JDBC ドライバー

JDBC 2.0 ドライバーは、JDBC 2.0 をサポートします。

solidDB JDBC ドライバーの場合、Java Development Kit (JDK) 1.4.2 以降がサポートされています。

solidDB JDBC ドライバーを使用すると、JDBC を使用してデータベースにアクセスする Java ツールでアプリケーションを開発できます。JDK 1.2 用のコア API である JDBC API は、データベース接続、SQL ステートメント、結果セット、データベース・メタデータなどを表現する Java クラスを定義します。これで、SQL ステートメントを発行し、結果を処理できます。JDBC は、Java でデータベースにアクセスするための基本 API です。

JDBC を使用するためには、solidDB JDBC ドライバーをインストールする必要があります。 JDBC ドライバーの使用法は、ご使用の Java 開発環境によって異なります。solidDB JDBC ドライバーの使用法に関する説明と例が、`/jdbc` サブディレクトリーおよび 93 ページの『4 章 solidDB JDBC ドライバーの使用』に記載されています。

クライアント・アプリケーションの作成

このセクションでは、 solidDB で動作するクライアント・アプリケーションの作成方法の概要を説明します。このセクションに記載された情報は、主に ODBC ドライバーまたは Light Client ドライバーを使用する C 言語プログラムに適用されます。

クライアントとは

クライアント・アプリケーション、または単に「クライアント」とは、サーバーに要求 (SQL 照会) をサブミットし、サーバーから返された結果を取得するプログラムです。

クライアント・プログラムは、サーバー・プログラムから分離されています。また、多くの場合、クライアントは、別のコンピューター上で実行されています。共有メモリー・アクセスまたはリンク・ライブラリー・アクセスを使用すると、クライアントのコードをサーバーのコードに直接リンクし、両方を単一のプロセスとして実行することができます。詳しくは、「*IBM solidDB 共有メモリー・アクセスおよびリンク・ライブラリー・アクセス・ユーザー・ガイド*」を参照してください。

クライアントは別個のプログラムなので、サーバー内の関数を直接呼び出すことはできません。その代わりに、通信プロトコル (TCP/IP、名前付きパイプ、共有メモリーなど) を使用してサーバーと通信する必要があります。プラットフォームが異なると、サポートされるプロトコルも異なります。プラットフォームによっては、使用するアプリケーションに特定のライブラリー・ファイル (特定のプロトコルをサポート) をリンクして、アプリケーションがサーバーと通信できるようにする必要があります。

照会をサーバーに渡す方法

照会は、SQL プログラミング言語を使用して作成されます。

サーバーとクライアント間でデータを交換する 1 つの方法は、単にリテラル・ストリングを相互に受け渡すことです。クライアントは、サーバーに以下のようなストリングを送信するとします。

```
SELECT name FROM employees WHERE id = 12;
```

そして、サーバーは、以下のようなストリングを送り返すとします。

```
"Smith, Jane"
```

しかし、実際には、通信は一般に ODBC ドライバーや JDBC ドライバーなどの「ドライバー」を介して行われます。「ODBC」は「Open DataBase Connectivity」を意味しており、ベンダー間でのデータベース・アクセスの整合性を高めるために Microsoft が設計した API (アプリケーション・プログラミング・インターフェース) です。クライアント・プログラムは、ODBC 規則に従っている場合、それらと同じ規則に従うどのデータベース・サーバーとも対話することができます。ほとんどの主要データベース・ベンダーは、少なくともある程度は ODBC をサポートしています。ODBC 標準は、一般に、C プログラミング言語で作成されたプログラムで使用されます。

「JDBC」は、「Java DataBase Connectivity」を意味しています。多くの部分で ODBC 標準に基づいており、当然のことながら、基本的には「Java プログラム用 ODBC」です。JDBC に関する情報は、メイン Java Web サイトから入手できます。

http://java.sun.com/

特定のデータ値 (例えば、″Smith, Jane″) をサーバーに渡すのに、2 つの主要な方法があります。最初の方法では、単に値をリテラルとして照会に埋め込みます。この方法は、以下のような SQL ステートメントで既に説明しました。

```
INSERT INTO employees (id, name) VALUES (12, 'Smith, Jane');
```

この方法は、単一のステートメントを実行したい場合に効果的です。しかし、同じ基本ステートメントを異なる値で実行したい場合があります。例えば、500 名の従業員のデータを挿入したい場合、以下のように、別々のステートメントを 500 個作成するのは望ましくありません。

```
INSERT INTO employees (id, name) VALUES (12, 'Smith, Jane');
INSERT INTO employees (id, name) VALUES (13, 'Jones, Sally');
...
```

その代わりに、単一の「汎用」ステートメントを作成して、そのステートメントの特定の値を渡す方が望ましい場合があります。例えば、以下のステートメントを作成します。

```
INSERT INTO employees (id, name) VALUES (?, ?);
```

そして、疑問符 (?) を特定のデータ値で置き換えます。このようにすれば、ループ内で 500 個の INSERT ステートメントを簡単に実行でき、従業員ごとにユニークな INSERT ステートメントを作成する必要はありません。パラメーターを使用することにより、ステートメントを実行するたびに異なる値を指定できます。パラメーターにより、クライアントとサーバーが交換する値を格納するために、クライアント・プログラムと ODBC ドライバーが使用する変数を指定できます。基本的には、ステートメント内に疑問符 (?) を指定した各位置でパラメーターを渡します。

パラメーターを使用してデータ値を交換したいもう 1 つの状況として、ストリング・リテラルとして表現するのが困難なデータを処理する場合があります。例えば、「American Pie」という曲のデジタル化コピーをデータベースに挿入したいが、そのデジタル化データを表す一連の 16 進数を含むリテラルで SQL ステートメントを作成したくはない場合、デジタル化データを配列に格納し、ODBC ドライバーにその配列の位置を通知します。

SQL ステートメントでパラメーターを使用するには、複数のステップから成る処理を実行します。以下に、データを挿入する際の処理を説明します。この処理は、データをリトリーブする際の処理にある程度似ています。

1. SQL ステートメントを「準備」します。「準備」段階で、サーバーがステートメントを分析し、(他の処理も行いますが特に) パラメーターが何個になるか確認します。パラメーターの数と意味は、SQL ステートメントに組み込まれた疑問符 (?) によって示されます。
2. ODBC ドライバーに、パラメーターとして使用される変数を通知します。(どの変数がどの列や値に関連付けられるかを ODBC ドライバーに通知することを、パラメーターを「バインド」すると言います。)
3. パラメーターに値を設定します (つまり、変数の値を設定します)。
4. 準備済みステートメントを「実行」します。

実行段階で、ODBC ドライバーは、パラメーターに格納された値を読み取り、あらかじめ準備されたステートメントで使用するために、これらの値をサーバーに渡します。

返された結果を取得する処理も同様であり、次のセクションで説明します。

クライアントに結果を返す方法

照会の結果は、0 個以上の一連の行です。ODBC ドライバー (または JDBC ドライバー) を使用している場合、該当する ODBC (または JDBC) 関数を使用して各行をリトリーブします。

一般的には、以下の手順を実行します。

1. SQL ステートメントを「準備」します。「準備」段階で、サーバーがステートメントを分析し、(他の処理も行いますが特に) パラメーターが何個になるか確認します。パラメーターの数と意味は、SQL ステートメントに組み込まれた疑問符 (?) によって示されます。
2. ODBC ドライバーに、パラメーターとして使用される変数を通知します。(どの変数がどの列や値に関連付けられるかを ODBC ドライバーに通知することを、パラメーターを「バインド」すると言います。)
3. 準備済みステートメントを「実行」します。これは、サーバーに対して照会を実行し、結果セットを収集するよう指示します。ただし、結果セットはすぐにクライアントに渡されるわけではないことに注意してください。
4. 結果セットの次の行を「フェッチ」します。「フェッチ」を行うと、結果セットから結果を 1 行リトリーブし、アプリケーションとの共有のためにあらかじめ ODBC ドライバー用に定義したパラメーターにその行の値を格納するように、サーバーおよび ODBC ドライバーに指示することになります。

当然のことながら、通常はループを実行することになり、一度に 1 行をフェッチして、各フェッチ後にパラメーターからデータを読み取ります。

ODBC ドライバーまたは Light Client ライブラリーの使用

これらのドライバーとライブラリーは、クライアント・アプリケーション・プログラムにリンクする必要があります。

静的ライブラリーと動的ライブラリー

次に、これらのライブラリー内に定義された関数を呼び出すことができます。ライブラリー名について詳しくは、solidDB パッケージの SDK Notes を参照してください。

ライブラリー・ファイルには静的なものがあります。つまり、これらのファイルは、コンパイルとリンク操作を行ったときにクライアント・アプリケーションの実行可能プログラムにリンクされます。ライブラリー・ファイルには動的なものもあります。これらは実行可能プログラムとは別に格納され、プログラムの実行時にメモリーにロードされます。

静的ライブラリーの利点は、アプリケーションが必要なものをほぼ完備していることです。アプリケーションを顧客に配布する場合、顧客は、アプリケーションをインストールすれば、そのほかに別の共有ライブラリーをインストールする必要がありません。

動的ライブラリーの利点は、複数のクライアントがそのライブラリーを使用する場合、多くのシステムで、必要なディスク・スペースが少なくなることです (また、プラットフォームによっては、必要なメモリー・スペースも少なくなります)。例え

ば、5 MB の静的ライブラリーにそれぞれリンクする 2 つのクライアント・アプリケーションが存在する場合、その静的ライブラリーを格納するのに 5 MB のディスク・スペースが必要なだけでなく、アプリケーションにリンクされる両方のライブラリーのコピーを格納するために 10 MB の追加ディスク・スペースが必要になります。しかし、2 つのクライアント・アプリケーションを動的ライブラリーにリンクすると、そのライブラリーの追加コピーは必要ありません。各アプリケーションは、独自のコピーを保持しません。

多くのライブラリーに関して、solidDB は、一部またはすべてのプラットフォーム上で静的バージョンと動的バージョンを提供しています。

さらに、Microsoft Windows 上で、solidDB は、場合によってはインポート・ライブラリーを提供しています。各インポート・ライブラリーは、対応するダイナミック・リンク・ライブラリーに関連付けられています。ご使用のアプリケーションは、インポート・ライブラリーにリンクされます。アプリケーションを実際にロードし、実行するときに、オペレーティング・システムが、対応するダイナミック・リンク・ライブラリーをロードします。

ステートメント・キャッシュ

照会の処理は、組み込み「ステートメント・キャッシュ」によりさらに最適化されます。

ステートメント・キャッシュは、これまでに準備されたいくつかの SQL ステートメントを格納する内部メモリーです。1 つのセッションでキャッシュされるステートメント数は、クライアント・サイドの `solid.ini` 構成パラメーター Client.StatementCache を使用して設定できます。デフォルト値は 6 です。

ステートメント・キャッシュは、準備済みステートメントがキャッシュ内に存在する場合には、準備段階を省略するようにして動作します。接続が閉じると、ステートメント・キャッシュはパージされます。

JDBC では、ステートメント・キャッシュ・サイズは、非標準のドライバー・プロパティーを使用して動的に設定できます。詳しくは、108 ページの『非標準の接続プロパティー』を参照してください。

2 solidDB ODBC API の使用

このセクションでは、ODBC API を使用するアプリケーションの開発に関して、solidDB に固有の情報と使用例を示します。

一般に、solidDB は、Microsoft ODBC 3.51 標準に準拠しています。 solidDB ODBC API は、Microsoft が提供する関数プロトタイプに基づいて定義されています。この章では、solidDB に固有の使用法が適用される領域と、オプション、データ型、および関数のサポートが異なる領域について詳細に説明します。

注: この「solidDB プログラマー・ガイド」には、ODBC API 解説の全体は含まれていません。ODBC API を使用したアプリケーションの開発について詳しくは、「*Microsoft Data Access SDK オンライン ODBC プログラマーズ リファレンス (Microsoft Data Access SDK Online ODBC Programmer's Reference)*」を参照してください。

solidDB は、Unicode 用および ASCII 用として、2 つのバージョンの ODBC ドライバーを提供しています。Unicode バージョンは ASCII バージョンのスーパーセットであり、Unicode 文字セットと ASCII 文字セットで使用できます。

solidDB ODBC ドライバーのインストール

solidDB インストーラーは、2 つの ODBC ドライバー (1 つは Unicode 用、もう 1 つは ASCII 用) をインストールします。Unicode バージョンは ASCII バージョンのスーパーセットであり、Unicode 文字セットと ASCII 文字セットで使用できます。

Windows

solidDB インストーラーは、ODBC ドライバーと以下のシステム・データ・ソース名 (DSN) を自動的にインストールします。自分自身のユーザー DSN を追加することもできます。

- Windows 32 ビット・オペレーティング・システムの場合
  - IBM solidDB 6.5 32 ビット – ANSI
  - IBM solidDB 6.5 32 ビット – Unicode
- Windows 64 ビット・オペレーティング・システムの場合
  - IBM solidDB 6.5 64 ビット – ANSI
  - IBM solidDB 6.5 64 ビット – Unicode

Linux® および UNIX

Linux および UNIX 環境では、ODBC ドライバー・ライブラリー・ファイルは以下のディレクトリーにインストールされます。

- `<solidDBインストール・ディレクトリー>/bin/`: 動的ライブラリー・ファイル

  - sac<プラットフォーム><バージョン>.so または sac<プラットフォーム><バージョン>.so – ANSI
  - soc<プラットフォーム><バージョン>.so または soc<プラットフォーム><バージョン>.so – Unicode
- <solidDB インストール・ディレクトリー>/lib/: 静的ライブラリー・ファイル
  - solidodbca.so または solidodbca.so – ANSI
  - solidodbcu.so または solidodbcu.so – Unicode

オペレーティング・システムに応じて、ファイル拡張子は .sa または .so のどちらかになります。

solidDB ODBC ドライバー 3.51 機能のサポート

このセクションでは、以前のバージョン (1.0、2.0、および 3.0) の solidDB ODBC ドライバーから solidDB ODBC ドライバー 3.51 にマイグレーションしたユーザーを対象にした、ODBC ドライバー 3.51 の機能のサポートを詳細に説明します。

このドライバーでは、以下の機能がサポートされます。

- 記述子の完全なサポート
- すべてのカタログ API のサポート
- Unicode のサポート
- マルチスレッドのサポート
- ADO/DAO/RDO/OLE DB のサポート
- MS Access と MS Query を通したデータ・アクセス
- ブロック・カーソルのサポート

Microsoft Windows 上での使用法の概要

Microsoft Windows オペレーティング・システムでは、solidDB ODBC ライブラリーは .DLL ファイルとして提供されます。

ファイルは、Unicode バージョンと ASCII バージョンで、それぞれ socw32VV.dll と sacw32VV.dll という名前になります (ここで、「VV」はバージョン番号を示しています)。例えば、バージョン 4.1 の Unicode ODBC ドライバーは socw3241.dll という名前になります。これらの .DLL ファイルのいずれかにある関数を呼び出すには、solidDB インポート・ライブラリー・ファイルにリンクする必要があります。Microsoft Windows 上の solidDB では、このインポート・ライブラリー・ファイルは solidimpodbcu.lib (Unicode) または solidimpodbca.lib (ASCII) という名前になります。このインポート・ライブラリー・ファイルには、対応する solidDB ODBC DLL (例えば socw3241.dll) へのエントリー・ポイントが含まれます。

注: ライブラリー・ファイルは、Microsoft C++ で作成されています。その他の開発ツールキット・メーカーのリンカーでは、異なるライブラリー・ファイル・フォーマットが想定される場合があります。このような場合には、開発ツールキットのインポート・ライブラリー・ユーティリティーを使用して、ご使用のリンカーと互換性のあるライブラリー・ファイルを構築する必要があります。これは、「Light

Client」ライブラリーでのみ必要です。他のライブラリー・ファイルについては、Microsoft 以外の大部分のリンカーと互換性のあるインポート・ライブラリーが solidDB に含まれています。

solidDB クライアント DLL (Solid® ODBC ドライバー・ファイル) の使用法に関する説明

solidDB ODBC ドライバーを使用するアプリケーション・プログラムを構築するには、以下の 2 つの方法から選択できます。

1. Microsoft ODBC ドライバー・マネージャーを使用する。

   すべてのクライアント・ワークステーションに Microsoft ODBC ソフトウェアをインストールし、solidDB ODBC ドライバーを使用してデータ・ソースを定義する必要があります。ドライバー・マネージャーを使用すると、solidDB ODBC ドライバーを使用できるアプリケーションであれば、さらに、他のどの ODBC 準拠エンジンとも連携します。

2. solidDB ODBC ドライバーを直接使用する。

   接続は、Microsoft ODBC ドライバー・マネージャーを使用せずに、サーバー・プロセスに対して直接開かれます。これにより、通常、solidDB の組み込みデプロイメントがより簡単になります。ただし、アプリケーションが使用できる関数は、solidDB ライブラリー (つまり、Solid ODBC ドライバー) によって提供された関数に限られます。アプリケーションは、Microsoft ODBC ドライバー・マネージャーまたは Microsoft Cursor Library でインプリメントされた ODBC 関数を使用できません。

solidDB は、Microsoft ODBC ドライバー・マネージャーで使用できる、またはそれなしで使用できるサンプル・プログラムをいくつか提供しています。これらのサンプルは、solidDB Development Kit (SDK) をインストールしたときに作成された「samples」ディレクトリーのサブディレクトリーに入っています。以下に、選択可能な方法の両方について、提供されているサンプルを作成および実行する方法を簡単に説明します。

1.

   ODBC ドライバー・マネージャーを使用するサンプルを作成する。

   a. 新しいアプリケーション・プロジェクトを作成します。
   b. プロジェクトに C ソース・ファイル (例えば `sqled.c` や `embed.c`) を追加します。
   c. solidDB SQL API ヘッダーをコンパイラーに可視になるように設定します。
   d. コンパイラー用に SS_WINDOWS を定義します。
   e. コンパイルおよびリンクを行います。
   f. solidDB ODBC ドライバーをインストールしたことを確認します。また、使用する接続ストリングが ODBC データ・ソース名として定義されていることを確認します。
   g. 実行して、listen 状態の solidDB サーバーに接続します。

2. solidDB ODBC ライブラリーを直接使用するサンプルを作成する。

以下に、ODBC ドライバー・マネージャーの構成に対して必要な変更をリストします。

a. プロジェクトに、solidDB ODBC ドライバー・ライブラリー・ファイル (solidimpodbcu.lib) を追加します。
b. デフォルトのライブラリー・リストから ODBC ドライバー・マネージャー・ライブラリー ODBC*.LIB を除去します。
c. コンパイルおよびリンクを行います。
d. これで、ODBC ドライバー・マネージャーをバイパスして、データ・ソースに接続できます。SQL API DLL socw32<VV>.dll (ここで、「VV」はバージョン番号を示す) および solidDB 通信 DLL が使用可能であることを確認します。データ・ソースは、solid.ini または ODBC 管理ウィンドウで定義できます。
e. クライアントを実行して、listen 状態の solidDB サーバーに接続します。

unixODBC を使用した solidDB ODBC ドライバーの使用

unixODBC は、UNIX 型環境に対応した ODBC ドライバー・マネージャーです。アプリケーションを直接 solidDB ODBC ドライバーにリンクしないで、unixODBC DriverManager を使用することができます。

unixODBC および unixODBC DriverManager について詳しくは、http://www.unixodbc.org/ にアクセスしてください。

構成ファイル

unixODBC DriverManager は、以下の 2 つの構成ファイルの指定に従って、適切なデータ・ソース・ドライバーをロードします。

- odbc.ini または .odbc.ini: データ・ソースおよび実際の ODBC ドライバーの論理名を指定します。
- odbcinst.ini または .odbcinst.ini: 論理ドライバー名を、ファイル・システム上のその物理的位置に関連付けます。

上記のファイルに加えて、実際の ODBC ドライバーには、異なる構造を持つクライアント・サイドの solid.ini 構成ファイルが必要です。このファイルでは、論理データ・ソース名が、物理接続ストリングに関連付けられます。

odbc.ini 構成ファイルの構文

odbc.ini ファイルまたは .odbc.ini ファイルは、データ・ソースごとに、少なくとも以下の 2 つの項目を示す必要があります。

- 大括弧で囲まれたデータ・ソースの論理名。例えば、[my_solid]。
- 構文 Driver=<driver name> を使用することによって使用される実際の ODBC ドライバーの論理名。例えば、Driver = solid_odbc。

  構文 Description=My first Solid を使用して、さらに記述を追加することができます。

  追加情報は、すべて無視されます。

odbcinst.ini 構成ファイルの構文

ODBC ドライバーの論理名と物理的位置は、以下のように、odbcinst.ini ファイルまたは .odbcinst.ini ファイルに指定される必要があります。

- [<the logical name of the driver>]、例えば、[solid_odbc]
- Driver = <absolute path to the driver>、例えば、Driver = /home/jsmith/sacl2x64.so

クライアント・サイドの solid.ini 構成ファイルの構文

クライアント・サイドの solid.ini ファイルでは、論理データ・ソース名は、それらの物理接続ストリングに、以下のようにして関連付けられる必要があります。

- [Data Sources]
- <the logical data source name> = <physical connect string> 、例えば、my_solid=tcp my_machine 1964

構成ファイルの場所

システム・レベルのデータ・ソースは、/usr/local/etc/odbc.ini に指定されます。

ユーザー・レベルの DSN は、~/.odbc.ini に指定されます。

同様に、システム・レベルのドライバーは、/usr/local/etc/odbcinst.ini に指定され、ユーザー・レベルのドライバーは、~/.odbcinst.ini に指定されます。

クライアント・サイドの solid.ini ファイルは、SOLIDDIR 環境変数によって設定されたディレクトリーか、現在の作業ディレクトリーに置くことができます。

ドライバーのリンク

solidDB ODBC ドライバーではなく、unixODBC にリンクするには、以下のようにします。

1. unixODBC ドライバーを選択した場所にコピーします。
2. solidDB ODBC ドライバーのライブラリー・ファイルを、unixODBC ライブラリー・ファイルに置き換えます。

   例えば、以下のようにします。

   直接リンク: LDFLAGS = $(SOLID_LIB)/linux/sacl2x65.so

   unixODBC ドライバー・マネージャー: LDFLAGS = $(SOLID_LIB)/linux/libodbc.so

構成ファイルの例

~/.odbc.ini :

```
[foo]
Description         = Testing Solid
Driver              = solid_driver_65
```

~/.odbcinst.ini :

```
    [solid_driver_63]
    Description             = The newest ODBC driver
    Driver          = /home/jsmith/Solid6.50.0009/bin/sacl2x64.so

$SOLIDDIR/solid.ini    :

    [Data Sources]
    foo = tcp 1964
```

関数の呼び出し

このセクションでは、プログラムが ODBC ドライバーの関数を呼び出す方法を説明します。

ヘッダー・ファイルと関数プロトタイプ

プログラムで ODBC ドライバーの関数を呼び出す場合、ODBC ヘッダー・ファイルをインクルードする必要があります。これらのファイルには、ODBC 関数、および ODBC 関数で使用されるデータ型と定数が定義されています。ヘッダー・ファイルは、solidDB に固有ではなく、Microsoft が提供する標準ヘッダー・ファイルです。solidDB ODBC ドライバーは (他の ODBC ドライバーと同様に)、これらのヘッダー・ファイルに指定された関数をインプリメントします。

ASCII と unicode

ODBC ドライバーには、ASCII と Unicode の 2 つの「フレーバー」があります。ASCII フレーバーは、ASCII 文字セットのみをサポートします。Unicode フレーバーは、Unicode 文字セットと ASCII 文字セットの両方をサポートします。

プログラムで、ASCII フレーバーの ODBC 関数のみを呼び出す場合には、以下のヘッダー・ファイルをインクルードする必要があります。

- SQL.H
- SQLEXT.H

プログラムで、Unicode の ODBC 関数を呼び出す場合には、以下のヘッダー・ファイルをインクルードする必要があります。

- SQLUCODE.H
- WCHAR.H。このファイルは、Microsoft Visual C++ (または Developer Studio) で提供されています。

プログラムで、ASCII フレーバーと Unicode フレーバーの両方の ODBC 関数を呼び出す場合には、Unicode のヘッダー・ファイルのみをインクルードしてください。(Unicode バージョンのヘッダー・ファイルには、ASCII 関数の定義も含まれています。言い換えると、Unicode ヘッダーは、ASCII ヘッダーのスーパーセットです。)

ドライバー、API、および SQL 適合レベルについて詳しくは、solidDB の Web サイト (http://www.ibm.com/software/data/soliddb) で、「Microsoft ODBC API 仕様書 (Microsoft ODBC API Specification)」(Part I の PDF ファイル) の『ODBC の概要 (Introduction to ODBC)』を参照してください。

ODBC ドライバー・マネージャーの使用

アプリケーションは、solidDB ODBC ドライバーに直接リンクするか、または ODBC ドライバー・マネージャーにリンクします。このセクションでは、ODBC ドライバー・マネージャーの使用について説明します。

Microsoft Windows では、solidDB に接続するアプリケーションが OLE DB または ADO API を使用する場合、またはドライバー・マネージャーを必要とする Microsoft Access、FoxPro、または Crystal Reports などのデータベース・ツールを使用する場合、ドライバー・マネージャーが必要になります。その他のほとんどの場合、ドライバー・マネージャーにリンクする代わりに、ODBC ドライバーに直接リンクすることができます。

Microsoft Windows プラットフォームでは、Microsoft がドライバー・マネージャーを提供しており、ドライバー・マネージャー・インポート・ライブラリー (ODBC32.LIB) にリンクしてドライバー・マネージャーにアクセスできます。他のプラットフォームでは、別のベンダーのドライバー・マネージャーにリンクできます。例えば、Linux 上および Solaris 8 上では、Merant のドライバー・マネージャーまたは iODBC のドライバー・マネージャーを使用できます。

アプリケーションが ODBC 関数を呼び出すときの基本的なアプリケーション手順、および ODBC 関数の呼び出しについて詳しくは、solidDB の Web サイトで、「Microsoft ODBC API 仕様書 (Microsoft ODBC API Specification)」(Part I の PDF ファイル) の『ODBC の概要 (Introduction to ODBC)』を参照してください。

データ型

277 ページの『付録 E. データ型』 では、solidDB がサポートする SQL データ型に関する情報が提供されます。Microsoft からのヘッダー・ファイルには、クライアント・プログラムで使用する C 言語のデータ型に関する情報が含まれています。アプリケーション・プログラムとデータベース・サーバー間でデータを転送するには、適切な型を使用する必要があります。例えば、ほとんどの 32 ビット・プラットフォームでは、C 言語の「int」データ型は、SQL の「INT」データ型に対応します。C 言語の「float」データ型は、SQL の「REAL」データ型に対応します。(C の「float」は、SQL の「FLOAT」に対応しないことに注意してください。) ODBC 呼び出しを介したデータの転送に使用する C 言語のデータ型について詳しくは、該当するヘッダー・ファイル SQL.H、SQLEXT.H、SQLUCODE.H、および WCHAR.H を参照してください。WCHAR.H には、Unicode に対応する「ワイド」文字フォーマットに関する情報が含まれることに注意してください。

スカラー関数

スカラー関数は、各行に対する値を返します。例えば、「絶対値」スカラー関数は、引数として数値列を取り、列内の各値の絶対値を返します。スカラー関数は、以下の ODBC エスケープ・シーケンスで呼び出します。

```
{fn scalar-function}
```

注: 開始文字と終了文字には、括弧ではなく中括弧を使用します。

スカラー関数のリストと使用法に関する詳細な例については、325 ページの『付録 F. スカラー関数』を参照してください。

solidDB のネイティブ・スカラー関数

solidDB は、以下のネイティブ・スカラー関数を提供しますが、ODBC エスケープ・シーケンスで呼び出すことはできません。以下の関数が提供されています。

- CURRENT_CATALOG() : 現行のアクティブ・カタログ名を含む WVARCHAR ストリングを返します。この名前は、ODBC スカラー関数 {fn DATABASE()} と同じです。
- LOGIN_CATALOG(): 接続されているユーザーに関するログイン・カタログを含む WVARCHAR ストリングを返します (現在、ログイン・カタログは、システム・カタログと同じです)。
- CURRENT_SCHEMA() : 現行のアクティブ・スキーマ名を含む WVARCHAR ストリングを返します。

関数からの戻りコード

アプリケーションが関数を呼び出すと、ドライバーはその関数を実行し、事前定義されたコードを返します。これらの戻りコードは、成功、警告、または失敗の状況を示します。以下の戻りコードがあります。

- SQL_SUCCESS
- SQL_SUCCESS_WITH_INFO
- SQL_NO_DATA_FOUND
- SQL_ERROR
- SQL_INVALID_HANDLE
- SQL_STILL_EXECUTING
- SQL_NEED_DATA

関数から SQL_SUCCESS_WITH_INFO または SQL_ERROR が返された場合、アプリケーションは、SQLError を呼び出して、エラーに関する追加情報をリトリーブすることができます。

データ・ソースへの接続

データ・ソースとしては、データベース・サーバー、フラット・ファイル、または別のデータ・ソースを使用できます。

データ・ソースにアクセスするには、アプリケーションで、SQLConnect() 呼び出しでデータ・ソース名を使用します。データ・ソース名は、接続ストリング、論理データ・ソース名、および空のデータ・ソース名の 3 つの方法のいずれかで指定します。

注: HotStandby を使用している場合には、基本接続と HotStandby 接続の 2 つの接続タイプから選択できます。以下に、基本接続について説明します。HotStandby 接続について詳しくは、「*solidDB 高可用性ユーザー・ガイド*」の『透過接続の使用』のトピックを参照してください。

solidDB 接続ストリングの使用

solidDB 接続ストリングは、*通信プロトコル*、使用可能な一連の*特殊オプション*、オプションのホスト・コンピューター名、および *サーバー名* で構成されています。

クライアントは、この組み合わせで接続先のサーバーを指定します。通信プロトコルとサーバー名は、サーバーがネットワーク listen 名で使用するものと一致する必要があります。さらに、クライアントとサーバーを別のマシンで実行している場合には、大部分のプロトコルで、ホスト・コンピューター名を指定する必要があります。クライアントのネットワーク名のすべてのコンポーネントでは、大/小文字を区別しません。

注: HSB 構成またはクラスター構成では、接続ストリングは、TC 情報 (透過接続情報) のより一般的な形式を取ることができます。詳しくは、「*solidDB 高可用性ユーザー・ガイド*」を参照してください。

同じフォーマットの接続ストリングが、`solid.ini` ファイル内の接続構成パラメーター、および ODBC アプリケーションと solidDB Light Client アプリケーションで使用されるデータ・ソース名の両方に適用されます。

接続ストリングのフォーマットは以下のとおりです。

protocol_name [*options*] [*server_name*] [*port_number*]

ここで、options には、以下の項目を任意の数だけ指定できます。

表 4. 接続ストリングのオプション

オプション	意味
-z	この接続では、データ圧縮が使用可能です。
-c *milliseconds*	ログイン・タイムアウトを指定します (デフォルトは、オペレーティング・システムに固有です)。指定された時間が経過すると、ログイン要求が失敗します。 **注:** TCP プロトコルにのみ適用されます。
-r *milliseconds*	接続 (または読み取り) タイムアウトを指定します (デフォルトは 60 秒です)。指定された時間内に応答が受信されないと、ネットワーク要求が失敗します。0 の値を指定すると、タイムアウトは無限に設定されます。 **注:** TCP プロトコルにのみ適用されます。

例

```
tcp localhost 1315
tcp 1315
tcp -z -c1000 1315
nmpipe host22 SOLIDDB
```

論理データ・ソース名の使用

データ・ソース名が有効な solidDB 接続ストリングではない場合、ドライバーは、それを論理データ・ソース名と想定します。

solidDB クライアントは、論理データ・ソース名をサポートします。これらの名前は、データベースに記述名を指定するのに使用できます。この名前は、以下の 3 つの方法でデータ・ソースにマップできます。

1. アプリケーションの solid.ini ファイル内のパラメーター設定を使用する。
2. Microsoft Windows オペレーティング・システムのレジストリー設定を使用する。
3. Windows ディレクトリーの中にある solid.ini ファイルの設定を使用する。

この機能は、サポート対象のすべてのプラットフォームで使用可能です。ただし、Windows 以外のプラットフォームでは、使用できるのは最初の方式のみです。

Windows で SQLConnect() を呼び出すと、solidDB ODBC ドライバーは、ODBC レジストリー内のすべての論理データ・ソースを検査し、論理データ・ソース名と有効な solidDB ODBC ドライバー接続ストリング間のマッピングを検出します。この操作で消費される時間は、定義されたデータ・ソースの量に比例します。接続時間は、以下のように想定できます。

- データ・ソースが少数 (1 から 5) のみの場合、接続時間は約 5 ms です。
- データ・ソースが 1000 個の場合、接続時間は約 200 ms です。

solidDB クライアントは、まず SOLIDDIR 環境変数で設定されたディレクトリーで solid.ini ファイルを開こうとします。この変数で指定されているパスにファイルがない場合、または変数が設定されていない場合には、現行作業ディレクトリーからファイルを開こうとします。

solid.ini ファイルを使用して論理データ・ソース名を定義するには、セクション [Data Sources] を含む solid.ini ファイルを作成する必要があります。このセクションに、定義する「論理名」と「ネットワーク名」のペアを入力する必要があります。パラメーターの構文は以下のとおりです。

```
[Data Sources]
logical_name = connect_string, Description
```

Description フィールドに、この論理名の目的に関するコメントを入力できます。

注: solidDB ODBC ドライバーは、接続を試行するたびに、solid.ini ファイルが存在するか検査します。例えば、作業ディレクトリーがネットワーク・ドライブにマップされているなどの理由で、ファイル・システムが著しく低速の場合、パフォーマンスにかなり重大な影響が及ぶ可能性があります。しかし、solid.ini ファイルの [Data Sources] セクションに、データ・ソース名とネットワーク名のマッピングが含まれる場合、ドライバーは、マッピングのためにレジストリーにアクセスを試みることはありません。

例えば、アプリケーション *My_application* 用に論理名を定義しようとしており、また接続するデータベースは、TCP/IP を使用する UNIX サーバーに存在すると仮定します。この場合、solid.ini ファイルをアプリケーションの作業ディレクトリーに配置し、そのファイルに以下の行を指定する必要があります。

```
[Data Sources]
My_application = tcpip irix 1313, Sample data source
```

これで、アプリケーションがデータ・ソース「My_application」を呼び出すと、solidDB クライアントは、それを「tcpip irix 1313」の呼び出しにマップします。

Windows プラットフォームでは、一般にデータ・ソースのマップにレジストリーが使用されます。GUI インターフェースでレジストリーをセットアップするには、Windows の「コントロール パネル」で、「管理ツール」、「データ ソース (ODBC)」の順に選択します。

レジストリー内の ODBC データ・ソースをマッピングするルールを、以下に詳細に説明します。

エントリーは、以下の場所に存在するパス `software¥odbc¥odbc.ini` から検索されます。

1. まず、ルート HKEY_CURRENT_USER で検索し、見つからない場合、
2. ルート HKEY_LOCAL_MACHINE で検索します。

Microsoft Windows システムでデータ・ソース名を解決する順序は以下のようになります。

1. 現行作業ディレクトリーの中にある `solid.ini` ファイルのセクション [Data Source] からデータ・ソース名が検索されます。
2. 以下のレジストリー・パスで、データ・ソース名を検索します。

   HKEY_CURRENT_USER¥software¥odbc¥odbc.ini¥DSN
3. 以下のレジストリー・パスでデータ・ソース名を検索します。

   HKEY_LOCAL_MACHINE¥software¥odbc¥odbc.ini¥DSN

注: solidDB ODBC ドライバーは、ODBC レジストリー内のすべての論理データ・ソースを検査し、論理データ・ソース名と有効な solidDB ODBC ドライバー接続ストリング間のマッピングを検出します。この操作で消費される時間は、定義されたデータ・ソースの量に比例します。1000 個のデータ・ソースでの SQLConnect() 接続時間は、約 200 ms です。

ドライバー・マネージャーをバイパスして、ドライバーに直接リンクすることで solidDB データベースのデータにアクセスするアプリケーションは、有効な接続ストリングを使用してサーバーに接続する必要があります。データ・ソース名が有効な solidDB 接続ストリングでない場合、すべての solidDB クライアント・アプリケーションが、以下で有効なデータ・ソース名を検索します。

1. `solid.ini` ファイル
2. `ODBC.INI` またはレジストリー

空のデータ・ソース名

アプリケーションが (空ストリングを指定することにより) solidDB サーバー・ネットワーク名を指定せずに ODBC API を直接使用し、SQLConnect() を呼び出す場合、それをクライアント・アプリケーションの `solid.ini` ファイルの [Com] セクション内のパラメーター Connect から読み取ります。

`solid.ini` ファイルは、アプリケーションの現行作業ディレクトリー内、または SOLIDDIR 環境変数で指定されたパスに存在する必要があります。

アプリケーションのワークステーションの `solid.ini` の中にある以下の接続行は、「spiff」という名前のホスト・コンピューターで実行し、名前「1313」(この場合は

ポート番号) を listen している solidDB サーバーに、TCP/IP プロトコルを使用して、アプリケーション (クライアント) を接続します。

```
[Com]
Connect = tcpip spiff 1313
```

solid.ini 構成ファイルの中に Connect パラメーターが存在しない場合、クライアントは代わりにその環境に応じたデフォルトを使用します。Listen パラメーターと Connect パラメーターのデフォルトは、アプリケーション (クライアント) が、デフォルトのネットワーク名で listen するローカル solidDB サーバーに常に接続するように選択されます。したがって、ローカル通信 (1 台のマシン内) では、接続の確立のために必ずしも構成ファイルが必要となるわけではありません。

Windows 用の solidDB ODBC データ・ソースの構成

Windows プラットフォーム用に ODBC データ・ソースを構成するには、このセクションで説明する手順を実行する必要があります。

始める前に

solidDB ODBC データ・ソースを構成するには、solidDB ODBC ドライバーがインストールされている必要があります。

手順

1. 「**コントロール パネル**」 → 「**管理ツール**」から「**データソース (ODBC)**」を起動します。
2. 「**ユーザー DSN**」タブを開きます。
3. 「**追加...**」ボタンをクリックします。
4. solidDB ODBC ドライバー (データベース要件に従って ANSI または UNICODE) を選択します。
5. 以下の例に示す「solidDB ODBC ドライバー セットアップ」ボックスで、データ・ソース構成を入力します。

   注: 「**NetworkName**」項目は、solid.ini で定義したデータベース・サーバーの listen アドレスと対応している必要があります。ネットワーク名は、 3 ページの『接続ストリングのフォーマット』に示す接続ストリング・フォーマットに従います。

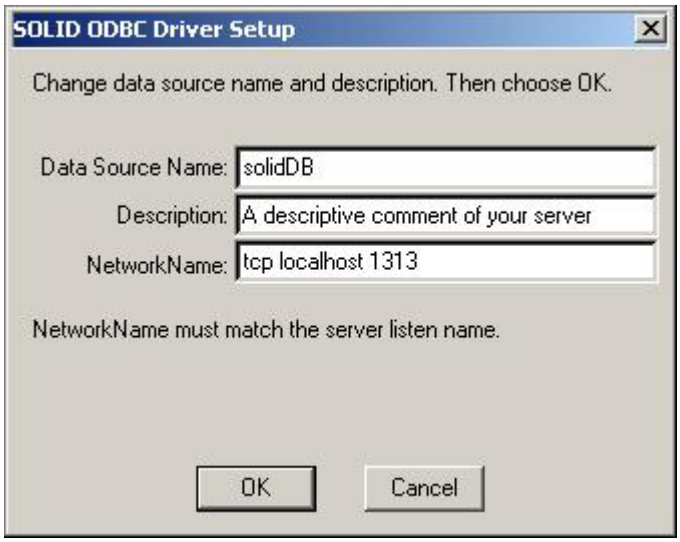

図 1. ODBC ドライバー セットアップ

ユーザー・ログイン情報のリトリーブ

このセクションでは、ドライバー・マネージャーがログイン情報をリトリーブする方法を説明します。

アプリケーションが SQLDriverConnect() を呼び出し、ユーザーに情報を促すプロンプトを出すよう要求した場合、ドライバー・マネージャーは、以下の例に示すようなダイアログ・ボックスを表示します。

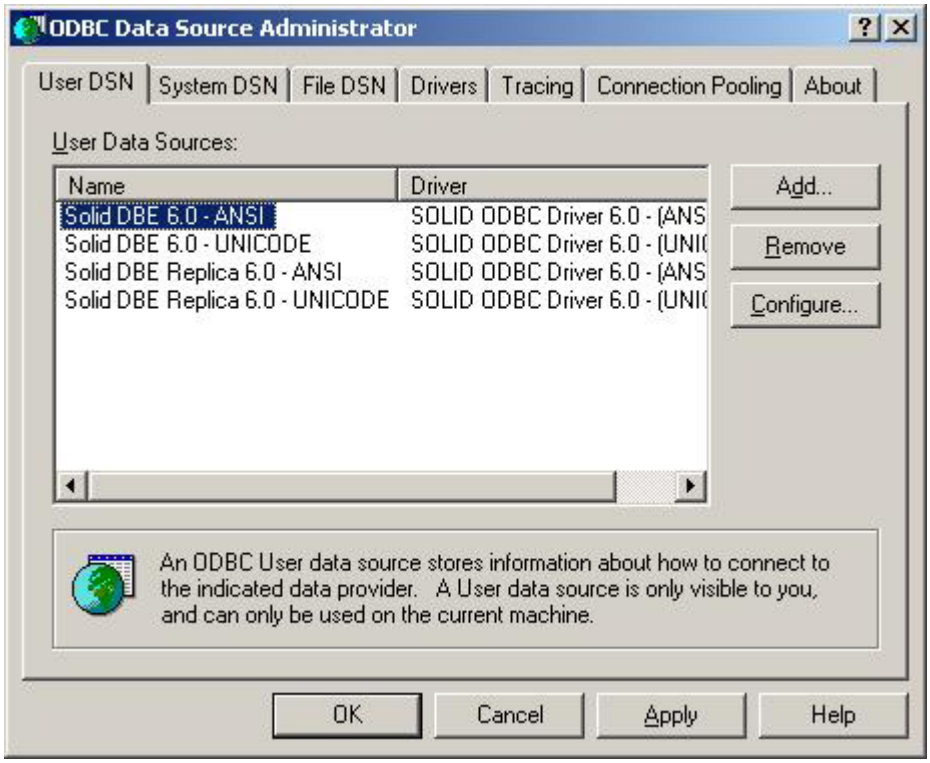

図 2. ODBC データ・ソース管理者

アプリケーションからの要求に従って、ドライバーは、ダイアログ・ボックスを表示してログイン情報をリトリーブします。

ODBC ハンドル妥当性検査

ODBC ハンドル妥当性検査は、オン/オフを切り替えることができます。

デフォルトではオフです。切り替えは、`solid.ini` のパラメーター Client.ODBCHandleValidation または ODBC 属性 `SQL_ATTR_HANDLE_VALIDATION` に基づいて行うことができます。

環境属性 `SQL_ATTR_HANDLE_VALIDATION` は、標準ではありません。(環境にバインドされない) グローバルな効力を持っており、接続を行う前に設定されている必要があります。その理由は、アプリケーションが、検証されたハンドルと検証されていないハンドルを割り振るのを防止して、整合性を維持するためです。

ハンドル妥当性検査をオフにするには、以下のように指定します。

SQLSetEnvAttr(henv, SQL_ATTR_HANDLE_VALIDATION, (SQLPOINTER)0, XX);

4 番目のパラメーター (XX) が無視されます。

ハンドル妥当性検査をオンにするには、以下のように指定します。

SQLSetEnvAttr(henv, SQL_ATTR_HANDLE_VALIDATION, (SQLPOINTER)1, XX);

トランザクションの実行

このセクションでは、トランザクションをコミットする方法について説明します。

*自動コミット・モード*では、各 SQL ステートメントは完全なトランザクションであり、ステートメントの実行が完了すると自動的にコミットされます。『*読み取り専用トランザクションのコミット*』セクションの読み取り専用 SELECT のコミットに関する重要な注意事項を参照してください。

*手動コミット・モード*では、トランザクションは 1 つ以上のステートメントで構成されています。手動コミット・モードでは、アプリケーションが SQL ステートメントをサブミットし、トランザクションが開かない場合には、ドライバーが暗黙的にトランザクションを開始します。このトランザクションは、アプリケーションが SQLEndTran でトランザクションをコミットまたはロールバックするまで開いたままになります。

読み取り専用トランザクションのコミット

重要:

- 分離レベルが READ COMMITED 以外の場合には、読み取り専用ステートメント (例えば SELECT) の場合であってもコミットする必要があります。さらに、サーバーが自動コミット・モードの場合であっても、ユーザーが SELECT ステートメントをコミットする必要があります。ステートメントのコミットに失敗すると、パフォーマンスが低下するか、またはサーバーがメモリーを使い尽くすことがあります。この点については、以下でさらに詳しく説明します。
- 分離レベルが READ COMMITTED の場合には、読み取り専用ステートメントをコミットする必要はありません。その場合、以下の説明は適用されません。

読み取り専用ステートメントであっても、コミットする必要があります。その理由は、solidDB は、各トランザクションの「読み取りレベル」を保存し、そのトランザクションがコミットするまで、他の接続からのそれ以降のトランザクションもすべてメモリー内に維持されるからです。(この動作は、Bonsai ツリー・テクノロジーで実行される行バージョン管理の一部です。 Bonsai ツリーについて詳しくは、「*solidDB 管理者ガイド*」を参照してください。) トランザクションをコミットしないと、他のトランザクションが累積するのに従って、サーバーはより多くのメモリーを必要とするようになります。その結果、パフォーマンスが低下し、最終的には、サーバーが使用可能メモリーを使い尽くすことがあります。詳しくは、「*solidDB 管理者ガイド*」の『パフォーマンス・チューニング』の章を参照してください。

SELECT および自動コミット

自動コミット・モードを使用しても、SELECT ステートメントが必ずコミットされるわけではありません。SELECT は単一のステートメントとして実行されるわけではないので、サーバーは、SELECT を自動的にコミットすることはできません。各 SELECT では、カーソルがオープンされ、行がフェッチされ、さらにカーソルがクローズされます。

複数行をフェッチするときに、サーバーが自動的にコミットするための方法が 2 つ考えられます。1 つは、最終フェッチ後にコミットする方法で、もう 1 つは各フェ

ッチ後にコミットする方法です。残念ながら、どちらの方法も実用的ではないため、サーバーは、自動コミット・モードであっても SELECT ステートメントをコミットすることはできません。

サーバーはどのフェッチが最終フェッチか分からない、すなわち、サーバーはユーザーがフェッチする行の数が分からないため、最終フェッチ後に自動的にコミットすることはできません。(ユーザーがカーソルをクローズするまで、サーバーは、ユーザーがフェッチを完了したことが分かりません。)

各フェッチ後にコミットするのは実用的ではありません。各トランザクションは、トランザクション開始時点のデータを参照する必要があるので、異なるトランザクションで各フェッチが行われている場合、データベースの異なる「スナップショット」からデータを取得する可能性があるからです。異なるトランザクションで各フェッチを実行すると、たとえカーソルが単一 SELECT ステートメントに関するものであっても、そのカーソルに対してトランザクション分離レベルの REPEATABLE READ および SERIALIZABLE が混乱するか、または無意味になる可能性があります。

SELECT ステートメントをコミットする場合、ユーザーは、以下の操作を実行できます。

- COMMIT WORK ステートメントを明示的に実行する。
- 自動コミットが適用されるステートメント (つまり、SELECT 以外のステートメント) を実行する。
- カーソルがオープン・カーソルのみの場合、ユーザーがカーソルを明示的にクローズすることによりコミットできる (サーバーが自動コミット・モードの場合で、ほかにオープン・カーソルが存在しない場合、カーソルをクローズするとサーバーは自動的にコミットします)。これは、カーソルごとに処理が完了したら、すぐにそのカーソルを明示的にクローズするよう推奨されている理由の 1 つです。

注: カーソル内のデータが整合していること、および最新であることを保証するため、サーバーは、実際には、カーソルをオープンする直前に自動コミットを実行します (自動コミットがオンの場合)。次に、サーバーは、それに続く FETCH ステートメント (複数可) を含む新しいトランザクションをすぐに開始します。他のトランザクションと同様に、この新しいトランザクションもコミット (またはロールバック) する必要があります。

サマリー

READ COMMITTED 以外の分離レベルを使用している場合、読み取り専用ステートメントであっても、すべてのステートメントをコミットする必要があります。

ほとんどの場合、自動コミット・モードで SELECT ステートメントを実行するときは、各カーソルの処理の完了時にすぐにそのカーソルを明示的にクローズし、自動コミット・モードであっても、明示的に COMMIT を行う必要があります。

データ・ソースのカタログに関する情報のリトリーブ

このセクションでは、データ・ソースのカタログに関する情報を返す関数 (*カタログ関数* と呼ばれる) について説明します。

- SQLTables は、データ・ソースに格納されている表の名前を返します。
- SQLTablePrivileges は、1 つ以上の表に関連付けられた特権を返します。
- SQLColumns は、1 つ以上の表内の列の名前を返します。
- SQLColumnPrivileges は、単一の表内の各列に関連付けられた特権を返します。
- SQLPrimaryKeys は、単一の表の主キーを含む列の名前を返します。
- SQLForeignKeys は、単一の表内の外部キーである列の名前を返します。また、指定された表の主キーを参照する他の表内の列の名前を返します。
- SQLSpecialColumns は、単一の表内の行をユニークに識別する最適な列セットに関する情報、またはトランザクションによって行内の値が更新された場合に自動的に更新されるその表内の列に関する情報を返します。
- SQLStatistics は、単一の表およびその表に関連付けられた索引に関する統計を返します。
- SQLProcedures は、データ・ソースに格納されたプロシージャーの名前を返します。
- SQLProcedureColumns は、1 つ以上のプロシージャーに関して、入出力パラメーターのリスト、および結果セット内の列の名前を返します。

各関数は、情報を結果セットとして返します。アプリケーションは、SQLBindCol() と SQLFetch() を呼び出すことにより、これらの結果をリトリーブします。

関数の非同期実行

注: すべての solidDB 製品の ODBC ドライバーは、非同期実行をサポートしません。

SQL に対する ODBC 拡張機能の使用

ODBC は、SQL に対する拡張機能を定義しており、これは、大部分のデータベース管理システムに対して共通です。

SQL 拡張機能について詳しくは、「Microsoft ODBC *プログラマーズ リファレンス (Microsoft ODBC Programmer's Reference)*」の導入部分が記載されている「Microsoft ODBC API 仕様書 (Microsoft ODBC API Specification)」(IBM Corporation の Web サイトから Part I を PDF ファイルで提供) の『ODBC のエスケープ シーケンス (Escape Sequences in ODBC)』を参照してください。

SQL に対する ODBC 拡張機能には、以下の内容が含まれます。

- プロシージャー
- ヒント

これらの拡張機能に関する solidDB の使用について詳しくは、以下の各セクションで説明します。

プロシージャー

ストアード・プロシージャーは、1 つ以上の SQL ステートメントとプログラム・ロジックを含むプロシージャー型プログラム・コードです。

ストアード・プロシージャーはデータベースに格納されており、アプリケーションまたは他のストアード・プロシージャーからの 1 回の呼び出しで実行されます。solidDB ストアード・プロシージャーの完全な説明については、「*solidDB SQL ガイド*」でストアード・プロシージャーに関する説明を参照してください。

アプリケーションは、SQL ステートメントの代わりにプロシージャーを呼び出すことができます。プロシージャーの呼び出しで ODBC が使用するエスケープ節は、以下のとおりです。

```
{call procedure-name [([parameter][,[parameter]]...)]}
```

ここで、procedure-name は、データ・ソースに格納されているプロシージャーの名前を指定し、parameter は、プロシージャー・パラメーターを指定します。

注: ODBC 標準は、エスケープ節を以下のように指定します。

```
{[?=] call procedure-name [([parameter][,[parameter]]...)]}
```

ただし、solidDB の構文では、オプションの「?=」部分がサポートされません。(以前のバージョンの「*solidDB プログラマー・ガイド*」では、solidDB は「?=」構文をサポートすると記載されていますが、これは正しくありません。)

プロシージャーには、ゼロ個以上のパラメーターを指定できます。入力パラメーターおよび入出力パラメーターでは、*parameter* にリテラルまたはパラメーター・マーカーを使用できます。データ・ソースによってはリテラル・パラメーター値を受け入れないため、相互運用可能なアプリケーションではパラメーター・マーカーを使用するようにしてください。出力パラメーターについては、*parameter* は、パラメーター・マーカーにする必要があります。プロシージャー呼び出しにパラメーター・マーカーが含まれる場合、アプリケーションは、プロシージャーを呼び出す前に SQLBindParameter() を呼び出して、各マーカーをバインドする必要があります。

プロシージャー呼び出しでは、入力パラメーターおよび入出力パラメーターは必須ではありません。以下のルールに注意してください。

- {call *procedure_name()*} のように、パラメーターを省略した括弧付きで呼び出されたプロシージャーは、失敗する可能性があります。
- {call *procedure_name*} のように、括弧なしで呼び出されたプロシージャーは、パラメーター値を返しません。
- 入力パラメーターは省略できます。入力パラメーターまたは入出力パラメーターを省略すると、ドライバーは、データ・ソースに対してパラメーターのデフォルト値を使用するよう指示します。オプションとして、パラメーターのデフォルト値は、そのパラメーターにバインドされた長さ/標識バッファーの値を使用して、SQL_DEFAULT_PARAM に設定できます。
- パラメーターを省略する場合には、他のパラメーターと区切るためのコンマを入れる必要があります。
- 入出力パラメーターまたはリテラル・パラメーター値を省略すると、ドライバーは、出力値を破棄します。

- プロシージャーの戻り値に対するパラメーター・マーカーを省略すると、ドライバーは、戻り値を破棄します。
- アプリケーションで、値を返さないプロシージャーに戻り値パラメーターを指定すると、ドライバーは、そのパラメーターにバインドされた長さ/標識バッファーの値を SQL_NULL_DATA に設定します。

データ・ソースがプロシージャーをサポートするかどうかを判断するには、アプリケーションで、SQL_PROCEDURES 情報タイプを指定して SQLGetInfo() を呼び出します。プロシージャーについて詳しくは、「*solidDB SQL ガイド*」でストアード・プロシージャーの説明を参照してください。

ヒント

照会内で、オプティマイザー・ディレクティブまたはヒント を指定して、使用する照会実行プランを決定できます。

ヒントは、SQL-92 の疑似コメント構文を通して検出されます。solidDB は、ヒントに対して独自の拡張機能を提供します。

```
--(* vendor (Solid), product (Engine), option(hint)
--hint
-- *)--
hint :=
    [MERGE JOIN |
    LOOP JOIN |
    JOIN ORDER FIXED |
    INTERNAL SORT |
    EXTERNAL SORT |
    INDEX [REVERSE] table_name.index_name |
    PRIMARY KEY [REVERSE] table_name |
    FULL SCAN table_name |
    [NO] SORT BEFORE GROUP BY]
```

疑似コメント接頭部の後に、識別情報が続きます。vendor は *Solid*、product は *Engine*、および疑似コメント・クラス名である option は有効な hint で指定します。

終止符はそれ自体で 1 行にすることも、またヒントの最終行の末尾に置くこともできます。例えば、以下のどちらも受け入れられます。

```
--(* vendor (Solid), product (Engine), option(hint)
--hint
-- *)--
```

または

```
--(* vendor (Solid), product (Engine), option(hint)
--hint *)--
```

スペーシングは重要であることに注意してください。疑似コメント接頭部 --(* および接尾部 *)-- では、括弧とアスタリスクの間にスペースを入れることはできません。*)-- 終止符の前、つまりアスタリスクの前にはスペースが必要です (上記の例を参照)。--(* 内で、左括弧の前にはスペースは必要ありません。終止符 *)-- は、コメント区切り -- の後以外では、それ自体で 1 行にすることはできません。

ヒントは、常に、適用対象の SELECT、UPDATE、または DELETE のキーワードに続きます。

注: ヒントは、INSERT キーワードの後には指定できません。

各副選択では、独自のヒントが必要です。例えば、以下のヒント構文は有効な使用方法です。

```
INSERT INTO ... SELECT hint FROM ...
UPDATE hint TABLE ... WHERE column = (SELECT hint ... FROM ...)
DELETE hint TABLE ... WHERE column = (SELECT hint ... FROM ...)
```

ヒント例 1

```
SELECT--(* vendor(SOLID), product(Engine), option(hint)
--MERGE JOIN
--JOIN ORDER FIXED
-- *)--
col1, col2
FROM TAB1 A, TAB2 B;
WHERE A.INTF = B.INTF;
```

ヒント例 2

```
SELECT--(* vendor(SOLID), product(Engine), option(hint)
--INDEX TAB1.INDEX1
--INDEX TAB1.INDEX1 FULL SCAN TAB2
-- *)--
*
FROM TAB1, TAB2
WHERE TAB1.INTF = TAB2.INTF;
```

ヒントは、特定の動作に対応して特定のセマンティックを使用します。以下に、solidDB がサポートするヒントのリストを示します。

表 5. solidDB がサポートするヒント

ヒント	定義
MERGE JOIN	FROM 節内にリストされたすべての表に関して、選択照会内でマージ結合アクセス・プランを選択するよう、オプティマイザーに指示します。MERGE JOIN オプションは、2 つの表のサイズがほぼ同じ場合で、データが均等に分散している場合に使用します。同じ数量の行を結合する場合には、LOOP JOIN よりも高速です。データの結合に関して、MERGE JOIN は最大で 3 つの表をサポートします。結合表の順序は、列を結合し、列の結果を組み合わせることによって決まります。 このヒントは、データが結合キーによってソートされる場合で、ネストされたループ結合パフォーマンスが十分でない場合に使用できます。オプティマイザーは、表間に等価述部 (例えば「table1.col1 = table2.col1」) が存在する場合にのみ、マージ結合を選択します。その他の場合には、オプティマイザーは、MERGE JOIN ヒントが指定されている場合であっても LOOP JOIN を選択します。 マージ操作を行う前にデータがソートされていない場合には、solidDB 照会実行プログラムがデータをソートすることに注意してください。 ソートを行うマージ結合では、ソートを行わないマージ結合と比較して、多くのリソースが必要となることに注意してください。
LOOP JOIN	FROM 節内にリストされたすべての表に関して、選択照会内でネストされたループ結合を選出するよう、オプティマイザーに指示します。デフォルトでは、オプティマイザーは、ネストされたループ結合を選出しません。 LOOP JOIN は、内部表と外部表をループし、内部表と外部表の列間の一致を検索します。パフォーマンスを向上させるために、結合列は索引付けする必要があります。 表が小さく、メモリー内に収まる場合には、ループ結合を使用すると、他の結合アルゴリズムを使用するよりも効率が高くなることがあります。
JOIN ORDER FIXED	オプティマイザーが、照会の FROM 節でリストされた順序で結合内の表を使用するよう指定します。これは、オプティマイザーが結合順序を再整理を試みないこと、および結合を完了するための代替アクセス・パスを検索を試みないこと意味しています。 EXPLAIN PLAN 出力を実行してヒントを「テスト」し、生成されたプランが指定された照会に対して最適であるか確認することを推奨します。

表 5. *solidDB* がサポートするヒント *(続き)*

ヒント	定義
INTERNAL SORT	照会実行プログラムが内部ソーターを使用するよう指定します。このヒントは、想定される結果セットが小さい場合 (数千行ではなく、数百行) に使用します。例えば、総計、小さな結果セットでの ORDER BY や GROUP BY などを実行する場合などです。 このヒントを使用すると、よりコストのかかる外部ソーターを使用する必要がなくなります。

EXTERNAL SORT	照会実行プログラムが外部ソーターを使用するよう指定します。このヒントは、想定される結果セットが大きく、メモリーに収まらない場合に使用します。例えば、数千行の結果セットが想定される場合などです。
	
	さらに、外部ソート・ヒントを使用する前に、`solid.ini` に SORT 作業ディレクトリーを指定します。作業ディレクトリーを指定しないと、ランタイム・エラーが発生します。作業ディレクトリーは、`solid.ini` 構成ファイルの [sorter] セクションに指定します。例えば、以下のようにします。
	
	`[sorter]`
	`TmpDir_1=c:¥soldb¥temp1`

INDEX [REVERSE] *table_name.index_name*	指定した表に対して、指定した索引スキャンを強制的に実行します。この場合、アクセス・プランの作成に使用できるその他の索引が存在するか、または指定した照会に対して表スキャンの方が優れているかの評価には、オプティマイザーは進みません。 EXPLAIN PLAN 出力を実行してヒントを「テスト」し、生成されたプランが指定された照会に対して最適であるか確認することを推奨します。 オプションのキーワード REVERSE を指定すると、行が逆順に返されます。この場合、照会実行プログラムは索引の最終ページから開始して、索引のキーの降順 (逆) で行を返し始めます。 *tablename.indexname* 内では、tablename は完全修飾表名であり、*catalogname* および *schemaname* を含むことができます。
PRIMARY KEY [REVERSE] *table_name*	指定した表に対して、主キー・スキャンを強制的に実行します。 オプションのキーワード REVERSE を指定すると、行が逆順に返されます。 指定した表に対して主キーを使用できない場合、ランタイム・エラーが発生します。

表 5. solidDB がサポートするヒント (続き)

ヒント	定義
FULL SCAN *table_name*	指定した表に対して、表スキャンを強制的に実行します。この場合、アクセス・プランの作成に使用できるその他の索引が存在するか、または指定した照会に対して表スキャンの方が優れているかの評価には、オプティマイザーは進みません。 このヒントを使用する前に、EXPLAIN PLAN 出力を実行してヒントを「テスト」し、生成されたプランが指定された照会に対して最適であるか確認することを推奨します。
[NO] SORT BEFORE GROUP BY	GROUP BY 列で結果セットをグループ化する前に、SORT 操作を行うかどうかを示します。 グループ項目が少数 (数百行) の場合、NO SORT BEFORE を使用します。一方、グループ項目が多数 (数千行) の場合、SORT BEFORE を使用します。

ODBC 拡張機能での追加の関数

ODBC は SQL ステートメントに関する以下の関数を提供しています。

これらの関数について詳しくは、「Microsoft ODBC API 仕様書 (Microsoft ODBC API Specification)」を参照してください (IBM Corporation の Web サイトに Part II の PDF ファイルが提供されています)。

表 6. ODBC 拡張機能での追加の関数

関数	説明
SQLDescribeParam	準備済みパラメーターに関する情報をリトリーブします。
SQLNumParams	SQL ステートメント内のパラメーター数をリトリーブします。
SQLSetStmtAttr SQLSetConnectAttr SQLGetStmtAttr	これらの関数は、非同期処理、行セットのバインド方向、返す可変長データの最大量、返す結果セット行の最大数、照会タイムアウト値などのステートメント・オプションを設定またはリトリーブします。SQLSetConnectAttr は接続内のすべてのステートメントのオプションを設定することに注意してください。

ODBC API に対する solidDB 拡張機能

以下の関数および接続属性は、ODBC API に対する solidDB 固有の拡張機能です。

非標準の ODBC 関数

表 7. ODBC API に対する solidDB 固有の ODBC 関数

関数	説明
SQLFetchPrev	この関数は、ODBC 関数 SQLFetch と同じですが、過去のレコードをフェッチします。SQLFetch 関数について詳しくは、「Microsoft ODBC API 仕様書 (Microsoft ODBC API Specification)」を参照してください (IBM Corporation の Web サイトに Part II の PDF ファイルが提供されています)。
SQLSetParamValue	この関数は、SQLPrepare で指定されている SQL ステートメント内にパラメーター・マーカー値を設定します。パラメーター・マーカーは、1 から順に左から右へ番号が付けられており、任意の順序で設定できます。 この関数について詳しくは、85 ページの『非 ODBC solidDB Light Client 関数』を参照してください。
SQLGetCol	この関数は、ODBC 関数 SQLGetData と同じです。この関数について詳しくは、「Microsoft ODBC API 仕様書 (Microsoft ODBC API Specification)」を参照してください (IBM Corporation の Web サイトに Part II の PDF ファイルが提供されています)。
SQLGetAnyData	この関数は、ODBC 関数 SQLGetData と同じです。この関数について詳しくは、「Microsoft ODBC API 仕様書 (Microsoft ODBC API Specification)」を参照してください (IBM Corporation の Web サイトに Part II の PDF ファイルが提供されています)。

非標準の ODBC 属性

以下の接続属性は、solidDB に固有のものです。

注: OUT のマークが付いている属性は読み取り専用であり、ODBC インターフェースでは設定できません。

- SQL_ATTR_TF_LEVEL

  OUT: 整数 (TF レベル: 0=NONE、1=CONNECTION、3=SESSION)

  障害透過性レベル。
- SQL_ATTR_TC_PRIMARY

  OUT: ストリング、1 次サーバー接続ストリング

  現行 1 次サーバーを示す値が常に存在します。
- SQL_ATTR_TC_SECONDARY

  OUT: ストリング、2 次サーバー接続ストリング

この値は、以下の場合にはワークロード割り当て済みサーバーを示します。

1. PA=READ_MOSTLY で、しかも
2. 2 次サーバーが指定されたワークロード・サーバーである場合。

それ以外の場合、返されるストリングは空です。

- SQL_ATTR_TF_WAITING

  OUT: ストリング、2 次サーバー接続ストリング

  この値は、割り当てられたウォッチドッグ (待機) 接続を示します。待機接続は、1 次サーバーの損失 (破損、使用不可の状態) の可能性をより素早く検出するために、ODBC ドライバーによって内部で使用されます。接続が TC 接続でない場合、このストリングは空です。

- SQL_ATTR_PA_LEVEL

  OUT: 整数 (優先アクセス・レベル: 0=WRITE_MOSTLY、1=READ_MOSTLY)

  この属性は、ロード・バランシングを使用するかどうかを示します。

- SQL_ATTR_TC_WORKLOAD_CONNECTION

  OUT: ストリング、ワークロード接続のサーバー名

  現行のワークロード接続サーバー。コミットの前に照会があった場合、値はトランザクションがコミットされるサーバーを示します。これは、ステートメント属性としても照会できます。その場合、これは次のステートメントを実行するサーバーを示します。

- SQL_ATTR_LOGIN_TIMEOUT_MS

  IN/OUT: 整数、ログイン・タイムアウト (ミリ秒単位)

  注: 秒単位でタイムアウトを設定するのに使用できる標準属性 SQL_ATTR_LOGIN_TIMEOUT もあります。

- SQL_ATTR_CONNECTION_TIMEOUT_MS

  IN/OUT: 整数、接続タイムアウト (ミリ秒単位)

  注: 秒単位でタイムアウトを設定するのに使用できる標準属性 SQL_ATTR_CONNECTION_TIMEOUT もあります。

- SQL_ATTR_QUERY_TIMEOUT_MS

  IN/OUT: 整数、照会タイムアウト (ミリ秒単位)

  注: 秒単位でタイムアウトを設定するのに使用できる標準属性 SQL_ATTR_QUERY_TIMEOUT もあります。

- SQL_ATTR_IDLE_TIMEOUT

  IN/OUT: 整数、接続アイドル・タイムアウト (分単位)

サーバーが使用する、接続固有のアイドル・タイムアウトを示します。指定された期間、接続上でアクティビティーがない場合、サーバーは自動的にその接続をシャットダウンし、事実上ユーザーを退去させます。

特殊なセマンティック:

– -1 (デフォルト): 接続タイムアウトは、サーバー・デフォルトと同等

– 0: アイドル・タイムアウトなし。接続は決して閉じられない

このプロパティー値は、SQLConnect() の実行前にのみ設定できます。

- SQL_ATTR_HANDLE_VALIDATION (環境ハンドル属性)

  IN/OUT: 整数、ODBC 標準ハンドル妥当性検査をオン (1) またはオフ (0) にします。

  デフォルトは 0 です。

  この属性はグローバルです。つまり、いったん設定すると、アプリケーションによって開始されるすべての solidDB ODBC 接続に影響します。特定のシステム、例えば ODBC ドライバー・マネージャーを含んでいる Windows では、そのドライバー・マネージャーがハンドルの妥当性検査を行うため、solidDB ODBC ドライバー単独で同じ妥当性検査手順を繰り返す必要はありません。また、入念に作成された ODBC アプリケーションでは、通常、無効なハンドルは使用されません。その場合、ODBC ドライバーでのハンドルの妥当性検査は必要ありません。どちらの場合でも、ドライバー内でのハンドルの妥当性検査をスキップすると、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。ハンドルの妥当性検査をオフにし、アプリケーションで無効なハンドルを使用した場合、ODBC ドライバーの動作は予測できず、多くの場合、アプリケーションは異常終了します。

- SQL_ATTR_SET_CONNECTION_DEAD

  IN/OUT: 整数、必要な場合は 1 に設定する必要がある

  この属性を接続に対して設定すると、ドライバーは強制的に接続を中止しますが、サーバーとのハンドシェークは切断しません。この属性を 1 に設定した後、接続は使用不能になります。

カーソルの使用

ODBC ドライバーではカーソル という概念を使用して、結果セット、すなわちデータベースからリトリーブしたデータ行内の位置を追跡します。コンピューター画面上のカーソルが現在位置を示すのと同様に、カーソルを使用して現在位置を追跡し示します。

アプリケーションが SQLFetch を呼び出すたびに、ドライバーはカーソルを次の行に移動して、その行を返します。アプリケーションは SQLFetchScroll または SQLExtendedFetch (ODBC 2.x) を呼び出すこともでき、これによって 1 回のフェッチまたは呼び出しで複数行をフェッチし、アプリケーション・バッファーに入れます。これは「ブロック・カーソル」サポートとして知られています。実際にフェッチされる行の数は、アプリケーションが指定する行セットのサイズによって異なることに注意してください。

アプリケーションは SQL_POSITION オプションを使用して SQLSetPos を呼び出し、フェッチされたデータ・ブロック内にカーソルを位置付けることができます。これによって、アプリケーションは行セット内のデータをリフレッシュできます。SQLSetPos は、SQL_UPDATE オプションを指定してデータを更新したり、SQL_DELETE オプションを指定して結果セット内のデータを削除する場合も呼び出されます。

コア ODBC 関数がサポートしているカーソルは、一度に 1 行ずつ、前方スクロールのみ行います。(結果セットから既にリトリーブしたデータ行を再度リトリーブするには、アプリケーションで SQL_CLOSE オプションを指定して SQLFreeStmt を呼び出すことによりカーソルをクローズし、SELECT ステートメントを再実行し、さらに SQLFetch、SQLFetch Scroll、または SQLExtendedFetch (ODBC 2.x) を使用して、対象の行がリトリーブされるまで行をフェッチする必要があります。) 前方だけでなく、後方スクロールの機能も必要な場合は、ブロック・カーソルを使用してください。

行セットへのストレージの割り当て (バインディング)

個別のデータ行をバインドするだけでなく、アプリケーションは SQLBindCol を呼び出して、*行セット* (1 つ以上のデータ行) にストレージを割り当てることができます。デフォルトでは、行セットは列方向にバインドされます。行方向にバインドすることもできます。

行セットに含まれるデータ行の数を指定するには、アプリケーションで SQL_ROWSET_SIZE オプションを指定して SQLSetStmtAttr を呼び出します。

列方向バインディング

列方向のバインド結果にストレージを割り当てるには、アプリケーションでバインド対象の各列に対して、以下の手順を実行します。

1. データ・ストレージ・バッファーの配列を割り振ります。この配列のエレメントの数は、行セット内にある行の数と同じです。
2. データ値ごとに返すことができるバイト数を保持するストレージ・バッファーを割り振ります。この配列のエレメントの数は、行セット内にある行の数と同じです。
3. SQLBindCol を呼び出して、データ配列のアドレス、データ配列の 1 エレメントのサイズ、バイト数配列のアドレス、およびデータ変換後の型を指定します。データをリトリーブすると、ドライバーは配列エレメントのサイズに基づき、データの連続行を配列内のどこに格納するかを判別します。

行方向バインディング

行方向のバインド結果にストレージを割り当てるには、アプリケーションで以下の手順を実行します。

1. リトリーブしたデータの 1 行と、それに関連するデータ長を保持できる構造体を宣言します。(バインドする列ごとに、構造体は、データ用の 1 つのフィールドと、返すことができるデータのバイト数用の 1 つのフィールドを含みます。)
2. これらの構造体の配列を割り振ります。この配列のエレメントの数は、行セット内にある行の数と同じです。

3. バインドする列ごとに SQLBindCol を呼び出します。各呼び出しでは、アプリケーションが最初の配列エレメント内の列のデータ・フィールドのアドレス、そのデータ・フィールドのサイズ、最初の配列エレメント内の列のバイト数フィールドのアドレス、およびデータ変換後の型を指定します。
4. SQL_BIND_TYPE オプションを指定して SQLSetStmtAttr を呼び出し、構造体のサイズを指定します。データをリトリーブすると、ドライバーは構造体のサイズに基づき、データの連続行を配列内のどこに格納するかを判別します。

カーソル・サポート

結果セットの基礎表への変更を検知するために、アプリケーションはさまざまな手段を必要とします。このようなニーズに対応するために、各種のカーソル・モデルが設計されており、それぞれが結果セットの基礎表の変更へのセンシティビティーが異なります。

例えば、財務データの残高処理を行う場合、経理担当者は静的に見えるデータを必要とします。データが継続的に変化している場合、帳簿で残高処理を行うことはできません。コンサートのチケットを販売する場合、販売員は、どのチケットがまだあるのかを示す最新の、すなわち動的なデータが必要です。

solidDB のカーソルは SQLSetStmtAttr で「動的」として設定されますが、部分的に動的な動作をする静的カーソルと非常に似ています。solidDB の動的カーソルの動作は、ユーザーに対し、他のユーザーが行った結果セットへの変更が可視である ODBC の動的カーソルとは異なり、そのような変更がユーザーに可視ではないという点で静的といえます。

solidDB では、カーソルをブロックからブロックに前方にスクロールし、後方にスクロールしない限り、あるいは更新後に同じブロック内で前後に移動している限り、カーソルの動作は動的なものになります。これは、すべての変更が可視であることを意味します。ただし、この動作は、solidDB の AUTOCOMMIT モード設定によって影響されることに注意してください。詳しくは、39 ページの『カーソルと自動コミット』を参照してください。 SQLSetPos を使用した場合のカーソルの動作の例については、41 ページの『カーソルと位置付け操作』を参照してください。

solidDB のカーソルの動作に関するもう 1 つの特性は、トランザクションがそれ自体によるデータ変更は (いくつかの制限付きで) 表示できるが、時間的にオーバーラップしている他のトランザクションによる変更は表示できないという点です。(ユーザー自身のデータ変更を表示する場合の制限について詳しくは、41 ページの『カーソルと位置付け操作』を参照してください。) 例えば Transaction_A が開始すると、Transaction_A の開始前に作業をコミットしていなかった他のどのトランザクションによる変更も表示されません。ユーザー自身の変更がそのユーザーに不可視になる solidDB の条件には以下があります。

- SELECT ステートメントで、ORDER BY 節、または GROUP BY 節を使用したときに、solidDB が結果セットをキャッシュに入れると、ユーザー自身の変更がそのユーザーに不可視になります。
- ADO または OLE DB を使用して作成されたアプリケーションでは、solidDB のカーソルは行セットの更新などの機能を有効にするために、動的 ODBC カーソルのように機能します。

カーソル・タイプの指定

カーソル・タイプを指定するには、アプリケーションで SQL_CURSOR_TYPE オプションを指定して SQLSetStmtAttr を呼び出します。アプリケーションは、前方スクロールのみを行うカーソル、静的カーソル、または動的カーソルを指定することができます。

カーソルが前方スクロール・カーソルでない限り、アプリケーションは SQLExtendedFetch (ODBC 2.x) または SQLFetchScroll (ODBC 3.x) を呼び出して、カーソルを後方または前方にスクロールさせます。

カーソル・サポート

このセクションでは、solidDB がサポートしているカーソル・タイプについて説明します。

ODBC 3.51 では 3 つのタイプのカーソルが定義されています。

- ドライバー・マネージャーがサポートしているカーソル
- サーバーがサポートしているカーソル
- ドライバーがサポートしているカーソル

solidDB のカーソルは、サーバーがサポートしているカーソルです。

カーソルと自動コミット

このセクションでは、カーソルと自動コミットの情報について説明します。

カーソルと自動コミットに関する solidDB 固有の情報については、25 ページの『読み取り専用トランザクションのコミット』を参照してください。

ご使用のアプリケーションがブロック・カーソルと、位置付け更新および位置付け削除を使用する場合は、solidDB の自動コミット・モードの使用には、いくつかの制限もあります。これらのカーソル機能の簡単な説明については、36 ページの『カーソルの使用』を参照してください。

ブロック・カーソルと、位置付け更新および位置付け削除を使用する際は、以下を行う必要があります。

- アプリケーション内で、コミット・モードを SQL_AUTOCOMMIT_OFF に設定します。
- すべてのフェッチ操作と位置付け操作の完了後にのみ、アプリケーションでの変更をコミットします。
- 位置付け操作を行うたびに変更をコミットしないようにしてください。

重要:

アプリケーションが SQL_AUTOCOMMIT_ON のコミット・モードを使用する場合や、すべての位置付け操作が完了する前に変更をコミットする場合は、結果セットのブラウズ中にアプリケーションに予測不能な動作が発生する可能性があります。詳しくは、以下のセクションを参照してください。

位置付けカーソル操作と SQL_AUTOCOMMIT_ON

solidDB ODBC ドライバーは、データベースからリトリーブしたデータ行である行セット内の各行の行番号/カウンターを保持しています。アプリケーションのコミット・モードが SQL_AUTOCOMMIT_ON に設定されており、行セット内の行に対して位置付け更新または位置付け削除を実行した場合は、データベース内のその行がすぐに更新されます。行の新しい値によっては、行が結果セット内の元の位置から移動されることがあります。更新された行が移動され、新しい位置は予測不能なので (新しい値によってまったく異なるため)、ドライバーはこの行のカウンターを失います。

また、更新された行の位置が変わったため、行セット内の他のすべての行のカウンターも無効になる可能性があります。したがって、次回フェッチ操作または SQLSetPos 操作を行うときに、アプリケーションに不正な動作が発生する可能性があります。

以下の例では、この制限を説明します。

アプリケーションで以下の手順を実行するとします。

1. コミット・モードを SQL_AUTOCOMMIT_ON に設定します。
2. 行セット・サイズを 5 に設定します。
3. 照会を実行して、n 行を含む結果セットを生成します。
4. SQLFetchScroll を使用して、5 行の最初の行セットをフェッチします。

   サンプル結果セットを以下に示します。このサンプルでは、結果セットに含まれる列は 1 つだけです (varchar(32) として定義されています)。以下の表では、最初の列は、ドライバーによって内部で維持されている行番号を示します。 2 番目の列は、行の実際の値を示します。

表 8. サンプル結果セット

ドライバーにより内部に保管された行カウンター	行の値
1	Antony
2	Ben
3	Charlie
4	David
5	Edgar

ここで、アプリケーションが SQLSetPos を呼び出して、3 行目を新しい値 Gerard で更新するとします。この更新を行うには、新しい行の値を移動して、以下のように位置付けします。

表 9. サンプル結果セット

ドライバーにより内部に保管された 行カウンター	行の値
1	Antony
2	Ben
空の行	
4	David
5	Edgar
新しい行	Gerard

ここで「David」の行カウンターは 4 ではなく 3 になります。一方、「Edgar」のカウンターは 5 ではなく 4 になります。行カウンターの一部が無効になったので、ドライバーがカーソルの相対的位置付けまたは絶対位置付けを行うと間違った結果を返すことになります。

コミット・モードを SQL_AUTOCOMMIT_OFF に設定しておくと、SQLEndTran 関数を呼び出して変更をコミットするまでデータベースは更新されません。

カーソルと自動コミットに関する solidDB 固有の情報については、25 ページの『読み取り専用トランザクションのコミット』を参照してください。

カーソルと位置付け操作

アプリケーションが位置付け操作 (SQLSetPos の呼び出しの際の更新や削除など) を実行する場合、結果セットの可視性には制限があります。

ケース 1 は、SQLSetPos を使用した場合のカーソルの動作を示したものです。ケース 1 では、更新の適用後に、同じブロック内でカーソルが前後にスクロールします。

ケース 1 は、カーソル使用時の結果セット内の更新の可視性を示すものですが、更新が可視になる正確な環境は、いくつかの要因によって異なります。例えば、メモリー・バッファー・サイズに対する結果セットの相対的なサイズ、トランザクションの分離レベル、データをコミットする頻度などの要因が挙げられます。

ケース 2 は、行セット内でカーソルが後方にスクロールする場合、または更新の適用後に異なる行セット内でカーソルが前後に移動する場合に、SQLSetPos を使用する際のカーソル動作の制限を示したものです。

ケース 1

以下の例では、位置付け操作を使用したカーソル動作と、位置付け更新をユーザーに可視にする方法を示します。

アプリケーションで以下の手順を実行するとします。

1. コミット・モードを SQL_AUTOCOMMIT_OFF に設定します。

   これは 39 ページの『カーソルと自動コミット』で説明している要件です。
2. 行セット・サイズを 5 に設定します。
3. 照会を実行して、n 行の結果セットを生成します。
4. SQLFetchScroll を使用して、5 行の最初の行セットをフェッチします。

サンプル結果セットを以下に示します。このサンプルでは、結果セットに含まれる列は 1 つだけです (varchar(32) として定義されています)。以下の表では、最初の列は、ドライバーによって内部で維持されている行番号を示します。 2 番目の列は、行の実際の値を示します。

表 10. サンプル結果セット

ドライバーにより内部に保管された行カウンター	**行の値**
1	Antony
2	Ben
3	Charlie
4	David
5	Edgar

ここで、アプリケーションが SQLSetPos を呼び出して、結果セットの 3 行目と 4 行目を Caroline および Debbie という名前で更新するとします。更新後、実際の行の値には以下のように Caroline と Debbie が含まれています。

表 11. サンプル結果セット

ドライバーにより内部に保管された行カウンター	**行の値**
1	Antony
2	Ben
3	Caroline
4	Debbie
5	Edgar

注: 場合によっては、SELECT ステートメントの結果セットが大きすぎてメモリーに入りきらないことがあります。ユーザーが結果セット内を前後にスクロールすると、ODBC ドライバーはメモリー内の一部の行を破棄して、別の行を読み取ることがあります。これによって、予期しない結果が生じることがあります。状況によっ

ては、以前に変更された行に対して、カーソルが (ディスクなどから) 元の値を再度読み込むと、カーソル内のデータの更新がいったん「消失」し、「再表示」するように見えることがあります。

ケース 2

ケース 2 は、位置付け操作の使用時の制限を示します。以下の例は、位置付け操作を使用した場合のカーソル動作と、位置更新がユーザーに可視でない場合を示します。

アプリケーションで以下の手順を実行するとします。

1. コミット・モードを SQL_AUTOCOMMIT_OFF に設定します。

   これは 39 ページの『カーソルと自動コミット』で説明している要件です。
2. 行セット・サイズを 5 に設定します。
3. 照会を実行して、n 行の結果セットを生成します。
4. SQLFetchScroll を使用して、5 行の最初の行セットをフェッチします。

   サンプル結果セットを以下に示します。このサンプルでは、最初の 2 つの行セットが示されます。結果セットに含まれる列は 1 つだけです (varchar(32) として定義されています)。以下の表では、最初の列は、ドライバーによって内部で維持されている行番号を示します。 2 番目の列は、行の実際の値を示します。

表 12. サンプル結果セット

ドライバーにより内部に保管された 行カウンター	行の値
1	Antony
2	Ben
3	Charlie
4	David
5	Edgar
6	Fred
7	Gough
8	Harry
9	Ivor
10	John

   上記の最初の 4 つのステップの後で、アプリケーションが SQLSetPos を呼び出して以下のタスクを実行するとします。
5. 結果セットの 3 行目を更新します。

6. SQLFetchScroll を呼び出して次の行セットにスクロールします。これによって、6 行目から 10 行目までを取得し、カーソルは 6 行目を指すことになります。
7. 行セットを 1 つ後方にスクロールして、最初の行セットに戻ります。これは、FETCH_PRIOR オプションを指定して SQLScrollFetch を呼び出すことによって行われます。

このようなタスクの実行後も、ステップ 5 で更新された 3 行目には、「ケース 1」のように更新された値ではなく、古い値が含まれています。ケース 2 の場合は、変更のコミット後に限り、更新された値が可視となります。ただし、 40 ページの『位置付けカーソル操作と SQL_AUTOCOMMIT_ON』のセクションで説明した SQL_AUTOCOMMIT_ON 設定時の予測不能な動作により、ブロック・カーソルおよび位置付け操作に関連する作業がすべて完了するまで、コミットを行うことはできません。

ブックマークの使用

ブックマークは、アプリケーションが行に戻るために使用する 32 ビットの値です。solidDB は、ブックマークをサポートしていません。

エラー・テキスト・フォーマット

SQLError によって返されるエラー・メッセージには、データ・ソースと、ODBC 接続のコンポーネントという 2 つの原因があります。エラー・テキストは、エラーの発行場所に応じて、特定のフォーマットを使用しなければなりません。

通常、データ・ソースは ODBC を直接サポートしていません。そのため、ODBC 接続のコンポーネントがデータ・ソースからエラー・メッセージを受信した場合、エラーの原因として、データ・ソースを識別する必要があります。また、エラーを受信したコンポーネントとして、自身を識別する必要もあります。

エラーの原因がコンポーネント自体にある場合は、エラー・メッセージでそれを説明しなければなりません。したがって、SQLError が返すエラー・テキストには、データ・ソースで発生したエラー用のものと、ODBC 接続の他のコンポーネントで発生したエラー用のもの、という 2 つの異なるフォーマットがあります。

データ・ソース以外で発生したエラーの場合は、エラー・テキストは以下のフォーマットを使用する必要があります。

```
[vendor_identifier][ODBC_component_identifier]
component_supplied_text
```

データ・ソース内で発生したエラーの場合は、エラー・テキストは以下のフォーマットでなければなりません。

```
[vendor_identifier][ODBC_component_identifier]
[data_source_identifier] data_source_supplied_text
```

以下の表に、各エレメントの意味を示します。

表 13. データ・ソース内のエラー

エレメント	意味
vendor_identifier	エラーが発生したコンポーネントのベンダー、またはデータ・ソースから直接エラーを受信したコンポーネントのベンダーを識別します。
ODBC_component_identifier	エラーが発生したコンポーネント、またはデータ・ソースから直接エラーを受信したコンポーネントを識別します。
data_source_identifier	データ・ソースを識別します。単一層のドライバーでは、一般的にこれはファイル・フォーマットです。複数層ドライバーの場合は DBMS 製品です。
component_supplied_text	ODBC コンポーネントによって生成されます。
data_source_supplied_text	データ・ソースによって生成されます。

注: エラー・テキスト内の大括弧 ([ ]) は、オプション項目を示しません。

サンプル・エラー・メッセージ

以下の例は、ODBC 接続の各種コンポーネントがどのようにエラー・メッセージ・テキストを生成し、solidDB が SQLError でどのようにアプリケーションにエラー・メッセージ・テキストを返すかを示したものです。

表 14. サンプル・エラー・メッセージ

SQLSTATE	エラー・メッセージ
01000	通常の警告
01S00	接続ストリング属性が無効
08001	クライアントが接続を確立できない

SQLSTATE の値は、5 文字からなるストリングです。最初の 2 文字はクラス値で、その後に 3 文字のサブクラス値が続きます。例えば 01000 の場合は、01 がクラス値で、000 がサブクラス値です。なお、000 のサブクラス値は、その SQLSTATE にはサブクラスがないことを意味することに注意してください。クラスとサブクラスの値は SQL-92 で定義されています。

表 15. SQLSTATE の値

クラス値	意味
01	警告を表し、SQL_SUCCESS_WITH_INFO の戻りコードを含みます。

表 15. SQLSTATE の値 (続き)

クラス値	意味
07、 08、 21、 22、 25、 28、 34、 3C、 3D、 3F、 40、 42、 44、 HY	SQL_ERROR の戻り値を含むエラーを表します。
IM	ODBC から派生した警告およびエラーを表します。

エラー・メッセージの処理

アプリケーションは、ユーザーに対して、SQLError で得たすべてのエラー情報、すなわち ODBC SQLSTATE、ネイティブ・エラー・コード、エラー・テキスト、およびエラーの原因を提供します。

アプリケーションはエラー・テキストを解析して、エラーの原因を識別する情報からそのテキストを分離することができます。エラーに基づき適切なアクションを行うこと、またはいくつかのアクションの選択肢をユーザーに提供することは、アプリケーションの責任です。

ODBC インターフェースは、ステートメント、トランザクション、および接続を終了し、ステートメント、接続、および環境の各ハンドルを解放する関数を提供しています。

トランザクションと接続の終了

ODBC インターフェースは、ステートメント、トランザクション、および接続を終了し、ステートメント・ハンドル (hstmt)、接続ハンドル (hdbc)、および環境ハンドル (henv) を解放する関数を提供しています。

ステートメント処理の終了

ステートメント・ハンドルに関連付けられたリソースを解放するには、以下のオプションを指定して、アプリケーションで SQLFreeStmt を呼び出します。

- SQL_CLOSE: カーソルがあればそれをクローズして、保留中の結果を破棄します。アプリケーションは後で再度ステートメント・ハンドルを使用できます。ODBC 3.51 では、SQLCloseCursor も使用できます。
- SQL_UNBIND: ステートメント・ハンドルに対して、SQLBindCol によってバインドされたすべての戻りバッファーを解放します。
- SQL_RESET_PARAMS: ステートメント・ハンドルに対して、SQLBindParameter によって要求されたすべてのパラメーター・バッファーを解放します。

SQLFreeHandle は、カーソルがあればそれをクローズし、保留中の結果を破棄し、ステートメント・ハンドルに関連付けられているすべてのリソースを解放するために使用します。

トランザクションの終了

アプリケーションは SQLEndTran を呼び出して、現行の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。

接続の終了

ドライバーおよびデータ・ソースへの接続を終了するには、アプリケーションで以下の手順を実行します。

1. SQLDisconnect を呼び出して接続を閉じます。その後、アプリケーションはハンドルを使用して同じデータ・ソースまたは別のデータ・ソースに再接続することができます。
2. SQLFreeHandle を呼び出して、接続ハンドルまたは環境ハンドルを解放し、ハンドルに関連付けられたすべてのリソースを解放します。

アプリケーションの構成

このセクションでは、アプリケーションの C 言語ソース・コードの例を 2 つ紹介します。1 つは静的 SQL 関数を使用して表を作成し、そこにデータを追加し、挿入されたデータを選択する例です。もう 1 つは対話式のアドホック照会処理の例です。

Microsoft は、ASCII データ用と Unicode データ用に 2 つのタイプのヘッダー・ファイルを提供しています。この例ではいずれの Microsoft ODBC ヘッダー・ファイルでも使用できます。

静的 SQL の例

以下の例では、アプリケーション内で SQL ステートメントを構成します。

```
/*****************************************************************
    サンプル名: Example1.c
    作成者    : IBM SOLID Information Technology Ltd.

    場所      : アプリケーションの構成-
                プログラマー・ガイド
    目的      : 静的 SQL 関数を使用して
                表を作成し、そこにデータを追加して、
                挿入されたデータを選択するサンプル例です。
*****************************************************************/
#if (defined(SS_UNIX) || defined(SS_LINUX))
#include <sqlunix.h>
#else
#include <windows.h>
#endif


#if SOLIDODBCAPI
#include <sqlucode.h>
#include <wchar.h>
#else
#include <sql.h>
#include <sqlext.h>
#endif

#include <stdio.h>
#include <test_assert.h>
```

```
#define MAX_NAME_LEN 50
#define MAX_STMT_LEN 100

/************************************************************
    関数名    : PrintError
    目的......: ハンドルに関連付けられたエラーを表示します。
************************************************************/
SQLINTEGER PrintError(SQLSMALLINT handleType,SQLHANDLE handle)
{
        SQLRETURN  rc = SQL_ERROR;
        SQLWCHAR   sqlState[6];
        SQLWCHAR   eMsg[SQL_MAX_MESSAGE_LENGTH];
        SQLINTEGER nError;

        rc = SQLGetDiagRecW(handleType, handle, 1,
             (SQLWCHAR *)&sqlState, (SQLINTEGER *)&nError,
             (SQLWCHAR *)&eMsg, 255, NULL);
        if (rc == SQL_SUCCESS || rc == SQL_SUCCESS_WITH_INFO) {
            printf("¥n¥t Error:%ls¥n",eMsg);
        }
        return(SQL_ERROR);
}


/************************************************************
        関数名: DrawLine
        目的  : 指定された文字 (chr) を指定された回数 (len)
                描画します。
************************************************************/
void DrawLine(SQLINTEGER len, SQLCHAR chr)
{
        printf("¥n");
        while(len > 0)  {
            printf("%c",chr);
            len--;
        }
        printf("¥n");

}


/************************************************************
    関数名: example1
    目的  : 指定されたデータ・ソースに接続し、
            SQL ステートメント・セットを実行します。
************************************************************/
SQLINTEGER example1(SQLCHAR *server, SQLCHAR *uid, SQLCHAR *pwd)
{
        SQLHENV     henv;
        SQLHDBC     hdbc;
        SQLHSTMT    hstmt;
        SQLRETURN   rc;

        SQLINTEGER  id;
        SQLWCHAR    drop[MAX_STMT_LEN];
        SQLCHAR     name[MAX_NAME_LEN+1];
        SQLWCHAR    create[MAX_STMT_LEN];
        SQLWCHAR    insert[MAX_STMT_LEN];
        SQLWCHAR    select[MAX_STMT_LEN];
        SQLINTEGER  namelen;


        /* 環境ハンドルおよび接続ハンドルを割り振ります。*/
        /* データ・ソースに接続します。*/
        /* ステートメント・ハンドルを割り振ります。*/

        rc = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE,
```

```
     &henv);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
    return(PrintError(SQL_HANDLE_ENV, henv));

rc = SQLSetEnvAttr(henv,SQL_ATTR_ODBC_VERSION,
     (SQLPOINTER)SQL_OV_ODBC3,SQL_NTS);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
    return(PrintError(SQL_HANDLE_ENV, henv));

rc = SQLAllocHandle(SQL_HANDLE_DBC,henv,&hdbc);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
    return(PrintError(SQL_HANDLE_ENV, henv));

rc = SQLConnect(hdbc, server, SQL_NTS, uid, SQL_NTS,
     pwd, SQL_NTS);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
    return(PrintError(SQL_HANDLE_DBC, hdbc));

rc = SQLAllocHandle(SQL_HANDLE_STMT,hdbc,&hstmt);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
    return(PrintError(SQL_HANDLE_DBC, hdbc));

    /* 表 'nameid' が存在すればそれをドロップし、なければ続行します。 */
wcscpy(drop, L"DROP TABLE NAMEID");
printf("¥n%ls", drop);
DrawLine(wcslen(drop), '-');

rc = SQLExecDirectW(hstmt, drop, SQL_NTS);
if (rc == SQL_ERROR)  {
    PrintError(SQL_HANDLE_STMT, hstmt);
    }

/* 作業をコミットします。*/
rc = SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
    return(PrintError(SQL_HANDLE_DBC, hdbc));

/* 表 nameid(id integer,name varchar(50)) を作成します。*/
wcscpy(create,
      L"CREATE TABLE NAMEID(ID INT,NAME VARCHAR(50))");
printf("¥n%ls",create);
DrawLine(wcslen(create),'-');

rc = SQLExecDirectW(hstmt,create,SQL_NTS);
if (rc == SQL_ERROR)
    return(PrintError(SQL_HANDLE_STMT, hstmt));

/* 作業をコミットします。*/
rc = SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
    return(PrintError(SQL_HANDLE_DBC, hdbc));

/* パラメーターを使用してデータを挿入します。*/
wcscpy(insert, L"INSERT INTO NAMEID VALUES(?,?)");
printf("¥n%ls", insert);
DrawLine(wcslen(insert), '-');

rc = SQLPrepareW(hstmt, insert, SQL_NTS);
if (rc == SQL_ERROR)
    return(PrintError(SQL_HANDLE_STMT, hstmt));

/* integer(id) データのバインディング*/
rc = SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT,
         SQL_C_LONG, SQL_INTEGER, 0, 0, &id, 0, NULL);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
    return(PrintError(SQL_HANDLE_DBC, hdbc));
```

```
/* char(name) データのバインディング*/
rc = SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT,
        SQL_C_CHAR, SQL_VARCHAR, 0, 0, &name,
        sizeof(name), NULL);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
    return(PrintError(SQL_HANDLE_DBC, hdbc));

id = 100;
strcpy(name, "SOLID");

rc = SQLExecute(hstmt);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
    return(PrintError(SQL_HANDLE_DBC, hdbc));

/* 作業をコミットします。*/
rc = SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
    return(PrintError(SQL_HANDLE_DBC, hdbc));

/* ステートメント・バッファーを解放します。*/
rc = SQLFreeStmt(hstmt, SQL_RESET_PARAMS);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
        return(PrintError(SQL_HANDLE_STMT, hstmt));

rc = SQLFreeStmt(hstmt, SQL_CLOSE);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
        return(PrintError(SQL_HANDLE_STMT, hstmt));

/* 表 nameid からデータを選択します。*/
wcscpy(select, L"SELECT * FROM NAMEID");
printf("¥n%ls", select);
DrawLine(wcslen(select), '-');

rc = SQLExecDirectW(hstmt, select, SQL_NTS);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
        return(PrintError(SQL_HANDLE_DBC, hdbc));

/* 出力データ用のバッファーをバインドします。*/
id = 0;
strcpy(name, "");

rc = SQLBindCol(hstmt, 1, SQL_C_LONG, &id, 0, NULL);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
    return(PrintError(SQL_HANDLE_DBC, hdbc));

rc = SQLBindCol(hstmt, 2, SQL_C_CHAR, &name,
     sizeof(name), &namelen);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
    return(PrintError(SQL_HANDLE_DBC, hdbc));

rc = SQLFetch(hstmt);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
    return(PrintError(SQL_HANDLE_DBC, hdbc));

printf("¥n Data ID      :%d", id);
printf("¥n Data Name    :%s(%d)¥n", name, namelen);

rc = SQLFetch(hstmt);
assert(rc == SQL_NO_DATA);


/* ステートメント・バッファーを解放します。*/
rc = SQLFreeStmt(hstmt, SQL_UNBIND);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
    return(PrintError(SQL_HANDLE_STMT, hstmt));
```

```
        rc = SQLFreeStmt(hstmt, SQL_CLOSE);
        if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
            return(PrintError(SQL_HANDLE_STMT, hstmt));

        /* ステートメント・ハンドルを解放します。 */
        rc = SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
        if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
            return(PrintError(SQL_HANDLE_STMT, hstmt));

        /* データ・ソースから切断します。*/
        rc = SQLDisconnect(hdbc);
        if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
            return(PrintError(SQL_HANDLE_DBC, hdbc));

        /* 接続ハンドルを解放します。*/
        rc = SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
        if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
            return(PrintError(SQL_HANDLE_DBC, hdbc));

        /* 環境ハンドルを解放します。*/
        rc = SQLFreeHandle(SQL_HANDLE_ENV,henv);
        if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
            return(PrintError(SQL_HANDLE_ENV, henv));

        return(0);
}


/*****************************************************************
        関数名: main
        目的  : すべての操作を制御します。
*****************************************************************/
void main(SQLINTEGER argc, SQLCHAR *argv[])
{
        puts("¥n¥t SOLID ODBC Driver 3.51:");
        puts("¥n¥t -Usage of static SQL functions");
        puts("¥n¥t ==============================");

        if (argc != 4){
            puts("USAGE: Example1 <DSN name> <username> <passwd>");
            exit(0);
            }
        else {
            example1(argv[1], argv[2], argv[3]);
            }
}
```

対話式アドホック照会のサンプル

以下の例では、アプリケーションが、結果をリトリーブする前に、結果セットの性質を判別する方法を示します。

```
/*****************************************************************
    サンプル名: Example2.c (アドホック照会処理)
    作成者    : IBM SOLID Information Technology Ltd.

    場所      : アプリケーションの構成-
                プログラマー・ガイド
    目的      : アプリケーションが、結果をリトリーブする前に、
                結果セットの性質を判別する方法を示します。

*****************************************************************/
#if (defined(SS_UNIX) || defined(SS_LINUX))
#include <sqlunix.h>
#else
#include <windows.h>
```

```
#endif

#if SOLIDODBCAPI
#include <sqlucode.h>
#include <wchar.h>
#else
#include <sql.h>
#include <sqlext.h>
#endif

#include <stdio.h>

#ifndef TRUE
#define TRUE 1
#endif

#define MAXCOLS 100
#define MAX_DATA_LEN 255

SQLHENV  henv;
SQLHDBC  hdbc;
SQLHSTMT hstmt;


/*****************************************************************
    関数名: PrintError
    目的  : ハンドルに関連付けられたエラーを表示します。
*****************************************************************/
SQLINTEGER PrintError(SQLSMALLINT handleType, SQLHANDLE handle)
{
    SQLRETURN rc = SQL_ERROR;
    SQLCHAR sqlState[6];
    SQLCHAR eMsg[SQL_MAX_MESSAGE_LENGTH];
    SQLINTEGER nError;

        rc = SQLGetDiagRec(handleType, handle, 1,
             (SQLCHAR *)&sqlState, (SQLINTEGER *)&nError,
             (SQLCHAR *)&eMsg, 255, NULL);
        if (rc == SQL_SUCCESS || rc == SQL_SUCCESS_WITH_INFO)  {
            printf("¥n¥t Error:%s¥n",eMsg);
        }
        return(SQL_ERROR);
}


/*****************************************************************
    関数名: DrawLine
    目的  : 指定された文字 (line) を指定された回数 (len)
            描画します。
*****************************************************************/
void DrawLine(SQLINTEGER len, SQLCHAR line)
{
        printf("¥n");
        while(len > 0) {
            printf("%c",line);
            len--;
        }
        printf("¥n");

}


/*****************************************************************
    関数名: example2
    目的  : 指定されたデータ・ソースに接続し、
            特定の SQL ステートメントを実行します。
*****************************************************************/
```

```
SQLINTEGER example2(SQLCHAR *sqlstr)
{
    SQLINTEGER i;

    SQLCHAR colname[32];
    SQLSMALLINT coltype;
    SQLSMALLINT colnamelen;
    SQLSMALLINT nullable;
    SQLINTEGER collen[MAXCOLS];
    SQLSMALLINT scale;
    SQLINTEGER outlen[MAXCOLS];
    SQLCHAR data[MAXCOLS][MAX_DATA_LEN];
    SQLSMALLINT nresultcols;
    SQLINTEGER rowcount, nRowCount=0, lineLength=0;
    SQLRETURN rc;


    printf("¥n%s",sqlstr);
    DrawLine(strlen(sqlstr),'=');

    /* SQL ステートメントを実行します。*/
    rc = SQLExecDirect(hstmt, sqlstr, SQL_NTS);
    if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
            return(PrintError(SQL_HANDLE_STMT, hstmt));


    /* どのような種類のステートメントだったかを調べます。もし、*/
    /* 結果の列がなければ、ステートメントは SELECT */
    /* ステートメントではありません。もし、影響を受ける行の数が */
    /* 0 より大きい場合、ステートメントはおそらく */
    /* UPDATE、INSERT、DELETE のいずれかのステートメントだったので、*/
    /* 影響を受ける行の数を出力します。もし、*/
    /* 影響を受ける行の数が 0 の場合は、ステートメントはおそらく */
    /* DDL ステートメントなので、操作が */
    /* 成功したことを出力し、操作をコミットします。*/
rc = SQLNumResultCols(hstmt, &nresultcols);
    if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
        return(PrintError(SQL_HANDLE_STMT, hstmt));

    if (nresultcols == 0) {
        rc = SQLRowCount(hstmt, &rowcount);
        if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)  {
            return(PrintError(SQL_HANDLE_STMT, hstmt));
            }
        if (rowcount > 0 ) {
            printf("%ld rows affected.¥n", rowcount);
            }
        else {
            printf("Operation successful.¥n");
            }

        rc = SQLEndTran(SQL_HANDLE_DBC, hdbc, SQL_COMMIT);
        if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
            return(PrintError(SQL_HANDLE_DBC, hdbc));

        }
    /* それ以外の場合は、結果セットの列名を表示し、*/
    /* display_size() 関数を使用して、*/
    /* 各データ型に必要な長さを計算します。*/
    /* 次に、列をバインドして、すべてのデータを */
    /* char に変換するように指定します。最後に、*/
    /* 各行をフェッチして出力し、必要に応じて */
    /* 切り捨てメッセージを出力します。*/
    else {
        for (i = 0; i < nresultcols; i++) {
            rc = SQLDescribeCol(hstmt, i + 1, colname,
                (SQLSMALLINT)sizeof(colname),
```

```
                &colnamelen, &coltype, &collen[i],
                &scale, &nullable);
            if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO){
                return(PrintError(SQL_HANDLE_STMT, hstmt));
                }
            /* 列名を出力します。 */
            printf("%s¥t", colname);
            rc = SQLBindCol(hstmt, i + 1, SQL_C_CHAR,
                data[i], sizeof(data[i]), &outlen[i]);
            if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO){
                return(PrintError(SQL_HANDLE_STMT, hstmt));
                }
            lineLength += 6 + strlen(colname);
            }

        DrawLine(lineLength-6,'-');

        while (TRUE) {
            rc = SQLFetch(hstmt);
            if (rc == SQL_SUCCESS || rc == SQL_SUCCESS_WITH_INFO){
                nRowCount++;
                for (i = 0; i < nresultcols; i++) {
                    if (outlen[i] == SQL_NULL_DATA) {
                        strcpy((char *)data[i], "NULL");
                        }
                    printf("%s¥t",data[i]);
                    }
                printf("¥n");
                }

            else {
                if (rc == SQL_ERROR)
                    PrintError(SQL_HANDLE_STMT, hstmt);
                break;
                }
            }
        printf("¥n¥tTotal Rows:%d¥n", nRowCount);
        }

    SQLFreeStmt(hstmt, SQL_UNBIND);
    SQLFreeStmt(hstmt, SQL_CLOSE);
    return(0);

}


/************************************************************
    関数名: main
    目的  : すべての操作を制御します。
************************************************************/
int __cdecl main(SQLINTEGER argc, SQLCHAR *argv[])
{
    SQLRETURN   rc;

    printf("¥n¥t SOLID ODBC Driver 3.51-Interactive");
    printf("¥n¥t ad-hoc Query Processing");
    printf("¥n¥t ================================¥n");

    if (argc != 4)  {
        puts("USAGE: Example2 <DSN name> <username> <passwd>");
        exit(0);
        }


    /* 環境ハンドルおよび接続ハンドルを割り振ります。*/
    /* データ・ソースに接続します。*/
    /* ステートメント・ハンドルを割り振ります。*/
```

```
    rc = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
    if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
        return(PrintError(SQL_HANDLE_ENV, henv));

    rc = SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION,
        (SQLPOINTER)SQL_OV_ODBC3, SQL_NTS);
    if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
        return(PrintError(SQL_HANDLE_ENV, henv));

    rc = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
    if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
        return(PrintError(SQL_HANDLE_ENV, henv));

    printf("¥n Connecting to %s¥n ", argv[1]);
    rc = SQLConnect(hdbc, argv[1], SQL_NTS, argv[2], SQL_NTS,
        argv[3], SQL_NTS);
    if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
        return(PrintError(SQL_HANDLE_DBC, hdbc));

    rc = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
    if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
        return(PrintError(SQL_HANDLE_DBC, hdbc));

    /* 以下の SQL ステートメントを実行します。*/
    example2("SELECT * FROM SYS_TABLES");
    example2("DROP TABLE TEST_TAB");
    example2("CREATE TABLE TEST_TAB(F1 INT, F2 VARCHAR)");
    example2("INSERT INTO TEST_TAB VALUES(10, 'SOLID')");
    example2("INSERT INTO TEST_TAB VALUES(20, 'MVP')");
    example2("UPDATE TEST_TAB SET F2='UPDATED' WHERE F1 = 20");
    example2("SELECT * FROM TEST_TAB");


    /* ステートメント・ハンドルを解放します。 */
    rc = SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
    if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
        return(PrintError(SQL_HANDLE_STMT, hstmt));

    /* データ・ソースから切断します。*/
    rc = SQLDisconnect(hdbc);
    if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
        return(PrintError(SQL_HANDLE_DBC, hdbc));

    /* 接続ハンドルを解放します。*/
    rc = SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
    if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
        return(PrintError(SQL_HANDLE_DBC, hdbc));

    /* 環境ハンドルを解放します。*/
    rc = SQLFreeHandle(SQL_HANDLE_ENV, henv);
    if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
        return(PrintError(SQL_HANDLE_ENV, henv));

    return(0);

}
```

アプリケーションのテストとデバッグ

Microsoft ODBC SDK では、アプリケーション開発のためのツールを提供しています。

以下のツールが含まれています。

- ODBC Test。この対話式ユーティリティーを使用すると、ドライバーのアドホック・テストおよび自動テストが可能です。 ODBC ドライバーの適合性の基本的な領域を対象とした、サンプル・テスト DLL (Quick Test) が含まれています。
- ODBC Spy。このデバッグ・ツールを使用すると、データ・ソースの情報の収集およびドライバーやアプリケーションのエミュレートが可能です。
- サンプル・アプリケーション。ソース・コードと makefile を含んでおり、以下のように使用します。
  - #define および ODBCVER で、アプリケーションのコンパイルに使用する ODBC のバージョンを指定します。ODBC 3.51 定数とプロトタイプを使用するには、インクルード・ファイルの提供前に、以下の行をアプリケーション・コードに追加してください。

    ```
    #define ODBCVER 0X0352
    ```
  - ASCII データの場合は、以下の標準的な Microsoft インクルード・ファイルを使用してください。

    SQL.H および SQLEXT.H
  - Unicode データの場合は、以下の Microsoft インクルード・ファイルを使用してください。

    SQLUCODE.H および WCHAR.H

ODBC SDK ツールについて詳しくは、「*Microsoft ODBC SDK ガイド (Microsoft ODBC SDK Guide)*」を参照してください。

3 solidDB Light Client の使用

このセクションでは、solidDB Light Client の使用方法について説明します。Light Client は、フットプリントの非常に小さいデータベース・クライアントのライブラリーであり、また ODBC API のサブセットであり、特にメモリー・リソースの制限された組み込みソリューションのインプリメント用に設計されています。solidDB Light Client を使用すると、軽量のクライアント・アプリケーションで solidDB データベースの能力を最大限に使用できます。

solidDB Light Client とは

solidDB Light Client ライブラリーは ODBC API (ODBC 1.0 コア) の 20 の関数からなるサブセットで、solidDB データベースにアクセスするアプリケーション開発者に完全な SQL 機能を提供します。

solidDB Light Client は、データベース接続の制御、SQL ステートメントの実行、結果セットのリトリーブ、トランザクションのコミット、その他の solidDB 機能のための関数を提供します。solidDB Light Client はメモリー容量の少ないターゲット環境に適しています。

solidDB Light Client の概要

solidDB Light Client の使用を開始するには、インストール手順とプラットフォーム固有の資料に従って、TCP/IP インフラストラクチャーをセットアップする必要があります。

開発環境のセットアップとサンプル・プログラムの作成

solidDB Light Client ライブラリーを使用したプログラムの作成は、通常の C/C++ プログラムの作成とまったく同じです。

以下のタスクを実行する必要があります。

- ライブラリー・ファイルをご使用のプロジェクトに挿入します。
- ヘッダー・ファイルをインクルードします。
- ソース・コードをコンパイルします。
- プログラムをリンクします。

最初の 2 つのタスクについて、以下のセクションで詳しく説明します。

プロジェクトへのライブラリー・ファイルの挿入

ご使用の開発環境の資料で、プログラムへのライブラリーのリンク方法を確認してください。正しい Light Client ライブラリーをプログラムにリンクします。

プラットフォームによっては、動的ライブラリーを使用するか静的ライブラリーを使用するかを選択します。アプリケーションをコンパイルしてリンクするときに、ライブラリーのコードがアプリケーションにリンクされる場合は、「静的」ライブ

ラリーです。アプリケーション・コードをロードして実行するときに、ライブラリーがアプリケーションにリンクされる場合は、「動的」ライブラリーです。

UNIX ベースのプラットフォームでは、ライブラリー名は以下のパターンになります。

```
slcPPPvv.EEE
```

ここで、

- 「slc」は、「solidDB Light Client」です。
- 「PPP」は、以下のようなプラットフォーム固有の ID です。
  - マルチスレッド Linux の場合は「l2x」、
  - Solaris 2.9 の場合は「s9x」、
  - HP-UX (PA-RISC 上) の場合は「h1x」のようになります。
- 「vv」はバージョン標識で、例えばバージョン 4.1 の場合は「41」のようになります。
- 「EEE」は、以下のようなプラットフォーム関連の拡張子です。
  - UNIX 共有オブジェクト・ファイル (動的) の場合は「.so」、
  - UNIX ライブラリー・ファイル (静的) の場合は「.a」です。

したがって、例えば `slcs8x41.a` は、Solaris 8 用の solidDB 4.1 Light Client ライブラリーです。

```
slc s8x 41.a
  ^^^ ^^^ ^^ ^^^
   |   |   |  |
   |   |   |  |
   |   |   |  |
   |   |   |  ------> .a は例えば Solaris 8 用の静的ライブラリー
   |   |   ---------> バージョン 4.1
   |   ------------> Solaris 8
   ----------------> Light Client ライブラリー用の solidDB のニーモニック
```

Microsoft Windows では、Light Client は静的ライブラリーとして提供されています。

ヘッダー・ファイルのインクルード

以下の行を Light Client プログラムに組み込む必要があります。

```
#include "cli0lcli.h"
```

必要なその他すべての Light Client ヘッダーを含むディレクトリーを、ご使用の開発環境のインクルード・ディレクトリー設定に挿入してください。

開発環境セットアップの検証

Light Client サンプル・プログラムを作成して、開発セットアップが正しいことを検証できます。これによって、コードを書かなくても、開発環境を検証できます。

ご使用の開発環境には以下のことが当てはまるという点に注意してください。

- Windows 環境では、TCP/IP サービスは、標準 DLL である `wsock32.dll` によって提供されます。これらのサービスをプロジェクトにリンクするには、`wsock32.lib` をリンカーの lib ファイル・リストに追加します。

- Windows 環境では、開発ツールによっては、標準 ODBC サービスを提供する odbc32.lib を、デフォルト・ライブラリーとしてプロジェクトにリンクする場合があります。 ODBC の関数は、名前とインターフェースが solidDB Light Client に似ているので、Light Client の代わりに ODBC を使用するようにプログラムがリンクされることがあります。 odbc32.lib をリンカーのファイル・リストから削除します。
- ChorusOS および VxWorks のターゲット・マシン上では、作業用 TCP/IP スタックが稼働しているカーネルを実行する必要があります。通常は、ping 要求にターゲット・マシンが応答することを確認して、これを検証できます。例えば、ターゲット・マシンの IP アドレスを 192.168.1.111 に構成した場合、LAN 上の別のワークステーションから以下のように「ping 192.168.1.111」を実行し、ターゲットがアクティブであることを証明する応答を確認します。

  ```
  C:¥>ping 192.168.1.111
  Pinging 192.168.1.111 with 32 bytes of data:
  Reply from 192.168.1.111: bytes=32 time=260ms TTL=62
  ```

  検証後、Light Client アプリケーションはそのターゲット・マシンで機能します。

サンプル・アプリケーションを使用したデータベースへの接続

solidDB Light Client ライブラリーを使用したデータベースへの接続の確立は、ODBC を使用した接続の確立と似ています。

アプリケーションは環境ハンドルを取得し、接続のためのスペースを割り振り、接続を確立する必要があります。サンプル・プログラムを実行して、ご使用の環境内の solidDB データベースに対する接続を取得できるかどうかを確認します。

solidDB への接続の確立

以下のコードは、マシン 192.168.1.111 で実行中でポート 1313 で TCP/IP を listen している solidDB データベースへの接続を確立します。パスワードが DBA のユーザー・アカウント DBA がデータベース内で定義されています。

```
HENV henv;          /* 環境オブジェクトを指すポインターです。*/
HDBC hdbc;          /* データベース接続オブジェクトを指すポインターです。*/
RETCODE rc;         /* 戻りコード用の変数です。*/

rc = SQLAllocEnv(&henv);
if (SQL_SUCCESS != rc)
{
    printf("SQLAllocEnv fails.¥n");
    return;
}

rc = SQLAllocConnect(&henv, &hdbc);
if (SQL_SUCCESS != rc)
{
    printf("SQLAllocConnect fails.¥n");
    return;
}

rc = SQLConnect(hdbc, (UCHAR*)"192.168.1.111 1313", SQL_NTS,
(UCHAR*)"DBA", SQL_NTS, (UCHAR*)"DBA", SQL_NTS);
if (SQL_SUCCESS != rc)
{
    printf("SQLConnect fails.¥n");
    return;
}
```

上記で確立された接続は、以下のコードを使用して切断できます。読みやすくするため、戻りコードの検査は含んでいません。

```
SQLDisconnect(hdbc);
SQLFreeConnect(hdbc);SQLFreeEnv(henv);
```

solidDB Light Client での SQL ステートメントの実行

このセクションでは、SQL を使用した基本的なデータベース操作の方法を簡単に説明します。

以下の操作について説明します。

- solidDB Light Client によるステートメントの実行
- 結果セットの読み取り
- トランザクションと自動コミット・モード
- データベース・エラーの処理

solidDB Light Client によるステートメントの実行

以下のコードでは単純 SQL ステートメントを実行します。

```
INSERT INTO TESTTABLE (I,C) VALUES (100, 'HUNDRED');
```

このコードでは、有効な HENV henv および有効な HDBC henv が存在することと、RETCODE 型の変数 rc が定義されることが必要です。また、このコードでは、列 I および C を持つ表 TESTTABLE がデータベース内に存在することが必要です。

```
rc = SQLAllocStmt(hdbc, &hstmt);

if (SQL_SUCCESS != rc)
{
    printf("SQLAllocStmt failed ¥n");
}
rc = SQLExecDirect(hstmt,
    (UCHAR*)"INSERT INTO TESTTABLE (I,C) VALUES (100, 'HUNDRED')",
    SQL_NTS);
if (SQL_SUCCESS != rc)
{
    printf("SQLExecDirect failed ¥n");
}

rc = SQLTransact(SQL_NULL_HENV, hdbc, SQL_COMMIT);
if (SQL_SUCCESS != rc)
{
    printf("SQLTransact failed ¥n");
}

rc = SQLFreeStmt(hstmt, SQL_DROP);
if (SQL_SUCCESS != rc)
{
    printf("SQLFreeStmt failed ¥n");
}
```

パラメーターを指定したステートメントの実行

以下のコード例では、異なるパラメーター値を指定して、単純ステートメント INSERT INTO TESTTABLE (I,C) VALUES (?,?) を複数回実行する準備をします。Light Client は ODBC のようなパラメーター・バインディングは行わない点に注意

してください。その代わりに、SQLSetParamValue 関数を使用してパラメーター値を割り当てる必要があります。以下のような変数定義が必要です。

```
#include "cli0lcli"
int i;
char buf[255];
SDWORD dwPar;
```

上記と同様に、このコードでは、有効な HENV henv および有効な HDBC henv が存在することと、RETCODE 型の変数 rc が定義されること、および列 I および C を持つ表 TESTTABLE がデータベース内に存在することが必要です。

```
rc = SQLAllocStmt(hdbc, &hstmt);

if (SQL_SUCCESS != rc) {
    printf("Alloc statement failed. ¥n");
}

rc = SQLPrepare(hstmt,
    (UCHAR*)"INSERT INTO TESTTABLE(I,C)VALUES   (?,?)", SQL_NTS);

    if (SQL_SUCCESS != rc) {
         printf("Prepare failed. ¥n");
    }

    for (i=1; i<100; i++)
    {
    dwPar = i;
    sprintf(buf,"line%i",i);

    rc = SQLSetParamValue(
    hstmt,1,SQL_C_LONG,SQL_INTEGER,0,0,&dwPar,NULL );
    if (SQL_SUCCESS != rc) {
        printf("(SetParamValue 1 failed) ¥n");
        return 0;
    }
    rc = SQLSetParamValue(
    hstmt,2,SQL_C_CHAR,SQL_CHAR,0,0,buf,NULL );
    if (SQL_SUCCESS != rc) {
        printf("(SetParamValue 1 failed) ¥n");
        return 0;
    }

    rc = SQLExecute(hstmt);

    if (SQL_SUCCESS != rc) {
        printf("SQLExecute failed ¥n");
    }
}
rc = SQLFreeStmt(hstmt,SQL_DROP);
if (SQL_SUCCESS != rc) {
    printf("SQLFreeStmt failed. ¥n");
}
```

結果セットの読み取り

以下に抜粋したコードは SQL ステートメント SELECT I,C FROM TESTTABLE を準備し、実行して、データベースが返すすべての行をフェッチします。以下のコード例では、*rc*、henv、hstmt、および henv に対する有効な定義が必要です。

```
    rc = SQLAllocStmt(hdbc, &hstmt);

    if (SQL_SUCCESS != rc) {
         printf("SQLAllocStmt failed. ¥n");
         }
```

```
    rc = SQLPrepare(hstmt,(UCHAR*)"SELECT I,C FROM TESTTABLE", SQL_NTS);
    if (SQL_SUCCESS != rc) {
         printf("SQLPrepare failed. ¥n");
    }

    rc = SQLExecute(hstmt);

    if (SQL_SUCCESS != rc) {
         printf("SQLExecute failed. ¥n");
    }

    rc = SQLFetch(hstmt);

if ((SQL_SUCCESS != rc) &&SQL_NO_DATA_FOUND != rc)) {
    printf("SQLFetch returned an unexpected error code . ¥n");
}

while (SQL_NO_DATA_FOUND != rc)
{
    rc = SQLGetCol(hstmt, 1, SQL_C_LONG, &lbuf, sizeof(lbuf), NULL);
    if (rc == SQL_SUCCESS)
    {
        printf("LC_SQLGetCol(1) returns %d ¥n", lbuf);
        }
        else printf("Error in SQLGetCol(1) ¥n");
        rc = SQLGetCol(hstmt, 2, SQL_C_CHAR, buf, sizeof(buf), NULL);
        if (rc == SQL_SUCCESS)
        {
            printf("SQLGetCol(2) returns %s ¥n",buf);
         }
         else printf("Error in SQL_GetCol(2) ¥n");

         rc = SQLFetch(hstmt);
    }

    rc = SQLFreeStmt(hstmt,SQL_DROP);
    if (SQL_SUCCESS != rc)
    {
         printf("SQLFreeStmt failed. ");
    }
```

結果セットの処理には、以下の Light Client API 関数を使用すると便利です。

- SQLDescribeCol
- SQLGetCursorName
- SQLNumResultCols
- SQLSetCursorName

トランザクションと自動コミット・モード

すべての solidDB Light Client 接続では、自動コミット・オプションがオフに設定されています。Light Client ではこのオプションをオンに設定する方法がありません。各トランザクションを明示的にコミットする必要があります。

トランザクションをコミットするには、SQLTransact 関数を以下のように呼び出します。

```
rc = SQLTransact(SQL_NULL_HENV, hdbc, SQL_COMMIT);
```

トランザクションをロールバックするには、SQLTransact を以下のように呼び出します。

```
rc = SQLTransact(SQL_NULL_HENV, hdbc, SQL_ROLLBACK);
```

データベース・エラーの処理

Light Client API 関数が SQL_ERROR または SQL_SUCCESS_WITH_INFO を返すと、SQLError 関数を呼び出すことにより、エラーまたは警告に関してより詳細な情報を取得できます。表 TESTTABLE が定義されていないデータベースに対して以下のコードを実行すると、適切なエラー情報が生成されます。

通常どおり、このコードでは、有効な HENV henv および有効な HDBC henv が存在することと、RETCODE 型の変数 rc が定義されることが必要です。

```
rc = SQLPrepare(hstmt,(UCHAR*)"SELECT I,C FROM TESTTABLE", SQL_NTS);
if (SQL_SUCCESS != rc)
{
    char buf[255];
    RETCODE rc;

    char szSQLState[255];
    char szErrorMsg[255];
    SDWORD nativeerror = 0;
    SWORD maxerrmsg = 0;

    memset(szSQLState,0,sizeof(szSQLState));
    memset(szErrorMsg,0,sizeof(szErrorMsg));

    rc = SQLError(
    SQL_NULL_HENV, hdbc, hstmt, (UCHAR*)szSQLState, &nativeerror,
    (UCHAR*)szErrorMsg, sizeof(szErrorMsg), &maxerrmsg);

    if (rc == SQL_ERROR)
    {
        printf("SQLError failed ¥n.");
    }
    else
    {
            printf("Error information dump begins:-------------¥n");
            printf("SQLState '%s' ¥n",szSQLState);
    printf("nativeerror %i ¥n", nativeerror);
    printf("Errormsg '%s' ¥n", szErrorMsg);
printf("maxerrmsg %i ¥n", maxerrmsg);
printf("Error information dump ends:---------------¥n");
    }
}
```

solidDB Light Client の使用に関する考慮事項

このセクションには、solidDB Light Client の使用に関する重要な情報と制限事項を記載します。

データ・フェッチにおけるネットワーク・トラフィック

solidDB Light Client の通信は、solidDB の RowsPerMessage 設定をサポートしていません。Light Client が SQLFetch を呼び出すたびに、クライアントとサーバーの間でネットワーク・メッセージが送信されます。大量のデータをフェッチする場合は、これによってパフォーマンスに影響が及びます。

Unicode および ODBC のサポート

solidDB Light Client は Unicode および ODBC 3.51 API 機能を処理しません。 3.5 より前の ODBC API だけがサポートされています。

BIGINT はサポートされない

solidDB Light Client は BIGINT データ型をサポートしていません。

ODBC に精通したプログラマーのための注意点

Light Client API では提供されていない ODBC 関数を使用している場合は、標準の ODBC データベース・インターフェースから solidDB Light Client へのマイグレーションには、多少のプログラミングが必要になります。大まかなマイグレーション手順は以下のとおりです。

1. ご使用のアプリケーションによる ODBC の使用方法を検討し、Light Client API 機能だけで十分かどうかを評価します。独自のコードに以下のような小さい変更を加えることができます。
   - ODBC 拡張レベル 1 の関数の呼び出しを、ODBC コア・レベルの関数の呼び出しに変換する。
   - SQLBindParameter なしにアプリケーションを書き換える。
2. solidDB Light Client のサンプルを使用して、ご使用の環境を検証します。
3. 独自のコード内の ODBC の呼び出しを変更し、プログラムを再作成し、テストします。

solidDB Light Client 関数のサマリー

このセクションには、ODBC API のサブセットである solidDB Light Client API の関数をリストします。

実際の関数の説明については、『*solidDB Light Client 関数リファレンス*』セクションのトピックを参照してください。

注: solidDB Light Client は、パラメーター値設定用 (SQLBindParameter など) またはデータ・バインディング用 (SQLBindCol など) の ODBC 拡張レベル機能を提供していません。代わりに solidDB Light Client は SAG CLI 準拠の関数として、パラメーター値の設定用に SQLSetParamValue、結果セットからのデータの読み取り用に SQLGetCol を提供しています。これらの関数について詳しくは、85 ページの『非 ODBC solidDB Light Client 関数』を参照してください。

関数のサマリー

このセクションでは、solidDB Light Client が提供している関数のサマリーについて説明します。

solidDB Light Client API の使用方法を示す完全なプログラム例は、66 ページの『solidDB Light Client サンプル』を参照してください。

表 16. 関数のサマリー

タスク	関数
データ・ソースへの接続	70 ページの『SQLAllocEnv (ODBC 1.0、コア)』 69 ページの『SQLAllocConnect (ODBC 1.0、コア)』 71 ページの『SQLConnect (ODBC 1.0、コア)』
SQL ステートメントの準備	70 ページの『SQLAllocStmt (ODBC 1.0、コア)』 83 ページの『SQLPrepare (ODBC 1.0、コア)』 33 ページの『ODBC API に対する solidDB 拡張機能』の SqlSetParamValue この関数は、solidDB Light Client に固有のものである点に注意してください。この関数について詳しくは、この表の後のセクションを参照してください。 84 ページの『SQLSetCursorName (ODBC 1.0、コア)』 79 ページの『SQLGetCursorName (ODBC 1.0、コア)』
要求のサブミット	76 ページの『SQLExecute (ODBC 1.0、コア)』 75 ページの『SQLExecDirect (ODBC 1.0、コア)』
結果と結果に関する情報のリトリーブ	83 ページの『SQLRowCount (ODBC 1.0、コア)』 82 ページの『SQLNumResultCols (ODBC 1.0、コア)』 72 ページの『SQLDescribeCol (ODBC 1.0、コア)』 85 ページの『非 ODBC solidDB Light Client 関数』の SqlGetCol この関数は ODBC 準拠の関数 SQLGetData と同じである点に注意してください。 76 ページの『SQLFetch (ODBC 1.0、コア)』 79 ページの『SQLGetData (ODBC 1.0、レベル 1)』 この関数は対応する SAG CLI 関数 SQLGetCol と同じである点に注意してください。 74 ページの『SQLError (ODBC 1.0、コア)』
ステートメントの終了	78 ページの『SQLFreeStmt (ODBC 1.0、コア)』 85 ページの『SQLTransact (ODBC 1.0、コア)』

表 16. 関数のサマリー (続き)

タスク	関数
接続の終了	74 ページの『SQLDisconnect (ODBC 1.0、コア)』 71 ページの『SQLConnect (ODBC 1.0、コア)』 77 ページの『SQLFreeEnv (ODBC 1.0、コア)』

solidDB Light Client サンプル

このセクションには、solidDB Light Client API の使用方法を示すプログラム例が示されています。

solidDB Light Client サンプル 1

```
#include "sample1.h"

/************************************************************************
 *
 * ファイル:           SAMPLE1.C
 *
 * 説明:    Solid Light Client API のサンプル・プログラム
 *
 * 作成者:         Solid
 *
 *
 * Solid Light Client サンプル・プログラムは以下を実行します。
 *
 * 1. 十分な接続情報を含むのに必要なだけの入力パラメーターがあることを
 *    確認します。
 * 2. HENV および HDBC の各オブジェクトにメモリーを割り振って、
 *    Light Client による Solid への接続を準備します。
 * 3. Light Client ライブラリーを使用して Solid に接続します。
 * 4. SOLID システム表のいずれかからデータを読み取るために、
 *    1 つの照会用のステートメント 'SELECT
 *    TABLE_SCHEMA,TABLE_NAME,TABLE_TYPE FROM TABLES' を作成します。
 * 5. 照会を実行します。
 * 6. 結果セットのすべての行をフェッチして出力します。
 * 7. 正常に接続を閉じます。
 *
 * ************************************************************************/
void __cdecl main(int argc, char *argv[])
{
  HENV henv;      /* 環境オブジェクトを指すポインターです。*/
  HDBC hdbc;      /* データベース接続オブジェクトを指すポインターです。*/
  RETCODE rc;     /* 戻りコード用の変数です。*/
  HSTMT hstmt;    /* データベース・ステートメント・オブジェクトを指す
                     ポインターです。*/
  char buf[255];  /* db から取得するデータ用のバッファーです。*/
  char buf2[255]; /* 作成する出力可能な行用のバッファーです。*/
  int iCount = 0; /* フェッチする行のカウンターです。          */


  /* 1. 十分な接続情報を含むのに必要なだけの入力パラメーターがあることを */
  /*    確認します。                                                    */
if (argc != 4)
  {
    printf("Proper usage ¥"connect string¥" uid pwd ¥n");
    printf("argc %i ¥n",argc);
    return;
  }
```

```
  printf("Will connect SOLID at %s with uid %s and pwd%s.¥n", argv[1],
    argv[2], argv[3]);


  /* 2. HENV および HDBC の各オブジェクトのメモリーを割り振って、    /*
  /*    Light Client による SOLID への接続を準備します。*/
rc = SQLAllocEnv(&henv);
  if (SQL_SUCCESS != rc)
  {
    printf("SQLAllocEnv fails.¥n");
    return;
  }

  rc = SQLAllocConnect(henv, &hdbc);
  if (SQL_SUCCESS != rc)
  {
    printf("SQLAllocConnect fails.¥n");
    return;
  }

  /* 3. Light Client ライブラリーを使用して SOLID に接続します。*/
  rc = SQLConnect(hdbc, (UCHAR*)argv[1], SQL_NTS, (UCHAR*)argv[2],
    SQL_NTS, (UCHAR*)argv[3], SQL_NTS);
  if (SQL_SUCCESS != rc)
  {
    printf("SQLConnect fails.¥n");
    return;
  }
  else printf("Connect ok.¥n");


  /* 4. 1 つの照会用のステートメントを作成します。                    */
  /*  SOLID システム表のいずれかからデータを読み取るためのものです。*/
rc = SQLAllocStmt(hdbc, &hstmt);
  if (SQL_SUCCESS != rc) {
    printf("SQLAllocStmt failed. ¥n");
   }

  rc = SQLPrepare(hstmt,
    (UCHAR*)"SELECT TABLE_SCHEMA,TABLE_NAME,TABLE_TYPE FROM TABLES",
   SQL_NTS);

  if (SQL_SUCCESS != rc) {
    printf("SQLPrepare failed. ¥n");
  }
  else printf("SQLPrepare succeeded. ¥n");


  /* 5. 照会を実行します。*/
  rc = SQLExecute(hstmt);
  if (SQL_SUCCESS != rc) {
    printf("SQLExecute failed. ¥n");
  }
  else printf("SQLExecute succeeded. ¥n");


  /* 6. 結果セットのすべての行をフェッチし、出力します。*/
  rc = SQLFetch(hstmt);
  if ((SQL_SUCCESS != rc) && (SQL_NO_DATA_FOUND != rc)) {
    printf("SQLFetch returned an unexpected error code . ¥n");
  }
  else printf("Starting to fetch data.¥n");

  while (SQL_NO_DATA_FOUND != rc)
  {
    iCount++;
    sprintf(buf2, "Row %i :", iCount);
```

```
    rc = SQLGetCol(hstmt, 1, SQL_C_CHAR, buf, sizeof(buf), NULL);
    if (rc == SQL_SUCCESS)
    {
      strcat(buf2, buf);
      strcat(buf2, ",");
    }
    else printf("Error in SQL_GetCol(1) ¥n");

    rc = SQLGetCol(hstmt, 2, SQL_C_CHAR, buf, sizeof(buf), NULL);
    if (rc == SQL_SUCCESS)
    {
      strcat(buf2, buf);
      strcat(buf2, ",");
    }
    else printf("Error in SQL_GetCol(2) ¥n");

    rc = SQLGetCol(hstmt, 3, SQL_C_CHAR, buf, sizeof(buf), NULL);
    if (rc == SQL_SUCCESS)
    {
      strcat(buf2, buf);
    }
    else printf("Error in SQL_GetCol(3) ¥n");

    printf("%s ¥n", buf2);


    rc = SQLFetch(hstmt);
  }

  rc = SQLFreeStmt(hstmt, SQL_DROP);
  if ((SQL_SUCCESS != rc))
  {
    printf("SQLFreeStmt failed. ");
  }

  /* 7. 正常に接続を閉じます。                          */
  SQLDisconnect(hdbc);
  SQLFreeConnect(hdbc);  SQLFreeEnv(henv);
  printf("Sample program ends successfully.¥n");

}
```

solidDB Light Client サンプル 2

```
#ifndef SAMPLE1_H
#define SAMPLE1_H

/*********************************************************************
 *
 * ファイル:           SAMPLE1.H
 *
 * 説明:    Solid Light Client API ヘッダー・ファイルのサンプル・プログラム
*
 * 作成者:         Solid
 *
 *  ********************************************************************
/


#include <stdio.h>
#include <string.h>

#include "cli0lcli.h"

#endif
```

solidDB Light Client サンプル 3

```
C:¥solid¥lcli¥samples>sample1 "fb1 1313" DBA DBA
Will connect Solid at fb1 1313 with uid DBA and pwd DBA.
Connect ok.
SQLPrepare succeeded.
SQLExecute succeeded.
Starting to fetch data.
Row 1 :_SYSTEM,SYS_TABLES,BASE TABLE
Row 2 :_SYSTEM,SYS_COLUMNS,BASE TABLE
Row 3 :_SYSTEM,SYS_USERS,BASE TABLE
Row 4 :_SYSTEM,SYS_UROLE,BASE TABLE
Row 5 :_SYSTEM,SYS_RELAUTH,BASE TABLE
Row 6 :_SYSTEM,SYS_ATTAUTH,BASE TABLE
Row 7 :_SYSTEM,SYS_VIEWS,BASE TABLE
Row 8 :_SYSTEM,SYS_KEYPARTS,BASE TABLE
Row 9 :_SYSTEM,SYS_KEYS,BASE TABLE
Row 10 :_SYSTEM,SYS_CARDINAL,BASE TABLE
Row 11 :_SYSTEM,SYS_INFO,BASE TABLE
Row 12 :_SYSTEM,SYS_SYNONYM,BASE TABLE
Row 13 :_SYSTEM,TABLES,VIEW
Row 14 :_SYSTEM,COLUMNS,VIEW
Row 15 :_SYSTEM,SQL_LANGUAGES,BASE TABLE
Row 16 :_SYSTEM,SERVER_INFO,VIEW
Row 17 :_SYSTEM,SYS_TYPES,BASE TABLE
Row 18 :_SYSTEM,SYS_FORKEYS,BASE TABLE
Row 19 :_SYSTEM,SYS_FORKEYPARTS,BASE TABLE
Row 20 :_SYSTEM,SYS_PROCEDURES,BASE TABLE
Row 21 :_SYSTEM,SYS_TABLEMODES,BASE TABLE
Row 22 :_SYSTEM,SYS_EVENTS,BASE TABLE
Row 23 :_SYSTEM,SYS_SEQUENCES,BASE TABLE
Row 24 :_SYSTEM,SYS_TMP_HOTSTANDBY,BASE TABLE
Sample program ends successfully.
```

solidDB Light Client 関数リファレンス

以下のトピックでは、solidDB Light Client がサポートしている各 ODBC 関数をアルファベット順に説明します。各関数は、C プログラミング言語関数として定義されています。

SQLAllocConnect (ODBC 1.0、コア)

SQLAllocConnect は、henv で識別される環境内の接続ハンドル用にメモリーを割り振ります。

構文

```
RETCODE SQLAllocConnect(henv, phenv)
```

SQLAllocConnect 関数は以下の引数を受け入れます。

表 17. SQLAllocConnect の引数

型	引数	使用法	説明
henv	henv	入力	環境ハンドル
HDBC FAR *	phenv	出力	接続ハンドルのストレージを指すポインター

戻り値

SQL_SUCCESS、 SQL_SUCCESS_WITH_INFO、 SQL_ERROR、 または SQL_INVALID_HANDLE

SQLAllocConnect は、SQL_ERROR を返す場合、phenv で参照される henv を SQL_NULL_HDBC に設定します。追加情報を取得するために、アプリケーションは、指定した henv を使用し、さらに henv と hstmt をそれぞれ SQL_NULL_HDBC と SQL_NULL_HSTMT に設定して、SQLError を呼び出すことができます。

SQLAllocEnv (ODBC 1.0、コア)

SQLAllocEnv は、環境ハンドル用にメモリーを割り振り、アプリケーションで使用するために ODBC コール・レベル・インターフェースを初期化します。アプリケーションは、他の ODBC 関数を呼び出す前に、SQLAllocEnv を呼び出す必要があります。

構文

```
RETCODE SQLAllocEnv(phenv)
```

SQLAllocEnv 関数は以下の引数を受け入れます。

表 18. SQLAllocEnv の引数

型	引数	使用法	説明
HENV FAR *	phenv	出力	環境ハンドルのストレージを指すポインター

戻り値

SQL_SUCCESS または SQL_ERROR

SQLAllocEnv は、SQL_ERROR を返す場合、phenv で参照される henv を SQL_NULL_HENV に設定します。この場合、アプリケーションは、メモリー割り振りエラーが発生したことを想定できます。

SQLAllocStmt (ODBC 1.0、コア)

SQLAllocStmt は、ステートメント・ハンドル用にメモリーを割り振り、そのステートメント・ハンドルを henv で指定される接続に関連付けます。アプリケーションは、SQL ステートメントをサブミットする前に、SQLAllocStmt を呼び出す必要があります。

構文

```
RETCODE SQLAllocStmt(henv, phstmt)
```

SQLAllocStmt 関数は以下の引数を受け入れます。

表 19. SQLAllocStmt の引数

型	引数	使用法	説明
HDBC	henv	入力	接続ハンドル
HSTMT FAR *	phstmt	出力	ステートメント・ハンドルのストレージを指すポインター

戻り値

SQL_SUCCESS、 SQL_SUCCESS_WITH_INFO、 SQL_INVALID_HANDLE、 または SQL_ERROR

SQLAllocStmt は、SQL_ERROR を返す場合、phstmt で参照される hstmt を SQL_NULL_HSTMT に設定します。これでアプリケーションは、henv と SQL_NULL_HSTMT で SQLError を呼び出すことにより、追加情報を取得できます。

SQLConnect (ODBC 1.0、コア)

SQLConnect は、ドライバーをロードし、データ・ソースへの接続を確立します。接続ハンドルは、状況、トランザクション状態、およびエラー情報を含めて、接続に関するすべての情報のストレージを参照します。

構文

```
RETCODE SQLConnect(henv, szDSN, cbDSN, szUID, cbUID, szAuthStr, cbAuthStr)
```

SQLConnect 関数は以下の引数を受け入れます。

表 20. SQLConnect の引数

型	引数	使用法	説明
HDBC	henv	入力	接続ハンドル
UCHAR FAR *	szDSN	入力	データ・ソース名
SWORD	cbDSN	入力	szDSN の長さ
UCHAR FAR *	szUID	入力	ユーザー ID
SWORD	*cbUID*	入力	szUID の長さ
UCHAR FAR *	szAuthStr	入力	認証ストリング (一般にはパスワード)
SWORD	cbAuthStr	入力	szAuthStr の長さ

戻り値

SQL_SUCCESS、 SQL_SUCCESS_WITH_INFO、 SQL_ERROR、 または SQL_INVALID_HANDLE

SQLDescribeCol (ODBC 1.0、コア)

SQLDescribeCol は、結果セット内の 1 つの列に関して、列名、型、精度、位取り、および NULL 可能性からなる結果記述子を返します。ブックマーク列 (列 0) に関する情報を返すのに使用することはできません。

構文

```
RETCODE SQLDescribeCol(hstmt, icol, szColName,
cbColNameMax, pcbColName, pfSqlType, pcbColDef,
pibScale, pfNullable)
```

SQLDescribeCol 関数は以下の引数を受け入れます。

表 21. SQLDescribeCol の引数

型	引数	使用法	説明
HSTMT	hstmt	入力	ステートメント・ハンドル
UWORD	icol	入力	結果データの列番号。1 から順に左から右へ番号が付けられます。
UCHAR FAR *	szColName	出力	列名のストレージを指すポインター。列に名前がない場合、または列名を判断できない場合には、ドライバーは空ストリングを返します。
SWORD	cbColNameMax	入力	szColName バッファーの最大長
SWORD FAR *	pcbColName	出力	szColName で返すことのできるバイトの総数 (NULL 終了バイトを除く)。返すことのできるバイトの数が cbColNameMax 以上の場合には、szColName 内の列名は、cbColNameMax - 1 バイトに切り捨てられます。

表 21. SQLDescribeCol の引数 (続き)

型	引数	使用法	説明
SWORD FAR *	pfSqlType	出力	列の SQL データ型。以下の値のいずれかを取る必要があります。 SQL_BIGINT (注: solidDB Light Client は、BIGINT/SQL_BIGINT をサポートしません。) SQL_BINARY SQL_BIT (注: solidDB、solidDB Light Client、solidDB ODBC ドライバー、および solidDB JDBC ドライバーは、BIT/SQL_BIT をサポートしません。) SQL_CHAR SQL_DATE SQL_DECIMAL SQL_DOUBLE SQL_FLOAT SQL_INTEGER SQL_LONGVARBINARY SQL_LONGVARCHAR SQL_NUMERIC SQL_REAL SQL_SMALLINT SQL_TIME SQL_TIMESTAMP SQL_TINYINT SQL_VARBINARY SQL_VARCHAR または、ドライバー固有の SQL データ型。データ型を判断できない場合、ドライバーは 0 を返します。 詳しくは、 278 ページの『SQL データ型』を参照してください。ドライバー固有の SQL データ型については、ドライバーの資料を参照してください。

表 21. *SQLDescribeCol* の引数 (続き)

型	引数	使用法	説明
UDWORD FAR *	pcbColDef	出力	データ・ソース上の列の精度。精度を判断できない場合、ドライバーは 0 を返します。
SWORD FAR *	pibScale	出力	データ・ソース上の列の位取り。位取りを判断できないか、適用されない場合、ドライバーは 0 を返します。
SWORD FAR *	pfNullable	出力	列に NULL 値が許可されるかどうかを示します。以下の値のいずれかです。 • SQL_NO_NULLS: 列に NULL 値が許可されません。 • SQL_NULLABLE: 列に NULL 値が許可されます。 • SQL_NULLABLE_UNKNOWN: ドライバーは、列に NULL 値が許可されるかどうか判断できません。

戻り値

SQL_SUCCESS、 SQL_SUCCESS_WITH_INFO、 SQL_STILL_EXECUTING、 SQL_ERROR、 または SQL_INVALID_HANDLE

SQLDisconnect (ODBC 1.0、コア)

SQLDisconnect は、特定の接続ハンドルに関連付けられた接続を閉じます。

構文

```
RETCODE SQLDisconnect(henv)
```

SQLDisconnect 関数は以下の引数を受け入れます。

表 22. *SQLDisconnect* の引数

型	引数	使用法	説明
HDBC	henv	入力	接続ハンドル

戻り値

SQL_SUCCESS、 SQL_SUCCESS_WITH_INFO、 SQL_ERROR、 または SQL_INVALID_HANDLE

SQLError (ODBC 1.0、コア)

SQLError は、エラー情報または状況情報を返します。

構文

```
RETCODE SQLError(henv, henv, hstmt, szSqlState,
pfNativeError, szErrorMsg, cbErrorMsgMax, pcbErrorMsg)
```

SQLError 関数は以下の引数を受け入れます。

表 23. SQLError の引数

型	引数	使用法	説明
HENV	henv	入力	環境ハンドルまたは SQL_NULL_HENV
HDBC	henv	入力	接続ハンドルまたは SQL_NULL_HDBC
HSTMT	hstmt	入力	ステートメント・ハンドルまたは SQL_NULL_HSTMT
UCHAR FAR *	szSqlState	出力	NULL 終了ストリングとしての SQLSTATE。
SDWORD FAR *	pfNativeError	出力	ネイティブ・エラー・コード (データ・ソースに固有)
UCHAR FAR *	szErrorMsg	出力	エラー・メッセージ・テキストのストレージを指すポインター
SWORD	cbErrorMsgMax	入力	szErrorMsg バッファーの最大長。この値は、SQL_MAX_MESSAGE_LENGTH - 1 以下の必要があります。
SWORD FAR *	pcbErrorMsg	出力	szErrorMsg で返すことのできるバイトの総数 (NULL 終了バイトを除く) を指すポインター。返すことのできるバイトの数が cbErrorMsgMax 以上の場合、szErrorMsg 内のエラー・メッセージ・テキストは、cbErrorMsgMax - 1 バイトに切り捨てられます。

戻り値

SQL_SUCCESS、 SQL_SUCCESS_WITH_INFO、 SQL_NO_DATA_FOUND、 SQL_ERROR、 または SQL_INVALID_HANDLE

SQLExecDirect (ODBC 1.0、コア)

ステートメント内にいずれかのパラメーターが存在する場合、SQLExecDirect は、そのパラメーター・マーカー変数の現行値を使用して準備可能ステートメントを実行します。SQLExecDirect は、一回限りの実行のために SQL ステートメントをサブミットする最も迅速な方法です。

構文

```
RETCODE SQLExecDirect(hstmt, szSqlStr, cbSqlStr)
```

SQLExecDirect 関数は、以下の引数を使用します。

表 24. SQLExecDirect の引数

型	引数	使用法	説明
HSTMT	hstmt	入力	ステートメント・ハンドル
UCHAR FAR *	szSqlStr	入力	実行される入力 SQL ステートメント
SDWORD	cbSqlStr	入力	szSqlStr の入力長

戻り値

SQL_SUCCESS、 SQL_SUCCESS_WITH_INFO、 SQL_NEED_DATA、
SQL_STILL_EXECUTING、 SQL_ERROR、 または SQL_INVALID_HANDLE

SQLExecute (ODBC 1.0、コア)

ステートメント内にいずれかのパラメーター・マーカーが存在する場合、
SQLExecute は、そのパラメーター・マーカー変数の現行値を使用して準備済みステートメントを実行します。

構文

```
RETCODE SQLExecute(hstmt)
```

SQLExecute ステートメントは、以下の引数を受け入れます。

表 25. SQLExecute の引数

型	引数	使用法	説明
HSTMT	hstmt	入力	ステートメント・ハンドル

戻り値

SQL_SUCCESS、 SQL_SUCCESS_WITH_INFO、 SQL_NEED_DATA、
SQL_STILL_EXECUTING、 SQL_ERROR、 または SQL_INVALID_HANDLE

SQLFetch (ODBC 1.0、コア)

SQLFetch は、結果セットからデータの行をフェッチします。ドライバーは、
SQLGetCol でストレージ・ロケーションにバインドされたすべての列に関するデータを返します。

構文

```
RETCODE SQLFetch(hstmt)
```

SQLFetch 関数は以下の引数を受け入れます。

表 26. SQLFetch の引数

型	引数	使用法	説明
HSTMT	hstmt	入力	ステートメント・ハンドル

.

戻り値

SQL_SUCCESS、 SQL_SUCCESS_WITH_INFO、 SQL_NO_DATA_FOUND、 SQL_STILL_EXECUTING、 SQL_ERROR、 または SQL_INVALID_HANDLE

SQLFreeConnect (ODBC 1.0、コア)

SQLFreeConnect は、接続ハンドルを解放し、そのハンドルに関連付けられたすべてのメモリーを解放します。

構文

```
RETCODE SQLFreeConnect(henv)
```

SQLFreeConnect 関数は以下の引数を受け入れます。

表 27. SQLFreeConnect の引数

型	引数	使用法	説明
HDBC	henv	入力	接続ハンドル

戻り値

SQL_SUCCESS、 SQL_SUCCESS_WITH_INFO、 SQL_ERROR、 または SQL_INVALID_HANDLE

SQLFreeEnv (ODBC 1.0、コア)

SQLFreeEnv は、環境ハンドルを解放し、その環境ハンドルに関連付けられたすべてのメモリーを解放します。

構文

```
RETCODE SQLFreeEnv(henv)
```

SQLFreeEnv 関数は以下の引数を受け入れます。

表 28. SQLFreeEnv の引数

型	引数	使用法	説明
HDBC	henv	入力	接続ハンドル

戻り値

SQL_SUCCESS、 SQL_SUCCESS_WITH_INFO、 SQL_ERROR、 または SQL_INVALID_HANDLE

SQLFreeStmt (ODBC 1.0、コア)

SQLFreeStmt は、特定の hstmt に関連付けられた処理を停止し、hstmt に関連付けられたオープン・カーソルをクローズし、保留中の結果を破棄し、またオプションでステートメント・ハンドルに関連付けられたすべてのリソースを解放します。

構文

```
RETCODE SQLFreeStmt(hstmt, fOption)
```

SQLFreeStmt 関数は以下の引数を受け入れます。

表 29. SQLFreeStmt の引数

型	引数	使用法	説明
HSTMT	hstmt	入力	ステートメント・ハンドル
UWORD	fOption	入力	以下のオプションのいずれかを使用できます。 SQL_CLOSE: hstmt に関連付けられたカーソル (定義されている場合) をクローズし、保留中の結果をすべて破棄します。アプリケーションは、同じかまたは別のパラメーター値を指定して SELECT ステートメントを再実行すれば、後でこのカーソルを再オープンすることができます。カーソルがオープンしていない場合には、このオプションはアプリケーションには効果がありません。 SQL_DROP: hstmt を解放し、それに関連付けられているすべてのリソースを解放し、カーソルをクローズし (オープンしている場合)、保留中のすべての行を破棄します。このオプションは、hstmt に対するすべてのアクセスを終了します。 hstmt は、再利用のためには再割り振りを行う必要があります。 SQL_UNBIND: 指定した hstmt に対して SQLGetCol によってバインドされたすべての列バッファーを解放します。 SQL_RESET_PARAMS: 指定した hstmt に対して SQLParamValue によって設定されたすべてのパラメーター・バッファーを解放します。

戻り値

SQL_SUCCESS、 SQL_SUCCESS_WITH_INFO、 SQL_ERROR、 または SQL_INVALID_HANDLE

SQLGetCursorName (ODBC 1.0、コア)

SQLGetCursorName は、指定した hstmt に関連付けられたカーソル名を返します。

構文

```
RETCODE SQLGetCursorName(hstmt, szCursor, cbCursorMax, pcbCursor)
```

SQLGetCursorName 関数は以下の引数を受け入れます。

表 30. SQLGetCursorName の引数

型	引数	使用法	説明
HSTMT	hstmt	入力	ステートメント・ハンドル
UCHAR FAR *	szCursor	出力	カーソル名のストレージを指すポインター
SWORD	cbCursorMax	入力	szCursor の長さ
SWORD FAR *	pcbCursor	出力	szCursor で返すことのできるバイトの総数 (NULL 終了バイトを除く)。返すことのできるバイトの数が cbCursorMax 以上の場合には、szCursor 内のカーソル名は、cbCursorMax - 1 バイトに切り捨てられます。

戻り値

SQL_SUCCESS、 SQL_SUCCESS_WITH_INFO、 SQL_ERROR、 または SQL_INVALID_HANDLE

SQLGetData (ODBC 1.0、レベル 1)

SQLGetData は、現在行内の単一のアンバインドされた列に関する結果データを返します。この関数は、文字、バイナリー、またはデータ・ソース固有のデータ型の列から文字データ値またはバイナリー・データ値 (例えば、SQL_LONGVARBINARY 列または SQL_LONGVARCHAR 列からのデータ) を部分単位でリトリーブします。

アプリケーションは、SQLGetData を呼び出す前に、SQLFetch を呼び出して、カーソルをデータの行に位置付ける必要があります。

構文

```
RETCODE SQLGetData(hstmt, icol, fCType, rgbValue, cbValueMax, pcbValue)
```

SQLGetData 関数は以下の引数を受け入れます。

表 31. SQLGetData の引数

型	引数	使用法	説明
HSTMT	hstmt	入力	ステートメント・ハンドル
UWORD	icol	入力	結果データの列番号。1 から順に左から右へ番号が付けられます。列番号 0 は、行のブックマークのリトリーブに使用されます。ブックマークは、ODBC 1.0 ドライバーおよび SQLFetch ではサポートされていません。

表 31. SQLGetData の引数 (続き)

型	引数	使用法	説明
SWORD	fCType	入力	結果データの C データ型。以下の値のいずれかを取る必要があります。 SQL_C_BINARY SQL_C_BIT SQL_C_BOOKMARK SQL_C_CHAR SQL_C_DATE SQL_C_DEFAULT SQL_C_DOUBLE SQL_C_FLOAT SQL_C_SLONG SQL_C_SSHORT SQL_C_STINYINT SQL_C_TIME SQL_C_TIMESTAMP SQL_C_ULONG SQL_C_USHORT SQL_C_UTINYINT SQL_C_DEFAULT は、データがデフォルトの C データ型に変換されることを指定します。 **注:** ドライバーは、ODBC 1.0 から fCType の以下の値もサポートする必要があります。アプリケーションは、 ODBC 1.0 ドライバーの呼び出しでは、ODBC 2.0 値ではなく、これらの値を使用する必要があります。 SQL_C_LONG SQL_C_SHORT SQL_C_TINYINT データの変換方法については、 297 ページの『SQL から C データ型へのデータ変換』を参照してください。

表 31. SQLGetData の引数 (続き)

型	引数	使用法	説明
PTR	rgbValue	出力	データのストレージを指すポインター
SDWORD	cbValueMax	入力	rgbValue バッファーの最大長。文字データについては、rgbValue は、NULL 終了バイト用のスペースも含む必要があります。 文字およびバイナリーの C データについては、cbValueMax によって、SQLGetData の 1 回の呼び出しで受信するデータの量が決まります。その他すべての型の C データについては、cbValueMax は無視されます。ドライバーは、rgbValue のサイズを、fCType で指定された C データ型のサイズと想定し、データ値の全体を返します。
SDWORD FAR *	pcbValue	出力	SQL_NULL_DATA、SQLGetData の現行呼び出しの前に rgbValue で返すことのできるバイトの総数 (文字データ用の NULL 終了バイトを除く)、または使用可能なバイト数を判断できない場合には SQL_NO_TOTAL 文字データについては、pcbValue が SQL_NO_TOTAL か、または cbValueMax 以上の場合、rgbValue 内のデータは、cbValueMax - 1 バイトに切り捨てられ、ドライバーによって NULL 終了されます。 バイナリー・データについては、pcbValue が SQL_NO_TOTAL か、または cbValueMax より大きい場合、rgbValue 内のデータは、cbValueMax バイトに切り捨てられます。 その他すべてのデータ型については、cbValueMax の値は無視され、ドライバーは、rgbValue のサイズを、fCType で指定された C データ型のサイズと想定します。

戻り値

SQL_SUCCESS、 SQL_SUCCESS_WITH_INFO、 SQL_NO_DATA_FOUND、 SQL_STILL_EXECUTING、 SQL_ERROR、 または SQL_INVALID_HANDLE

SQLNumResultCols (ODBC 1.0、コア)

SQLNumResultCols は、結果セットの列数を返します。

構文

```
RETCODE SQLNumResultCols(hstmt, pccol)
```

SQLNumResultCols 関数は以下の引数を受け入れます。

表 32. SQLNumResultCols の引数

型	引数	使用法	説明
HSTMT	hstmt	入力	ステートメント・ハンドル
SWORD FAR *	pccol	出力	結果セット内の列の数

戻り値

SQL_SUCCESS、 SQL_SUCCESS_WITH_INFO、 SQL_STILL_EXECUTING、 SQL_ERROR、 または SQL_INVALID_HANDLE

SQLPrepare (ODBC 1.0、コア)

SQLPrepare は、SQL ストリングの実行を準備します。

構文

```
RETCODE SQLPrepare(hstmt, szSqlStr, cbSqlStr)
```

SQLPrepare 関数は以下の引数を受け入れます。

表 33. SQLPrepare の引数

型	引数	使用法	説明
HSTMT	hstmt	入力	ステートメント・ハンドル
UCHAR FAR *	szSqlStr	入力	SQL テキスト・ストリング
SDWORD	cbSqlStr	入力	szSqlStr の長さ

戻り値

SQL_SUCCESS、 SQL_SUCCESS_WITH_INFO、 SQL_STILL_EXECUTING、 SQL_ERROR、 または SQL_INVALID_HANDLE

SQLRowCount (ODBC 1.0、コア)

SQLRowCount は、UPDATE、INSERT、または DELETE のステートメントの影響を受ける行の数、または SQLSetPos 内の SQL_UPDATE、SQL_ADD、または SQL_DELETE の操作によって影響を受ける行の数を返します。

目的

```
RETCODE SQLRowCount(hstmt, pcrow)
```

SQLRowCount 関数は以下の引数を受け入れます。

表 34. SQLRowCount の引数

型	引数	使用法	説明
HSTMT	hstmt	入力	ステートメント・ハンドル
SDWORD FAR *	pcrow	出力	UPDATE、INSERT、および DELETE のステートメントでは、pcrow は、要求の影響を受ける行の数になります。または、影響される行の数を入手できない場合には -1 になります。 その他のステートメントと関数では、ドライバーは、pcrow の値を定義できます。例えば、データ・ソースによっては、行をフェッチする前に、SELECT ステートメントまたはカタログ関数によって返された行の数を返すことができます。 **注:** 多くのデータ・ソースでは、結果セット内の行をフェッチする前にその行数を返すことはできません。相互運用性を最大にするために、複数のアプリケーションでこの動作に依存してはいけません。

戻り値

SQL_SUCCESS、 SQL_SUCCESS_WITH_INFO、 SQL_ERROR、 または SQL_INVALID_HANDLE

SQLSetCursorName (ODBC 1.0、コア)

SQLSetCursorName は、カーソル名をアクティブ hstmt に関連付けます。アプリケーションが SQLSetCursorName を呼び出さない場合、ドライバーは、SQL ステートメント処理の必要に応じてカーソル名を生成します。

構文

```
RETCODE SQLSetCursorName(hstmt, szCursor, cbCursor)
```

SQLSetCursorName 関数は以下の引数を受け入れます。

表 35. SQLSetCursorName の引数

型	引数	使用法	説明
HSTMT	hstmt	入力	ステートメント・ハンドル
UCHAR FAR *	szCursor	入力	カーソル名

表 35. SQLSetCursorName の引数 (続き)

型	引数	使用法	説明
SWORD	cbCursor	入力	szCursor の長さ

戻り値

SQL_SUCCESS、 SQL_SUCCESS_WITH_INFO、 SQL_ERROR、 または SQL_INVALID_HANDLE

SQLTransact (ODBC 1.0、コア)

SQLTransact は、接続に関連付けられたすべての hstmt に対するアクティブなすべての操作に関してコミット操作またはロールバック操作を要求します。SQLTransact は、henv に関連付けられたすべての接続に関して、コミット操作またはロールバック操作を実行するよう要求することもできます。

構文

```
RETCODE SQLTransact(henv, henv, fType)
```

SQLTransact 関数は以下の引数を受け入れます。

表 36. SQLTransact の引数

型	引数	使用法	説明
HENV	henv	入力	環境ハンドル
HDBC	henv	入力	接続ハンドル
UWORD	fType	入力	以下の 2 つの値のどちらかです。 SQL_COMMIT SQL_ROLLBACK

戻り値

SQL_SUCCESS、 SQL_SUCCESS_WITH_INFO、 SQL_ERROR、 または SQL_INVALID_HANDLE

非 ODBC solidDB Light Client 関数

このトピックでは、solidDB Light Client がサポートする 4 つの非 ODBC 関数について説明します。

1. SQLFetchPrev

   SQLFetchPrev は、結果セットからデータの直前の行をフェッチします。その機能は、対応する ODBC API の SQLFetch (直前のレコードに対して使用した場合) と同じです。詳しくは、 76 ページの『SQLFetch (ODBC 1.0、コア)』を参照してください。

2. SQLGetAnyData

   SQLGetAnyData は、現在行内の単一のアンバインドされた列に関する結果データを返します。SQLGetAnyData の機能は、対応する ODBC API の SQLGetData と同じです。この関数について詳しくは、 79 ページの『SQLGetData (ODBC 1.0、レベル 1)』を参照してください。

3. SQLGetCol

   SQLGetCol は、現在行内の単一列に関する結果データを取得します。アプリケーションは、この関数を使用して、一度に 1 列のデータをリトリーブできます。また、簡単に管理可能なブロック単位で大きなデータ値をリトリーブするのにも使用できます。SQLGetCol の機能は、対応する ODBC API の SQLGetData と同じです。詳しくは、 79 ページの『SQLGetData (ODBC 1.0、レベル 1)』を参照してください。

4. SQLSetParamValue

   SQLPrepare で指定された SQL ステートメント内のパラメーター・マーカーの値を設定します。パラメーター・マーカーは、1 から順に左から右へ番号が付けられており、任意の順序で設定できます。引数 rgbValue の値は、SQLExecute が呼び出されたときにパラメーター・マーカー用に使用されます。

構文

```
RETCODE SQLSetParamValue(hstmt, ipar, fCType, fSqlType,
cbColDef, ibScale, rgbValue, pcbValue)
```

SQLSetParamValue 関数は以下の引数を受け入れます。

表 37. SQLSetParamValue の引数

型	引数	使用法	説明
HSTMT	hstmt	入力	ステートメント・ハンドル
UWORD	ipar	入力	パラメーターの番号。1 から順に左から右へ番号が付けられます。
SWORD	fCType	入力	結果データの C データ型。このトピックの最後で、許可されるデータ型変換を確認してください。 以下の値のいずれかを取る必要があります。 • SQL_C_BINARY • SQL_C_CHAR • SQL_C_DOUBLE • SQL_C_FLOAT • SQL_C_LONG • SQL_C_SHORT

表 37. SQLSetParamValue の引数 (続き)

型	引数	使用法	説明
SDWORD	fSqlType	入力	パラメーターの SQL データ型。この表の後で、許可されるデータ型変換を確認してください。 以下の値のいずれかを取る必要があります。 • SQL_C_BINARY • SQL_C_CHAR • SQL_DATE • SQL_DECIMAL • SQL_C_DOUBLE • SQL_C_FLOAT • SQL_INTEGER • SQL_LONGVARBINARY • SQL_LONGVARCHAR • SQL_NUMERIC • SQL_REAL • SQL_SMALLINT • SQL_TIME • SQL_TIMESTAMP • SQL_TINYINT • SQL_VARBINARY • SQL_VARCHAR
UDWORD	cbColDef	入力	対応するパラメーター・マーカーの列または式の精度
SWORD	ibScale	入力	対応するパラメーター・マーカーの列または式の位取り
PTR	rgbValue	入力	出力データ
SDWORD *	pcbValue	入力	rgbValue 内のデータの長さ

fCType は、rgbValue の内容を記述します。fCType は、SQL_C_CHAR か、引数 fSqlType に相当する C データ型である必要があります。fCType が SQL_C_CHAR で fSqlType が数値型の場合、rgbValue は、文字ストリングから fSqlType で指定された型に変換されます。

fSqlType は、パラメーター・マーカーで参照される列または式のデータ型です。実行時に、rgbValue の値が読み取られ、fCType から fSqlType に変換された後、solidDB に送信されます。 rgbValue の値は変更されないことに注意してください。

cbColDef は、参照される列または式の列定義の長さまたは精度です。以下のように、cbColDef は、データのクラスによって異なります。

表 38. cbColDef の差異化

型	説明
SQL_CHAR SQL_VARCHAR	列の最大長
SQL_DECIMAL SQL_NUMERIC	最大 10 進数精度 (つまり、可能な総桁数)

ibScale は、参照される列の小数点以下の総桁数です。ibScale は、SQL_DECIMAL および SQL_NUMERIC のデータ型に対してのみ定義されます。rgbValue は、パラメーター・マーカーに対する実際のデータを含む必要のある文字ストリングです。データは、fCType 引数で指定される形式である必要があります。

pcbValue は、rgbValue 内のパラメーター・マーカー値の長さを示す整数です。これは、fCType が SQL_C_CHAR の場合、または NULL データベース値を指定する場合にのみ使用されます。パラメーター・マーカーに NULL 値を指定する場合には、この変数は SQL_NULL_DATA に設定する必要があります。この変数を SQL_NTS に設定すると、rgbValue は NULL 終了ストリングとして扱われます。

戻り値

SQL_SUCCESS、SQL_ERROR、または SQL_INVALID_HANDLE

診断

- fcType 引数で識別されるデータを、fSqlType 引数で識別されるデータ値に変換できない場合、以下の SQL_ERROR が返されます。

  ```
  07006 - Restricted data type attribute violation
  ```

- fcType 引数が無効な場合、以下の SQL_ERROR が返されます。

  ```
  S1003 - Program type out of range
  ```

- fSqlType 引数が無効な場合、以下の SQL_ERROR が返されます。

  ```
  S1004 - SQL data type out of range
  ```

- ipar 引数が 1 より小さい場合、以下の SQL_ERROR が返されます。

  ```
  S1009 - Invalid argument value
  ```

使用上の注意

この関数で設定されたすべてのパラメーターは、SQL_UNBIND_PARAMS オプションまたは SQL_DROP オプションを指定して SQLFreeStmt を呼び出すか、または同じパラメーター番号で SQLSetParamValue を再び呼び出すまで有効です。パラメーターを含む SQL ステートメントを実行すると、パラメーターの設定値は、solidDB に送信されます。

注: パラメーターの数は、準備されたステートメント内に存在するパラメーター・マーカーの数と正確に一致する必要があります。SQL ステートメント内に存在するパラメーター・マーカーよりも少ない数のパラメーター値が設定された場合、代わりに NULL 値が使用されます。

コード例

以下のコード例では、異なるパラメーター値を指定して、単純ステートメント INSERT INTO TESTTABLE (I,C) VALUES (?,?) を複数回実行する準備をします。

```
...
    char buf[255];
    SDWORD dwPar;
...
    rc = SQLPrepare(hstmt,(UCHAR*)"INSERT INTO TESTTABLE(I,C)
    VALUES (?,?)", SQL_NTS);
    if (SQL_SUCCESS != rc) {
         printf("Prepare failed. ¥n");
    }
    for (i=1;i<100;i++)
    {
            dwPar = i;
            sprintf(buf,"line%i",i);

        rc = SQLSetParamValue(
        hstmt,1,SQL_C_LONG,SQL_INTEGER,0,0,&dwPar,NULL );
        if (SQL_SUCCESS != rc) {
            printf("(SetParamValue 1 failed) ¥n");
                    return 0;
                }

        rc = SQLSetParamValue(
        hstmt,2,SQL_C_CHAR,SQL_CHAR,0,0,buf,NULL );
        if (SQL_SUCCESS != rc) {
            printf("(SetParamValue 1 failed) ¥n");
                    return 0;> >
            }
```

関連関数

表 39. 関連関数

関連情報	参照先
ステートメントの実行の準備	SQLPrepare
準備済み SQL ステートメントの実行	SQLExecute
SQL ステートメントの実行	SQLExecDirect

solidDB Light Client 型変換マトリックス

以下の表は、solidDB Light Client の SQLGetCol 関数と SQLSetParamValue 関数によって提供される型変換を示しています。

以下に、表内で C 変数データ型に使用される省略形を示します。

表 40. C 変数データ型の省略形

省略形	API パラメーター定義	C 変数データ型
Bin	SQL_C_BINARY	void*
Char	SQL_C_CHAR	char[ ]、char*
Long	SQL_C_LONG	long int (*)、32 ビット
Short	SQL_C_SHORT	short int (*)、16 ビット
Float	SQL_C_FLOAT	float (*)
Double	SQL_C_DOUBLE	double (*)

(*) これらのデータ型の変数を Light Client 関数呼び出しでパラメーターとして使用する場合、実際には代わりに変数を指すポインターを渡す必要があることに注意してください。

SQL データ型の説明については、17 ページの『データ型』を参照してください。

SQLGetCol 関数と SQLGetData 関数は、データベース列型と C 変数データ型の間で、以下のデータ型変換を実行します。

表 41. データベース列型と C 変数データ型間の変換

SQL データ型/ C 変数データ型	Bin	Char	Long	Short	Float	Double
TINYINT	*	*	*	*	*	*
LONG VARBINARY	*	*				
VARBINARY	*	*				
BINARY	*	*				
LONG VARCHAR	*	*				
CHAR	*	*				
NUMERIC		*	*	*	*	*
DECIMAL		*	*	*	*	*
INTEGER	*	*	*	*	*	*
SMALLINT	*	*	*	*	*	*
FLOAT	*	*	*	*	*	*

表 41. データベース列型と C 変数データ型間の変換 (続き)

SQL データ型/ C 変数データ型	Bin	Char	Long	Short	Float	Double
REAL	*	*	*	*	*	*
DOUBLE	*	*	*	*	*	*
DATE		*				
TIME		*				
TIMESTAMP		*				
VARCHAR	*	*				

SQLSetParamValue 関数は、C データ型とデータベース列型の間で以下の型変換を提供します。

表 42. データベース列型と C 変数データ型間の変換

SQL データ型/ C 変数データ型	Bin	Char	Long	Short	Float	Double
TINYINT		*	*	*		
LONG VARBINARY	*					
VARBINARY	*					
BINARY	*					
LONG VARCHAR		*				
CHAR		*				
NUMERIC		*	*	*	*	*
DECIMAL		*	*	*	*	*
INTEGER		*	*	*		
SMALLINT		*	*	*		
FLOAT		*	*	*	*	*
REAL		*	*	*	*	*
DOUBLE		*	*	*	*	*
DATE		*				

表 42. データベース列型と C 変数データ型間の変換 (続き)

SQL データ型/ C 変数データ型	Bin	Char	Long	Short	Float	Double
TIME		*				
TIMESTAMP		*				
VARCHAR		*				

4 solidDB JDBC ドライバーの使用

solidDB JDBC ドライバー 2.0 は、JDBC Type 4 ドライバーです。*Type 4* は、Java Database Connectivity (JDBC) 2.0 標準の 100% Pure Java 実装であることを意味しています。

JDBC API は、データベース接続、SQL ステートメント、結果セット、データベース・メタデータなどを表す Java クラスを定義します。これにより、Java プログラマーは SQL ステートメントを発行し、結果を処理できます。 JDBC は、Java でデータベースにアクセスするための基本 API です。JDBC テクノロジーについて詳しくは、JDBC テクノロジーのホーム・ページ (http://java.sun.com/products/jdbc/) を参照してください。

solidDB JDBC ドライバーは完全に Java で記述されており、TCP/IP ネットワーク・プロトコルを使用して、solidDB サーバーと直接通信します。 solidDB JDBC ドライバーには、追加のデータベース・アクセス・ライブラリーは必要ありません。このドライバーでは、Java ランタイム環境 (JRE) または Java 開発キット (JDK) が使用可能である必要があります。

solidDB JDBC ドライバーとは

このトピックでは、solidDB JDBC ドライバーについて説明します。

JDBC API は、データベース接続、SQL ステートメント、結果セット、データベース・メタデータなどを表す Java クラスを定義します。Java プログラマーは、これを使用して SQL ステートメントを発行し、結果を処理できます。JDBC は、Java でデータベースにアクセスするための基本 API です。JDBC テクノロジーについて詳しくは、JDBC テクノロジーのホーム・ページ (http://java.sun.com/products/jdbc/) を参照してください。

solidDB の JDBC ドライバーは完全に Java で記述されており、TCP/IP ネットワーク・プロトコルを使用して solidDB サーバーと直接通信します。solidDB ドライバーには、ODBC のような追加のデータベース・アクセス・ライブラリーは必要ありません。このドライバーでは、JRE (Java ランタイム環境) または JDK (Java Development Kit) が使用可能である必要があります。

solidDB JDBC ドライバーは、JDBC 2.0 標準の solidDB インプリメンテーションです。このドライバーは、JDK 1.4.2 以上をサポートするすべての Java 環境で使用可能です。

solidDB JDBC ドライバーの概要

solidDB JDBC ドライバー (`SolidDriver2.0.jar`) は、solidDB のインストール中にインストールされます。 solidDB パッケージで提供されるサンプルの Java プログラムを使用して、インストールを検証することができます。ご使用の環境によっては、solidDB JDBC ドライバーを使用する前に、さまざまな構成設定を設定する必要があります。

デフォルトのインストール・ディレクトリー

solidDB JDBC ドライバーは、solidDB のインストール中に、solidDB インストール・ディレクトリーの jdbc ディレクトリーにインストールされます。

jdbc ディレクトリーには、WebSphere® で使用するための、solidDB データ・ストア・ヘルパー・クラス (SolidDataStoreHelper.jar) も含まれています。

solidDB インストール・ディレクトリーの samples/jdbc ディレクトリーには、solidDB JDBC ドライバーを使用する Java コードのサンプルが含まれています。このサンプルの実行に関する説明は、同じディレクトリーにある readme.txt ファイルにあります。

Java 環境の要件

- JDBC API 仕様書リリース 2.0 をサポートし正常に機能する Java ランタイム環境または開発環境があることを確認してください。
- Java 環境の資料を調べて、圧縮バイトコードを使用できるかどうかを確認してください。 SolidDriver2.0.jar には、大部分の Java 仮想マシンで使用可能な圧縮バイトコード・フォーマットの solidDB JDBC ドライバー・クラスが含まれています。ただし、一部の環境 (Microsoft J++ など) では、圧縮解除されたバイトコードが必要です。ご使用の環境で、圧縮解除されたバイトコードが必要な場合、長いファイル名をサポートするツールを使用して、SolidDriver2.0.jar を unzip する必要があります。

CLASSPATH 環境変数の設定

ご使用の環境の CLASSPATH 環境変数に、solidDB JDBC ドライバーの .jar ファイル・インストール・パスを含める必要があります。

- **Windows**

  インストールを行うと、solidDB JDBC ドライバーのデフォルトのインストール・パスがシステム CLASSPATH 環境変数に自動的に追加されます。

  システム CLASSPATH 環境変数は、以下のようにして、「コントロール パネル」で確認および設定できます。

  「コントロール パネル」 → 「システム」 → 「詳細設定」 → 「環境変数」

- **Linux および UNIX**

  solidDB JDBC ドライバー (SolidDriver2.0.jar) のインストール・パスを含むように、CLASSPATH 環境変数を設定します。

  例えば Bourne シェルの場合、以下のコマンドを使用します。

  ```
  export CLASSPATH=<solidDB installation directory>/jdbc/SolidDriver2.0.jar:$CLASSPATH
  ```

  Bourne シェル以外のシェルを使用している場合、ご使用のシェルに合うように、このコマンドを変更してください。

sample1.java を使用したインストールの検証

sample1.java サンプル・アプリケーション (samples/jdbc ディレクトリーで入手可) を使用して、solidDB JDBC ドライバーのインストールを検証することができます。

sample1.java サンプル・アプリケーションは、以下の処置を実行します。

1. JDBC ドライバー・マネージャー・サービスを使用して、solidDB JDBC ドライバーを登録します。
2. 実行中の solidDB プロセス用に接続ストリングを要求します。
3. ドライバーを使用して、solidDB に接続します。
4. solidDB システム表の 1 つからデータをリトリーブするために、1 つの照会に対して以下のステートメントを作成します。

   ```
   SELECT TABLE_CATALOG, TABLE_SCHEMA, TABLE_NAME,
     TABLE_TYPE FROM TABLES
   ```

5. 照会を実行します。
6. 結果セットのすべての行をフェッチします。

   空の solidDB データベース・ディクショナリーには、およそ 86 行が含まれます。

サンプルの実行

サンプルを実行する前に、以下を確認してください。

- 使用する PATH 環境変数に、Java コンパイラーと JRE を保持するディレクトリーが含まれている。
- サンプル/jdbc ディレクトリーに、有効な solidDB (評価) ライセンスが含まれている。

1. solidDB プロセスをまだ実行していない場合は、プロセスをすぐに開始し、空のデータベースを作成します。
2. 作業ディレクトリーを /samples/jdbc に変更します。これは、サンプルの Java プログラムを含むディレクトリーです。
3. 以下のようにして、Java サンプル・プログラムをコンパイルします。

   ```
   javac sample1.java
   ```

4. 以下のコマンドを使用して、サンプル・アプリケーションを開始します。

   ```
   java sample1
   ```

5. アプリケーションによって、有効な接続ストリングの入力を求めるプロンプトが出されます。接続ストリングのフォーマットは、以下のとおりです。

   ```
   jdbc:solid://<hostname>:<port>/<username>/<password>
   ```

   例えば、以下のストリングは、ポート 2315 で TCP/IP プロトコルを listen するホスト「mymachine」で、solidDB サーバーへの接続を試みます。

   ```
   jdbc:solid://localhost:2315/dba/dba
   ```

   接続ストリングを入力したら、サンプル・アプリケーションは照会結果を出力します。

sample1.java サンプル・アプリケーションのトラブルシューティング

sample1.java サンプル・アプリケーションの実行中に生じる可能性のある問題、およびそれらの解決策を以下にリストします。

1. ドライバーを正常に登録できない。
   - Java 環境が java.sql クラスをサポートしていません。
   - SolidDriver2.0.jar が CLASSPATH 定義にありません。
2. solidDB プロセスに接続できない。
   - solidDB サーバーのバージョンは、6.5 以上である必要があります。

     古い solidDB バージョンは、現在のリリースで提供されるドライバーからの接続を拒否する場合があります。
   - 接続ストリングが間違っているか、solidDB が指定した TCP/IP を listen していない可能性があります。

     solidDB が実行していることを確認し、listen 情報を検証してください。例えば、solidDB SQL エディター (solsql) を使用して、ネットワークを介して接続が確立できていることを確認することができます。

solidDB JDBC ドライバーの登録

JDBC ドライバー・マネージャーは、ドライバーのロードとアンロード、および接続要求の適切なドライバーとのインターフェースを処理します。

ドライバーは、以下に示すように登録できます。このコードを実行すると、ドライバーは DriverManager に自身を登録します。

```
/ Class.forName サービスを使用した登録
Class.forName("solid.jdbc.SolidDriver");
```

データベースへの接続

ドライバー・マネージャーにドライバーを正しく登録すると、java.sql.Connection のインスタンスを作成することにより、接続が確立されます。

以下のコード例によって、java.sql.Connection のインスタンスが作成されます。

```
Connection conn = null;
// sCon は、JDBC 接続ストリングです。
(jdbc:solid://hostname:port/login/password)
String sCon = "jdbc:solid://fb9:1314/dba/dba";
try {
    conn =  DriverManager.getConnection(sCon);
} catch (SQLException e) {
    System.out.println("Connect failed : " + e.getMessage());
}
```

DriverManager.getConnection 関数では、パラメーターとして JDBC 接続ストリングが必要です。JDBC 接続ストリングにより、データベース・サーバーが稼働しているコンピューターが識別されます。また、このストリングには、サーバーへの接続に必要なその他の情報も含まれます。

solidDB の JDBC URL (接続ストリング) の構文は、以下のとおりです。

```
jdbc:solid://<hostname>:<port>/<username>/<password>[?<property-name>=<value>]...
```

例えば、以下の接続ストリングは、マシン fb9 のポート 1314 で TCP/IP プロトコルを listen する solidDB サーバーへの接続を試みます。

```
"jdbc:solid://fb9:1314/dba/dba"
```

アプリケーションは、複数の Connection オブジェクトを使用することにより、データベースに対して複数の接続を確立できます。接続ライフ・サイクルは、非常に正確な方法で管理する必要があります。そうしなければ、データベースへのアクセスを試行する同時ユーザーとアプリケーションの間で、競合が発生する可能性があります。詳しくは、121 ページの『コード例』を参照してください。

注: solidDB JDBC ドライバーは、照会の許可されない管理オプションに関する接続のみをサポートします。このタイプの接続では、java.util.Properties 名 ADMIN_USER を TRUE に設定してください。TRUE に設定して接続を確立した後は、ADMIN コマンドのみが許可されます。

トランザクションと自動コミット・モード

JDBC 仕様に定義されているように、solidDB データベースには、自動コミット・モードまたは非自動コミット・モードのいずれかで接続できます。

- 自動コミット・モード以外の場合、各トランザクションによる変更を他のデータベース接続で認識できるようにするには、そのトランザクションを明示的にコミットする必要があります。
- 自動コミット状態は、Connection.getAutoCommit() メソッドでモニターできます。
- この状態は、Connection.setAutoCommit() で設定できます。自動コミット状態に対する solidDB サーバーのデフォルト設定は true です。
- 自動コミット・モードがオフの場合、トランザクションは以下の 2 つの方法でコミットできます。
  - Connection.commit() メソッドの呼び出し、または
  - SQL 'COMMIT WORK' に関するステートメントの実行

データベース・エラーの処理

JDBC のデータベース・エラーは、例外メカニズムで処理および管理されます。JDBC インターフェースで指定されるメソッドの大部分は、SQLException のインスタンスをスローできます。これらのエラーは、通常のアプリケーション・ワークフローに出現する可能性があるため (例えば、並行性競合を表します)、このようなエラーに対する耐性をコードに持たせる必要があります。基本的には、コードの実行結果に関係なく、接続を「閉じている」以外の状態のままにしておくことは許されません。このアプローチでは、未処理の例外のために、使用可能なすべての接続が開いたままになるのを防止できます。

e.getErrorCode() を呼び出すことにより、例外のエラー・コードを取得できます。solidDB エラー・コードのリストについては、「*IBM solidDB 管理者ガイド*」の付録『エラー・コード』を参照してください。

以下のコード例は、データベースから返されたエラーを正しく処理する方法を示しています。

```
Public void listTablesExample() {
    try {
        Class.forName("solid.jdbc.SolidDriver");
    } catch (ClassNotFoundException e) {
        System.err.println("Solid JDBC driver is not registered
    in the classpath");
        return; //メソッドを終了します。
    }
    Connection conn = null;
    Statement stmt = null;
    ResultSet rs = null;
    try {
        conn = DriverManager.getConnection("jdbc:solid://
        localhost:1313", "dba", "dba");
        stmt = conn.createStatement();
        rs = stmt.executeQuery("SELECT * FROM tables");
        while (rs.next()) {
            System.out.println(rs.getObject(0));//結果を出力します。
}
    } catch (SQLException e) {
        e.printStackTrace();
    } finally {
        /* リソースが必要なくなった場合、
    finally{} ブロックで、
    作成したのと逆の順番で
    解放するのが良い考え方です。
    */
if (rs != null) {
            try {
                rs.close();
            } catch (SQLException sqlEx) { // 無視します。
                rs = null;
            }
        }
        if (stmt != null) {
            try {
                stmt.close();
            } catch (SQLException sqlEx) { // 無視します。
                stmt = null;
            }
        }
    }
    if (conn != null)
        try {
            conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            conn = null;
        }
}
```

solidDB および JDBC に関する特記事項

JDBC は、使用できる SQL ダイアレクトを指定しません。単に SQL をドライバーに渡すだけであり、ドライバーは、その SQL を直接データベースに渡すことも、SQL 自体を構文解析することもできます。このため、solidDB JDBC ドライバーの動作は、solidDB に固有のものになります。

関数によっては、JDBC 仕様のいくつかの詳細項目が未確定のままになっています。solidDB のメソッドのインプリメンテーションに固有の詳細については、99 ページの『JDBC ドライバーのインターフェースおよびメソッド』を参照してください。

solidDB JDBC ドライバーは、solidDB のカタログとスキーマをサポートしています。

ストアード・プロシージャーの実行

solidDB データベースでは、他の SQL ステートメントと同様に、`CALL proc_name [( parameter ...)]'` ステートメントを実行することにより、ストアード・プロシージャーを呼び出すことができます。また、プロシージャーは、標準 CallableStatement インターフェースを通して、JDBC 内で同様の方法で使用できます。

注: solidDB ストアード・プロシージャーは、結果セットを返すことができます。JDBC CallableStatement インターフェースを通してプロシージャーを呼び出す必要はありません。JDBC を使用して solidDB プロシージャーを呼び出す例については、121 ページの『コード例』で sample3 アプリケーションのソース・コードを参照してください。

JDBC ドライバーのインターフェースおよびメソッド

solidDB JDBC ドライバー 2.0 は、JDBC 2.0 標準と互換性があり、JDBC 2.0 オプション・パッケージ (以前はスタンダード・エクステンションと呼ばれていました) 内の一部の機能をサポートしています。

このトピックでは、標準 API との solidDB に固有の相異点について説明します。`java.sql` パッケージおよび `javax.sql` パッケージで、標準パッケージおよびインターフェースを参照できます。また、『既知の実装クラスの一覧 (All Known Implementing Classes)』のリストを確認することにより、特定のインプリメンテーション (実装) の詳細を参照できます。

solidDB JDBC ドライバーが各種データ型をサポートする方法については、133 ページの『solidDB JDBC ドライバー型変換マトリックス』を参照してください。

Array

java.sql.Array インターフェースは、サポートされません。このインターフェースは、Java プログラミング言語で、SQL の配列型のマップに使用されます。これは solidDB では現在使用できない SQL-99 規格を反映しています。

Blob

java.sql.Blob インターフェースはサポートされません。このインターフェースは、Java プログラミング言語で、SQL の Blob 型のマップに使用されます。これは solidDB では現在使用できない SQL-99 規格を反映しています。

CallableStatement

java.sql.CallableStatement インターフェースは、データベース・ストアード・プロシージャーの呼び出しをサポートするためのものです。したがって、solidDB ストアード・プロシージャーは、JDBC で他のステートメントと同じ方法で使用されます。solidDB サーバー上でのみアプリケーションを作成する場合には、

CallableStatement クラスを使用する必要はありません。しかし、移植性を考えると、CallableStatement を使用するのが賢明な選択です。

注: JDBC ドライバーでは、指定した型と並行性で ResultSet オブジェクトを生成する Statement オブジェクトを作成できます。デフォルトの結果セット型と結果セット並行性タイプをオーバーライドできることから、これは JDBC 1.0 の createStatement メソッドとは異なります。

標準 API との相異点

以下に、JDBC API で定義された標準 CallableStatement インターフェースとの相異点を示します。

表 43. 標準 CallableStatement インターフェースとの相異点

メソッド名	注
getArray(int i)	solidDB ではサポートされません。
getBlob(int i)	solidDB ではサポートされません。
getClob(int i)	solidDB ではサポートされません。
getDate(int parameterIndex, Calendar cal)	Java API の仕様どおりに機能します。 **注:** 指定された Calendar オブジェクトを使用して、デフォルトとは異なる時間帯とロケールを指定します。同じルールが、Calendar インスタンスで動作する他の類似したメソッドに対応します。
getObject (int i, Map map)	solidDB ではサポートされません。
getRef(int i)	solidDB ではサポートされません。
registerOutParameter(int parameterIndex, int sqlType, String typeName)	solidDB ではサポートされません。このメソッドは、「This method is not supported」というメッセージ付きで例外をスローします。

Clob

java.sql.Clob インターフェースはサポートされません。このインターフェースは、Java プログラミング言語で、SQL の Clob 型のマップに使用されます。これは solidDB では現在使用できない SQL-99 規格を反映しています。

Connection

java.sql.Connection インターフェースはパブリック・インターフェースです。指定したデータベースとの接続 (セッション) の確立に使用します。SQL ステートメントが実行され、接続のコンテキスト内で結果が返されます。

標準 API との相異点

以下に、JDBC API で定義された標準 Connection インターフェースとの相異点を示します。

表 44. 標準 Connection インターフェースとの相異点

メソッド名	注
getTypeMap()	solidDB はこのメソッドを提供していますが、このメソッドは常に NULL を返します。
isReadOnly()	データベースが読み取り専用として宣言されている場合、solidDB は、読み取り専用接続および読み取り専用トランザクションのみをサポートします。このメソッドは、常に false を返します。
nativeSQL(String sql)	Java API の仕様どおりに機能します。 solidDB JDBC ドライバーは、solidDB サーバーに渡す SQL を変更しません。ユーザーが渡す SQL 照会が返されます。
prepareCall(String sql)	Java API の仕様どおりに機能します。 **注:** エスケープ呼び出し構文はサポートされないことに注意してください。
setReadOnly(boolean readOnly)	データベースが読み取り専用として宣言されている場合、solidDB は、読み取り専用データベースおよび読み取り専用トランザクションのみをサポートします。このメソッドは存在しますが、接続の動作に影響を与えません。
setTransactionIsolation(int level)	Java API の仕様どおりに機能します。
setTypeMap(Map map)	solidDB ではサポートされません。

DatabaseMetaData

java.sql.DatabaseMetaData インターフェースは、パブリック抽象インターフェースです。データベースに関する一般的、総合的な情報を提供します。

solidDB は、このインターフェースのすべてのメソッドをサポートしています。

solidDB JDBC ドライバーが各種データ型をサポートする方法については、133 ページの『solidDB JDBC ドライバー型変換マトリックス』を参照してください。

ドライバー

java.sql.Driver インターフェースは、パブリック抽象インターフェースです。このインターフェースはすべてのドライバー・クラスでインプリメントされており、solidDB は、このインターフェースのすべてのメソッドをサポートしています。

PreparedStatement

java.sql.PreparedStatement インターフェースは、パブリック抽象インターフェースです。このインターフェースは、Statement インターフェースを拡張します。プリコンパイルされた SQL ステートメントを表すオブジェクトを提供します。

注: JDBC ドライバーでは、指定した型と並行性で ResultSet オブジェクトを生成する PreparedStatement オブジェクトを作成できます。デフォルトの結果セット型と結果セット並行性タイプをオーバーライドできることから、これは JDBC 1.0 の prepareStatement メソッドとは異なります。

サブインターフェース: CallableStatement

標準 API との相異点

以下に、JDBC API で定義された標準 PreparedStatement インターフェースとの相異点を示します。

表 45. 標準 PreparedStatement インターフェースとの相異点

メソッド名	注
setArray(int i, Array x)	solidDB ではサポートされません。
setBlob(int I, Blob x)	solidDB ではサポートされません。
setClob(int I, Clob x)	solidDB ではサポートされません。
setObject(int parameterIndex, Object x)	Java API の仕様どおりに機能します。 **注:** 以下のオブジェクトは、solidDB ではサポートされません。 BLOB、CLOB、ARRAY、REF、および java.util.Map を使用するオブジェクト。
setObject(int parameterIndex, Object x, int targetSqlType))	solidDB ではサポートされません。このメソッドは、「This method is not supported」というメッセージ付きで例外をスローします。
setObject(int parameterIndex, Object x, int targetSQLType, int scale)	solidDB ではサポートされません。このメソッドは、「This method is not supported」というメッセージ付きで例外をスローします。
setRef(int I, Ref x)	solidDB ではサポートされません。

Ref

java.sql.Ref インターフェースは、パブリック抽象インターフェースです。

このインターフェースは、データベース内の SQL 構造化型の値への参照です。このインターフェースは solidDB ではサポートされません。

ResultSet

java.sql.ResultSet インターフェースは、照会ステートメントからのデータベース結果セットを表すデータの表です。このオブジェクトには、そのデータの現在行を指すカーソルが含まれます。カーソルの初期位置は、先頭行の前です。次のメソッドで、次の行に移動します。結果セット内に行がそれ以上残されていない場合、このメソッドは false を返します。これにより、WHILE ループを使用して、結果セット内で処理を繰り返すことができます。

標準 API との相異点

以下に、JDBC API で定義された標準 ResultSet インターフェースとの相異点を示します。

表 46. 標準 ResultSet インターフェースとの相異点

メソッド名	注
getArray(int i)	solidDB ではサポートされません。
getArray(String ColName)	solidDB ではサポートされません。
getBigDecimal(String columnName)	Java API の仕様どおりに機能します。
getCharacterStream(int columnIndex)	Java API の仕様どおりに機能します。 注: JDBC ドライバーは、指定されたパラメーターを、指定された Reader オブジェクトに、指定された文字長で設定します。LONG VARCHAR/LONG WVARCHR パラメーターに大きな UNICODE 値を入力する場合、便宜上、java.io.Reader を通してその値を送信できます。JDBC ドライバーは、ファイルの終わりに到達するまで、必要に応じてストリームからデータを読み取ります。ドライバーは、UNICODE からデータベース CHAR フォーマットに、すべての必要な変換を行います。
getCharacterStream(String columnName)	Java API の仕様どおりに機能します。上の注は、このメソッドにも適用されます。
getFetchSize()	solidDB ではサポートされません。
getObject(int columnIndex)	Java API の仕様どおりに機能します。 注: 以下のオブジェクトは、solidDB ではサポートされません。 BLOB、CLOB、ARRAY、REF、および java.util.Map オブジェクトを使用する。
getObject(int i, Map map)	solidDB ではサポートされません。

表 46. 標準 *ResultSet* インターフェースとの相異点 (続き)

メソッド名	注
getObject(String colName, Map map)	solidDB ではサポートされません。このメソッドは、「This method is not supported」というメッセージ付きで例外をスローします。
getRef(int i)	solidDB ではサポートされません。
getRef(String colName)	solidDB ではサポートされません。
refreshRow()	solidDB ではサポートされません。
setFetchSize(int rows)	solidDB では操作が実行されません。毎回、データベースからフェッチする行の数の値を設定します。このメソッドでユーザーが設定した値は無視されます。

ResultSetMetaData

java.sql.ResultSetMetaData インターフェースは、パブリック抽象インターフェースです。このインターフェースは、ResultSet 内の列の型とプロパティーの検索に使用されます。

SQLData

java.sql.SQLData インターフェースはサポートされません。このインターフェースは、SQL ユーザー定義型のカスタム・マップに使用されます。これは solidDB では現在使用できない SQL-99 規格を反映しています。

SQLInput

java.sql.SQLInput インターフェースはサポートされません。このインターフェースは、SQL 構造化型または特殊型のインスタンスを表す入力ストリームです。これは solidDB では現在使用できない SQL-99 規格を反映しています。

SQLOutput

java.sql.SQLOutput インターフェースはサポートされません。このインターフェースは、ユーザー定義型の属性をデータベースに書き戻すために使用する出力ストリームです。これは solidDB では現在使用できない SQL-99 規格を反映しています。

Statement

java.sql.Statement インターフェースは、パブリック抽象インターフェースです。静的 SQL ステートメントを実行し、実行結果を取得するために使用するオブジェクトです。

注: JDBC ドライバーでは、指定した型と並行性で ResultSet オブジェクトを生成する Statement オブジェクトを作成できます。デフォルトの結果セット型と結果セッ

ト並行性タイプをオーバーライドできることから、これは JDBC 1.0 の CreateStatement メソッドとは異なります。

サブインターフェース:

- CallableStatement
- PreparedStatement

標準 API との相異点

以下に、JDBC API で定義された標準 Statement インターフェースとの相異点を示します。

表 47. 標準 Statement インターフェースとの相異点

メソッド名	注
getFetchSize()	solidDB では操作が実行されません。
getMaxFieldSize()	MaxFieldSize は、solidDB サーバーの動作に影響を与えません。
getMoreResults()	solidDB は、複数の結果セットをサポートしません。
getResultSetType()	solidDB ではサポートされません。
setFetchSize(int rows)	solidDB では操作が実行されません。毎回、データベースからフェッチする行の数の値を設定します。このメソッドでユーザーが設定した値は無視されます。
setMaxFieldSize(int max)	MaxFieldSize は、solidDB サーバーの動作に影響を与えません。

Struct

java.sql.Struct インターフェースはサポートされません。このインターフェースは、SQL 構造化型に関する Java プログラミング言語での標準マッピングを表します。これは solidDB では現在使用できない SQL-99 規格を反映しています。

ResultSet (更新可能)

java.sql.Resultset インターフェースは、更新可能な ResultSet オブジェクトを生成するためのメソッドを含みます。結果セットは、その並行性タイプが CONCUR_UPDATABLE の場合に更新可能です。update xxx メソッド (xxx はデータ型を示します)、および updateRow メソッドと deleteRow メソッドを使用して、結果セット内で行を更新または削除したり、新しい行を挿入したりすることができます。

標準 API との相異点

以下に、JDBC API で定義された標準 ResultSet インターフェースとの相異点を示します。

表 48. 標準 ResultSet インターフェースとの相異点

メソッド名	注
getRef(int i)	このメソッドはサポートされません。
getRef(String colName)	このメソッドはサポートされません。
refreshRow()	このメソッドはサポートされません。
rowDeleted()	このメソッドはサポートされません。
rowInserted()	このメソッドはサポートされません。
setFetchSize(int rows)	このメソッドはサポートされません。

solidDB JDBC ドライバーの拡張機能

このセクションでは、solidDB JDBC ドライバーに含まれている非標準の拡張機能について説明します。

WebSphere の互換性

solidDB JDBC ドライバーには、WebSphere の互換性を向上させる機能があります。

Java トランザクション API (JTA) のサポート

solidDB サーバーは、Java トランザクション API (JTA) インターフェースを使用して、分散トランザクションに関与することができます。「*Java Transaction API Specification 1.1*」で説明されているように、以下のインターフェースがサポートされます。

- XAResource インターフェース (javax.transaction.xa.XAResource)
- Xid インターフェース (javax.transaction.xa.Xid)

solidDB Universal Cache で SQL パススルーとともに JTA を使用する場合は、読み取りステートメント (SELECT) のみがサポートされます。

WebSphere での solidDB データ・ストア・ヘルパー・クラス

WebSphere では、WebSphere 内で使用する JDBC データ・ソース用にアダプター・クラスが必要です。これらのアダプターの基本クラスは com.ibm.websphere.rsadapter.GenericDataStoreHelper クラスであり、solidDB は、com.ibm.websphere.rsadapter.SolidDataStoreHelper というクラス内にこのアダプターの独自のバージョンを実装します。

このクラスは、solidDB 製品内で SolidDataStoreHelper.jar と呼ばれる別個のアーカイブ・ファイルとして提供されます。このファイルは、solidDB インストール・ディレクトリーの jdbc ディレクトリーにあります。

WebSphere に、新しい solidDB データ・ソースを構成する場合、以下を行う必要があります。

- 構成のデータ・ストア・ヘルパー・フィールドに、クラス com.ibm.websphere.rsadapter.SolidDataStoreHelper を指定する。
- WebSphere のデータ・ソース構成に、SolidDataStoreHelper.jar ファイルへの絶対パスを指定する。

WebSphere に新しいデータ・ソースを定義する方法について詳しくは、WebSphere の資料を参照してください。

solidDB の WebSphere サンプル・アプリケーションを Websphere Studio Application Developer のワークスペースにインストールする方法については、solidDB インストール・ディレクトリーにある samples/websphere ディレクトリーを参照してください。

solidDB データ・ソース・プロパティーおよび WebSphere

WebSphere に新しいデータ・ソースを構成する場合には、以下のプロパティーを定義する必要があります。

URL

- 型: java.lang.String
- 値には、'jdbc:solid://<hostname>:<port>' のような構文を使用する必要があります。

user

- 型: java.lang.String
- 値は、正当なユーザー名である必要があります。

password

- 型: java.lang.String
- 値は、有効なパスワードである必要があります。

JDBC での接続タイムアウト

このトピックでは、接続タイムアウトの設定に solidDB で使用可能な機能について説明します。

接続タイムアウト は、接続ソケットを通してデータ伝送を実行する任意の JDBC 呼び出しの応答タイムアウトを意味しています。指定された時間内に応答メッセージを受信できない場合には、I/O 例外がスローされます。JDBC 標準 (2.0/3.0) は、接続タイムアウトの設定をサポートしていません。solidDB では、この設定のために 2 つの方法が導入されています。1 つは、非標準ドライバー・マネージャー拡張メソッドを使用することであり、もう 1 つはプロパティー・メカニズムを使用することです。どちらの場合にも、時間単位は 1 ミリ秒です。

ドライバー・マネージャー・メソッド get/setConnectionTimeout()

以下の例で、ソリューションを説明します。設定の効果はすぐに有効になります。強制的に切断したい場合には、タイムアウトにゼロを設定できます。

```
//Solid JDBC をインポートします。
import solid.jdbc.*;

//接続を定義します。
solid.jdbc.SolidConnection conn = null;

//Solid に固有のメソッドを使用するために、SolidConnection にキャストします。
conn = (SolidConnection)java.sql.DriverManager.getConnection(sCon);

//接続タイムアウトをミリ秒単位で設定します。
conn.setConnectionTimeout(3000);
```

非標準の接続プロパティー

以下の接続プロパティーを使用して、接続固有の動作を実現できます。

ステートメント・キャッシュ・プロパティー (StatementCache)

solidDB JDBC ドライバーは、接続のステートメント・キャッシュの値を設定するプロパティーを導入します。

プロパティーの名前は「StatementCache」であり、キャッシュのデフォルト・サイズは 8 です。有効な値の範囲は 1 以上 512 以下です。値がこの範囲を超えると、ドライバーは暗黙に 1 か 512 の値を強制します。

以下に、このプロパティーの使用例を示します。

```
// Solid JDBC ドライバー・インスタンスを作成します。
Class.forName("solid.jdbc.SolidDriver");

// 新しい Properties インスタンスを作成し、
// StatementCache プロパティーの値を挿入します。
java.util.Properties props = new java.util.Properties();
props.put("StatementCache","32");

// 使用する接続ストリングを定義します。
String sCon="jdbc:solid://localhost:1315/uname1/pwd1";

// ステートメント・キャッシュ 32 で Connection オブジェクトを取得します。
java.sql.Connection conn = java.sql.DriverManager.getConnection(sCon, props);
```

タイムアウト・プロパティー

以下のセクションにリストしたタイムアウトは、接続プロパティーとして設定できます。

接続タイムアウト・プロパティー (solid_connection_timeout_ms)

プロパティー「solid_connection_timeout_ms」を使用して、接続タイムアウト値を (ミリ秒単位で) 設定できます。このプロパティーは、新しい接続を取得する前に設定する必要があります。いったん接続オブジェクトを作成すると、プロパティー値を変更しても効果がありません。

ログイン・タイムアウト・プロパティー (solid_login_timeout_ms)

プロパティー「solid_login_timeout_ms」を使用して、接続を開く場合のタイムアウトを (ミリ秒単位で) 設定できます。

注: メソッド DriverManger.setLoginTimeout(seconds) を使用して、ログイン・タイムアウトを設定することもできます。これは、標準に準拠したメソッドです。

アイドル・タイムアウト・プロパティー (solid_idle_timeout_min)

プロパティー「solid_idle_timeout_min」を使用して、接続が指定した時間より長い間アイドル状態になったときに起動するタイムアウトを (分単位で) 設定できます。

このプロパティー値を設定しなかった場合は、サーバー・サイドの構成パラメーター **ConnectTimeOut** の設定が適用されます。ファクトリー値は 480 分です。0 は「無期限のタイムアウト」(有効期限がありません) を意味します。

例

以下の例は、「solid_connection_timeout_ms」プロパティーを使用して接続タイムアウトを設定する方法を示しています。

```
// 「solid_connection_timeout_ms」プロパティーで接続タイムアウトを設定します。 //
public class Test {

  public static void main( String args[] ){

    // プロパティー・オブジェクトを作成します。
    Properties props = new Properties();

    // プロパティーにユーザー名とパスワードを設定します。
    props.put("user", "MYUSERNAME");
    props.put("password", "MYPASSWORD");

    //
    // プロパティー・オブジェクトに接続タイムアウトを設定します。
    //
    props.put("solid_connection_timeout_ms", "10000");

    try {

      // ドライバーを作成します。
      Driver d = (Driver)(
          Class.forName("solid.jdbc.SolidDriver").newInstance());

      // URL とプロパティー情報で接続を取得します。
      Connection c = DriverManager.getConnection(
          "jdbc:solid://localhost:1313", props );

      // 接続を閉じます。
      c.close();

    } catch ( Exception e ) {
      ; // 無事に終了することができました。
    }
  }
}
```

Appinfo プロパティー (solid_appinfo)

Appinfo という接続属性を使用すると、同じユーザー名の下で同じコンピューターで実行されているアプリケーションを、トレースと管理の目的でユニークに識別できます。Appinfo は、サーバー・サイドで、コマンド ADMIN COMMAND 'userlist'

によってリトリーブできます。デフォルトでは、値 (ストリング) は設定されません。この値は、接続プロパティー「solid_appinfo」で設定できます。

注: ODBC アプリケーションでは、Appinfo の値は環境変数 SOLAPPINFO を介して受け渡されます。

透過接続 (TC) プロパティー

透過接続 (TC) は、solidDB HA ソリューション (HotStandby) で使用できる接続モードの 1 つです。 JDBC では、TC モードは以下の接続プロパティーによって設定されます。

- 「solid_tf_level」

  透過的フェイルオーバー・レベルを「CONNECTION」、「SESSION」、または「NONE」に設定します。デフォルトは「NONE」です。

- 「solid_preferred_access」

  優先されるアクセス・モードを「WRITE_MOSTLY」または「READ_MOSTLY」に設定します。値「READ_MOSTLY」は、1 次サーバーと 2 次サーバーの間で、読み取り専用トランザクションの自動ロード・バランシングを設定します。デフォルトは「WRITE_MOSTLY」で、これは通常の HotStandby 操作に対応し、すべての負荷が 1 次サーバーへ転送されます。

- 「solid_tf1_reconnect_timeout」

  失敗した場合の再接続再試行のタイムアウトを設定します。単位はミリ秒です。デフォルトは 10000 (10 秒) です。

TC 接続プロパティーについて詳しくは、「*IBM solidDB 高可用性ユーザー・ガイド*」のセクション『*透過接続の使用*』を参照してください。

SQL パススルー・プロパティー

SQL パススルー・モードは、以下の接続プロパティーを使用して設定できます。

- ″solid_passthrough_read″

  読み取り型ステートメントの SQL パススルー・モードを「NONE」、「CONDITIONAL」または「FORCE」に設定します。

- ″solid_passthrough_write″

  書き込み型ステートメントの SQL パススルー・モードを「NONE」、「CONDITIONAL」または「FORCE」に設定します。

SQL パススルーおよびパススルー・モードについて詳しくは、「*IBM solidDB Universal Cache ユーザー・ガイド*」のセクション『*SQL パススルー・モードの設定*』を参照してください。

URL ストリングでの接続プロパティーの設定

すべての接続プロパティーは、接続時に、JDBC メソッド DriverManager.getConnection() に渡す JDBC URL の中でも設定できます。この場合、solidDB JDBC URL の構文は以下のとおりです。

```
"jdbc:solid://<hostname>:<port>/>username>/<password>[?<property-name>=<value>]..."
```

例

```
"jdbc:solid://locahost:1964/dba/dba"
"jdbc:solid://server1.acme.com:1964/dba/dba?solid_login_timeout_ms=100"
"jdbc:solid://server1.acme.com:1964/dba/dba?solid_login_timeout_ms=100?solid_idle_timeout_min=5"
```

JDBC 2.0 オプション・パッケージ API のサポート

solidDB JDBC ドライバー 2.0 は、JDBC 2.0 仕様のオプション・パッケージ (以前はスタンダード・エクステンションと呼ばれていました) 内の一部の機能をサポートしています。

JDBC 接続のプーリング

JDBC 2.0 スタンダード・エクステンション API では、ユーザーがニーズに最も適した特定のキャッシングまたはプーリングのアルゴリズムを使用して、プーリング技法をインプリメントできるように指定しています。solidDB は、標準の ConnectionPoolDataSource および PooledConnection のインターフェースを実装するクラスを提供します。

solidDB はこれらのクラスを以下のようにインプリメントします。

- ConnectionPoolDataSource

  javax.sql.ConnectionPoolDataSource インターフェースは、プールされた java.sql.Connection オブジェクトのリソース・マネージャー接続ファクトリーとして機能します。solidDB は、このインターフェースのインプリメンテーションを SolidConnectionPoolDataSource クラスで提供します。 API 関数については、『ConnectionPoolDataSource API 関数』を参照してください。

- PooledConnection

  javax.sql.PooledConnection インターフェースは、物理接続をデータベースにカプセル化します。solidDB はこのインターフェースのインプリメンテーションを、SolidPooledConnection クラスで提供します。 API 関数については、 118 ページの『PooledConnection API 関数 』を参照してください。

注: solidDB は、実際の接続プールの実装は提供しません。つまり、PooledConnection インスタンスを実際にプールするデータ構造とロジックは使用できません。ユーザーは、独自の接続プーリング・ロジック、つまり実際に接続をプールするクラスを実装する必要があります。

ConnectionPoolDataSource API 関数

パブリック・クラス SolidConnectionPoolDataSource は、javax.sql.ConnectionPoolDataSource をインプリメントします。javax.sql.ConnectionPoolDataSource インターフェースの API 関数は、以下のように記述されます。

表 49. コンストラクター

説明タイプ	**説明**
関数名	コンストラクター
関数型	solidDB プロプラエタリー API
説明	クラス変数を初期化します。
パラメーター	なし
戻り値	なし
構文および例外	public SolidConnectionPoolDataSource()

表 50. コンストラクター

説明タイプ	**説明**
関数名	コンストラクター
関数型	solidDB プロプラエタリー API
説明	クラス変数を初期化します。
パラメーター	DB サーバーを識別するストリングとしての URL
戻り値	なし
構文および例外	public SolidConnectionPoolDataSource(String urlStr)

表 51. setDescription

説明タイプ	**説明**
関数名	setDescription
関数型	solidDB プロプラエタリー API
説明	この関数は、説明ストリングを設定します。
パラメーター	説明ストリング (descString)
戻り値	なし
構文および例外	public void setDescription(String descString)

表 52. getDescription

説明タイプ	**説明**
関数名	getDescription
関数型	solidDB プロプラエタリー API
説明	この関数は、説明ストリングを返します。
パラメーター	なし
戻り値	ストリング (説明) を返します。
構文および例外	public String getDescription()

表 53. setURL

説明タイプ	**説明**
関数名	setURL
関数型	solidDB プロプラエタリー API
説明	この関数は、solidDB サーバーを指す URL ストリングを設定します。
パラメーター	URL ストリング (urlStr)
戻り値	なし
構文および例外	public void setURL(String urlStr)

表 54. getURL

説明タイプ	**説明**
関数名	getURL
関数型	solidDB プロプラエタリー API
説明	この関数は、DB URL ストリングを返します。
パラメーター	なし
戻り値	ストリング (URL) を返します。
構文および例外	public String getURL()

表 55. setUser

説明タイプ	説明
関数名	setUser
関数型	solidDB プロプラエタリー API
説明	この関数は、ユーザー名ストリングを設定します。 (WebSphere の互換性)
パラメーター	ユーザー名ストリング
戻り値	なし
構文および例外	public void setUser(String newUser)

表 56. getUser

説明タイプ	説明
関数名	getUser
関数型	solidDB プロプラエタリー API
説明	この関数は、ユーザー名ストリングを返します。 (WebSphere の互換性)
パラメーター	なし
戻り値	ストリング (ユーザー名) を返します。
構文および例外	public String getUser()

表 57. setPassword

説明タイプ	説明
関数名	setPassword
関数型	solidDB プロプラエタリー API
説明	この関数は、パスワード・ストリングを設定します。 (WebSphere の互換性)
パラメーター	パスワード・ストリング
戻り値	なし
構文および例外	public void setPassword(String newPassword)

表 58. *getPassword*

説明タイプ	**説明**
関数名	getPassword
関数型	solidDB プロプラエタリー API
説明	この関数は、パスワード・ストリングを返します。 (WebSphere の互換性)
パラメーター	なし
戻り値	ストリング (パスワード) を返します。
構文および例外	public String getPassword()

表 59. *setConnectionURL*

説明タイプ	**説明**
関数名	setConnectionURL
関数型	solidDB プロプラエタリー API
説明	この関数は、solidDB サーバーを指す URL ストリングを設定します。
パラメーター	URL ストリング
戻り値	なし
構文および例外	public void setConnectionURL(String newUrl)

表 60. *getConnectionURL*

説明タイプ	**説明**
関数名	getConnectionURL
関数型	solidDB プロプラエタリー API
説明	この関数は、URL ストリングを返します。
パラメーター	なし
戻り値	ストリング (URL) を返します。
構文および例外	public String getConnectionURL()

表 61. *getLoginTimeout*

説明タイプ	**説明**
関数名	getLoginTimeout
関数型	javax.sql.ConnectionPoolDataSource API
説明	この関数は、ログイン・タイムアウト値を返します。
パラメーター	なし
戻り値	タイムアウト値を整数 (秒数) として返します。
構文および例外	public int getLoginTimeout() は、java.sql.SQLException をスローします。

表 62. *getLogWriter*

説明タイプ	**説明**
関数名	getLogWriter
関数型	javax.sql.ConnectionPoolDataSource API
説明	この関数は、デバッグ・メッセージの出力に使用するライターへのハンドルを返します。
パラメーター	なし
戻り値	java.io.PrintWriter へのハンドルを返します。
構文および例外	public java.io.PrintWriter getLogWriter() は、java.sql.SQLException をスローします。

表 63. *getPooledConnection*

説明タイプ	**説明**
関数名	getPooledConnection
関数型	javax.sql.ConnectionPoolDataSource API
説明	この関数は、接続プールから PooledConnection オブジェクトを返します。このオブジェクトはデータベース・サーバーへの有効な接続を確立しています。
パラメーター	なし
戻り値	PooledConnection オブジェクトを返します。

表 63. *getPooledConnection (続き)*

説明タイプ	説明
構文および例外	public javax.sql.PooledConnection getPooledConnection() この関数は、java.sql.SQLException をスローします。

表 64. *getPooledConnection*

説明タイプ	説明
関数名	getPooledConnection
関数型	javax.sql.ConnectionPoolDataSource API
説明	この関数は、接続プールから PooledConnection オブジェクトを返します。このオブジェクトはデータベース・サーバーへの有効な接続を確立しています。
パラメーター	ユーザー (ストリングとしてのユーザー名)、パスワード (ストリングとしてのパスワード)
戻り値	PooledConnection オブジェクトを返します。
構文および例外	public javax.sql.PooledConnection getPooledConnection(String user, String password) この関数は、java.sql.SQLException をスローします。

表 65. *setLoginTimeout*

説明タイプ	説明
関数名	setLoginTimeout
関数型	javax.sql.ConnectionPoolDataSource API
説明	この関数は、ログイン・タイムアウト値を秒数で設定します。
パラメーター	秒数 (整数として)
戻り値	なし
構文および例外	public void setLoginTimeout(int seconds)

表 66. *setLogWriter*

説明タイプ	説明
関数名	setLogWriter

表 66. *setLogWriter (続き)*

説明タイプ	説明
関数型	javax.sql.ConnectionPoolDataSource API
説明	この関数は、デバッグ・メッセージの出力/ログに使用するライター・オブジェクトへのハンドルを設定します。
パラメーター	java.io.PrintWriter へのハンドル
戻り値	なし
構文および例外	public void setLogWriter(java.io.PrintWriter out) この関数は、java.sql.SQLException をスローします。

PooledConnection API 関数

パブリック・クラス SolidPooledConnection は、javax.sql.PooledConnection をインプリメントします。javax.sql.PooledConnection インターフェースの API 関数は以下のとおりです。

表 67. *addConnectionEventListener*

説明タイプ	説明
関数名	addConnectionEventListener
関数型	javax.sql.PooledConnection API
説明	このオブジェクトが接続を解放する際に通知するイベント・リスナーを追加します。このリスナーは通常、接続プーリング・モジュールです。
パラメーター	リスナー (javax.sql.ConnectionEventListener へのハンドル)
戻り値	なし
構文および例外	public void addConnectionEventListener(javax.sql.ConnectionEventListener listener)

表 68. *close*

説明タイプ	説明
関数名	close
関数型	javax.sql.PooledConnection API
説明	この関数は、物理接続を閉じます。

表 68. close (続き)

説明タイプ	説明
パラメーター	なし
戻り値	なし
構文および例外	public void close() この関数は、java.sql.SQLException をスローします。

表 69. getConnection

説明タイプ	説明
関数名	getConnection
関数型	javax.sql.PooledConnection API
説明	java.sql.Connection へのハンドルを返します。
パラメーター	なし
戻り値	java.sql.Connection
構文および例外	public java.sql.Connection getConnection() この関数は、java.sql.SQLException をスローします。

表 70. removeConnectionEventListener

説明タイプ	説明
関数名	removeConnectionEventListener
関数型	javax.sql.PooledConnection API
説明	この関数は、リスナーへの参照を解除します。
パラメーター	リスナー
戻り値	なし
構文および例外	public void removeConnectionEventListener(javax.sql.ConnectionEventListener listener)

solidDB の接続済み RowSet クラス: SolidJDBCRowSet

このトピックで説明されている RowSet は、solid.jdbc.SolidBaseRowSet (javax.sql.RowSet をインプリメント) コンストラクターを拡張します。

```
/**
 * 既存の接続ハンドルを使用して SolidJDBCRowSet を作成します。*/
public SolidJDBCRowSet(java.sql.Connection conn)

/**
 * 既存の ResultSet ハンドルを使用して SolidJDBCRowSet を作成します。*/
public SolidJDBCRowSet(java.sql.ResultSet rset)

/**
 * 特定の URL、ユーザー名、およびパスワードを使用して、新しい
 * SolidJDBCRowSet を作成します。
*/
public SolidJDBCRowSet(String url, String uname, String pwd)

/**
 * 特定の URL、ユーザー名、パスワード、および JNDI ネーミング・コンテキストを
 * 使用して、新しい SolidJDBCRowSet を作成します。
*/
public SolidJDBCRowSet(String dsname,
                       String username,
                       String password,
                       Context namingcontext)
```

例えば、「*Java 2 Platform, Standard Edition, v 1.4.2 API Specification*: http://java.sun.com/j2se/1.4.2/docs/api/javax/sql/RowSet.html」のメソッド・インターフェースの説明を参照してください。

SolidJDBCRowSet の使用に関する考慮事項

データベースへの接続を確立する前に呼び出すことができるメソッドがいくつかあります (通常は、実行するコマンドのパラメーターを設定するため、または RowSet インスタンスのプロパティーを設定するために呼び出します)。ただし、RowSet インターフェース・メソッドの大部分は、データベースへの接続を確立した後にのみ呼び出すことができます。これは、コマンドがメソッド setCommand(String) で設定され、メソッド execute() が呼び出されたことを意味しています。SolidJDBCRowSet インスタンスに以前の java.sql.Connection ハンドルがない場合は、execute() の呼び出し中に接続が確立されます。execute() の呼び出し後、行セット・インスタンスには java.sql.Connection オブジェクトおよび java.sql.PreparedStatement オブジェクトが含まれ、コマンドを実行したのが照会ステートメントであった場合は、java.sql.ResultSet ハンドルも含まれます。また、setString、setObject など、すべてのパラメーター設定メソッドも含まれます。

以下の例では、SolidJDBCRowSet クラスの正しい使用法を説明します。

```
/**
 * SolidJDBCRowSet の使用方法を示す簡単な例です。
 * 最初に、接続済み RowSet クラスのインスタンスを作成します。
 * URL、ユーザー名、パスワードを以下のコンストラクターにすぐに
 * 指定できますが、この例では対応する値にヌル・パラメーターを指定して
 * setUrl、setUsername などの RowSet クラスのメソッドの使用法を示します。
 */
SolidJDBCRowSet rs = new SolidJDBCRowSet(null, null, null);

// 接続の URL を設定します。
rs.setUrl("jdbc:solid://localhost:1313");

// ユーザー名を設定します。
rs.setUsername("user1");

// パスワードを設定します。
rs.setPassword("pwd1");
```

```
/**
 * 注意! コマンド・パラメーターおよびその他のプロパティーは
 * 任意の順序で設定できます。例えば、実行するコマンドを
 * 定義する前にパラメーターを設定できます。また、コマンド・
 * パラメーターを任意の順序で定義できます。コマンド・
 * ステートメントおよび指定したパラメーターは、execute() メソッド
 * 呼び出しでデータベースへの接続が完了するまで、
 * 構文解析されないからです。
 */

// コマンドにパラメーター #2 を設定します。
rs.setString(2,"'SYS_SYNC%'");

// コマンド・ストリングを設定します。
rs.setCommand("select table_name from tables where table_name like ?
and table_name not like ?;");

// パラメーター #1 を設定します。
rs.setString(1,"'SYS_%'");

// コマンドを実行します。データベースへの接続は、この呼び出しの前
// には確立されません。
rs.execute();

// これで、ResultSet をブラウズできます。
while( rowset.next() ){
// いろいろな操作をしてみます。
}

// 結果セットを閉じます。このメソッド呼び出しによって、データベースへの接続も
// 閉じられます。
rs.close()
```

Java Naming and Directory Interface (JNDI)

solidDB JDBC ドライバーは、Java Naming and Directory Interface (JNDI) をサポートしています。

JNDI により、アプリケーションは共通のインターフェースを使用してネーミング・サービスとディレクトリー・サービスにアクセスできます。JNDI はサービスではなく、一連のインターフェースです。これらのインターフェースによって、アプリケーションはファイル・システム、Lightweight Directory Access Protocol (LDAP) などのディレクトリー・サービス、Network Information System (NIS)、Common Object Request Broker Architecture (CORBA) などの分散オブジェクト・システム、Java リモート・メソッド呼び出し (RMI)、および Enterprise JavaBeans™ (EJB) といった数多くのさまざまなディレクトリー・サービスにアクセスできます。

コード例

このトピックでは、solidDB JDBC ドライバーを使用する 4 つの Java コード例を示します。

Java コード例 1: sample1.java

```
/**
 *      JDBC サンプル・アプリケーション sample1
 *
 *
 *      この単純な JDBC アプリケーションは、Solid JDBC ドライバーを使用して
```

```
 *      以下の処理を行います。
 *
 *  1. JDBC ドライバー・マネージャー・サービスを使用してドライバーを登録します。
 *  2. ユーザーに対して、有効な JDBC 接続ストリングの入力を促すプロンプトを
 *     出します。
 *  3. ドライバーを使用して、Solid に接続します。
 *  4. 1 つの照会用のステートメントを作成します。
 *     'SELECT TABLE_SCHEMA,TABLE_NAME,TABLE_TYPE FROM TABLES'
 *     これは、Solid システム表の 1 つからデータを読み取るためのものです。
 *  5. 照会を実行します。
 *  6. 結果セットのすべての行をフェッチおよびダンプします。
 *  7. 接続を閉じます。
 *
 *  アプリケーションを作成し実行するには、以下の手順を実行します。
 *
 *  1. 作業用の Java 開発環境を用意しておきます。
 *  2. 接続する Solid をインストールして開始します。サーバーが
 *     稼働中であることを確認します。
 *  3. 開発/稼働環境で使用する CLASSPATH 定義に
 *     SolidDriver2.0.jar を追加します。
 *  4. sample1.java ファイルに基づいて、Java プロジェクトを作成します。
 *  5. アプリケーションを作成し、実行します。
 *
 *  詳しくは、solidDB パッケージに含まれる、readme.txt ファイルを
 *  参照してください。
 *
 */

import java.io.*;

public class sample1 {

  public static void main (String args[]) throws Exception
  {
    java.sql.Connection conn;
    java.sql.ResultSetMetaData meta;
    java.sql.Statement stmt;
    java.sql.ResultSet result;
    int i;

    System.out.println("JDBC sample application starts...");
    System.out.println("Application tries to register the driver.");

    // これは、ドライバーの登録に関して推奨されている方法です。
    java.sql.Driver d =
       (java.sql.Driver)Class.forName("solid.jdbc.SolidDriver").newInstance();

    System.out.println("Driver succesfully registered.");

    // ユーザーは、接続ストリングの入力を求められます。
    System.out.println(
        "Now sample application needs a connectstring in format:¥n"
        );
    System.out.println(
        "jdbc:solid://<host>:<port>/<user name>/<password>¥n"
        );
    System.out.print("¥nPlease enter the connect string >");
    BufferedReader reader =
        new BufferedReader(new InputStreamReader(System.in));
    String sCon = reader.readLine();

    // 次に、接続を試行します。
    System.out.println("Attempting to connect :" + sCon);
    conn = java.sql.DriverManager.getConnection(sCon);

    System.out.println("SolidDriver succesfully connected.");
```

```
    String sQuery = "SELECT TABLE_SCHEMA,TABLE_NAME,TABLE_TYPE FROM TABLES";

    stmt= conn.createStatement();

    result = stmt.executeQuery(sQuery);
    System.out.println("Query executed and result set obtained.");

    // 取得した結果セットに関する情報を含む
    // メタデータ・オブジェクトを取得します。
    System.out.println("Obtaining metadata information.");
    meta = result.getMetaData();
    int cols = meta.getColumnCount();

    System.out.println("Metadata information for columns is as follows:");
    // 結果セットに関する列情報をダンプします。
    for (i=1; i <= cols; i++)
    {
        System.out.println("Column i:"+i+"  "+meta.getColumnName(i)+ "," +
             meta.getColumnType(i) + "," + meta.getColumnTypeName(i));
    }

    // 最後に、結果セットをダンプします。
    System.out.println("Starting to dump result set.");
    int cnt = 1;
    while(result.next())
    {
        System.out.print("¥nRow "+cnt+" : ");
        for (i=1; i <= cols; i++) {
            System.out.print(result.getString(i)+"¥t");
        }
        cnt++;
    }

    stmt.close();

    conn.close();
    // これで、すべて終了です。
    System.out.println("¥nResult set dumped. Sample application finishes.");
  }
}
```

Java コード例 1 の出力

```
Solid¥DatabaseEngine4.1¥jdbc¥samples>java sample1.java
JDBC sample application starts...
Application tries to register the driver.
Driver succesfully registered.
Now sample application needs a connectstring in format:

jdbc:solid://<host>:<port>/<user name>/<password>

Please enter the connect string >jdbc:solid://localhost:1313/dba/dba
Attempting to connect :jdbc:solid://localhost:1313/dba/dba
SolidDriver succesfully connected.
Query executed and result set obtained.
Obtaining metadata information.
Metadata information for columns is as follows:
Column i:1  TABLE_SCHEMA,12,VARCHAR
Column i:2  TABLE_NAME,12,VARCHAR
Column i:3  TABLE_TYPE,12,VARCHAR
Starting to dump result set.

Row 1 : _SYSTEM SYS_TABLES      BASE TABLE
Row 2 : _SYSTEM SYS_COLUMNS     BASE TABLE
Row 3 : _SYSTEM SYS_USERS       BASE TABLE
Row 4 : _SYSTEM SYS_UROLE       BASE TABLE
Row 5 : _SYSTEM SYS_RELAUTH     BASE TABLE
```

```
Row 6 : _SYSTEM SYS_ATTAUTH     BASE TABLE
Row 7 : _SYSTEM SYS_VIEWS       BASE TABLE
Row 8 : _SYSTEM SYS_KEYPARTS    BASE TABLE
Row 9 : _SYSTEM SYS_KEYS        BASE TABLE
Row 10 : _SYSTEM        SYS_CARDINAL    BASE TABLE
Row 11 : _SYSTEM        SYS_INFO        BASE TABLE
Row 12 : _SYSTEM        SYS_SYNONYM     BASE TABLE
Row 13 : _SYSTEM        TABLES  VIEW
Row 14 : _SYSTEM        COLUMNS VIEW
Row 15 : _SYSTEM        SQL_LANGUAGES   BASE TABLE
Row 16 : _SYSTEM        SERVER_INFO     VIEW
Row 17 : _SYSTEM        SYS_TYPES       BASE TABLE
Row 18 : _SYSTEM        SYS_FORKEYS     BASE TABLE
Row 19 : _SYSTEM        SYS_FORKEYPARTS BASE TABLE
Row 20 : _SYSTEM        SYS_PROCEDURES  BASE TABLE
Row 21 : _SYSTEM        SYS_TABLEMODES  BASE TABLE
Row 22 : _SYSTEM        SYS_EVENTS      BASE TABLE
Row 23 : _SYSTEM        SYS_SEQUENCES   BASE TABLE
Row 24 : _SYSTEM        SYS_TMP_HOTSTANDBY      BASE TABLE
Result set dumped. Sample application finishes.
```

Java コード例 2: sample2.java

```
/**
 *      JDBC サンプル・アプレット sample2
 *
 *
 *      この単純な JDBC アプレットは、Solid ネイティブ JDBC ドライバーを使用して
 *      以下の処理を行います。
 *
 *  1. JDBC ドライバー・マネージャー・サービスを使用してドライバーを登録します。
 *  2. ドライバーを使用して、Solid に接続します。
 *     使用する URL は、sample2.html から読み取ります。
 *  3. 指定した SQL ステートメントを実行します。
 *
 *  アプリケーションを作成し実行するには、以下の手順を実行します。
 *
 *  1. 作業用の Java 開発環境を用意しておきます。
 *  2. 接続する Solid をインストールして開始します。サーバーが
 *  稼働中であることを確認します。
 *  3. 開発/稼働環境で使用する CLASSPATH 定義に
 *     SolidDriver2.0.jar を追加します。
 *  4. sample2.java ファイルに基づいて、Java プロジェクトを作成します。
 *  5. アプリケーションを作成し、実行します。sample2.html に
 *     ご使用の環境を指す有効な URL が定義されていることを確認します。
 *
 *  詳しくは、IBM solidDB Development Kit パッケージに含まれる
 *  readme.txt ファイルを参照してください。
 *
 */

import java.util.*;
import java.awt.*;
import java.applet.Applet;
import java.net.URL;
import java.sql.*;

public class sample2 extends Applet {
  TextField textField;
  static TextArea textArea;

  String url = null;
  Connection con = null;

  public void init() {
      // URL の有効な値は、例えば以下のようになります。
      // url = "jdbc:solid://localhost:1313/dba/dba";
```

```
    url = getParameter("url");

    textField = new TextField(40);
    textArea = new TextArea(10, 40);
    textArea.setEditable(false);

    Font font = textArea.getFont();
    Font newfont = new Font("Monospaced", font.PLAIN, 12);
    textArea.setFont(newfont);

    // アプレットにコンポーネントを追加します。
    GridBagLayout gridBag = new GridBagLayout();
    setLayout(gridBag);
    GridBagConstraints c = new GridBagConstraints();
    c.gridwidth = GridBagConstraints.REMAINDER;

    c.fill = GridBagConstraints.HORIZONTAL;
    gridBag.setConstraints(textField, c);
    add(textField);

    c.fill = GridBagConstraints.BOTH;
    c.weightx = 1.0;
    c.weighty = 1.0;
    gridBag.setConstraints(textArea, c);
    add(textArea);

    validate();

    try {
        // Solid JDBC ドライバーをロードします。
        Driver d =
           (Driver)Class.forName ("solid.jdbc.SolidDriver").newInstance();

        // ドライバーへの接続を試行します。
        con = DriverManager.getConnection (url);

        // 接続できなかったのであれば例外が
        // スローされています。したがって、ここに到達した場合、
        // URL への接続が成功しています。
        // 接続で生成された警告を確認して
        // 表示します。
        checkForWarning (con.getWarnings ());

        // DatabaseMetaData オブジェクトを取得し、
        // 接続に関する情報を表示します。
        DatabaseMetaData dma = con.getMetaData ();

        textArea.appendText("Connected to " + dma.getURL() + "¥n");
        textArea.appendText("Driver       " + dma.getDriverName() + "¥n");
        textArea.appendText("Version      " + dma.getDriverVersion() + "¥n");
    }
    catch (SQLException ex) {
        printSQLException(ex);
    }
    catch (Exception e) {
        textArea.appendText("Exception:  " + e + "¥n");
    }
}

public void destroy() {
    if (con != null) {
        try {
            con.close();
        }
        catch (SQLException ex) {
            printSQLException(ex);
```

```
            }
            catch (Exception e) {
                textArea.appendText("Exception:  " + e + "¥n");
            }
        }
    }

    public boolean action(Event evt, Object arg) {
        if (con != null) {
            String sqlstmt = textField.getText();
            textArea.setText("");
            try {
                // Statement オブジェクトを作成して、
                // ドライバーに SQL ステートメントをサブミットできるようにします。
                Statement stmt = con.createStatement ();
                // 行制限を設定します。
                stmt.setMaxRows(50);
                // 照会をサブミットし、ResultSet オブジェクトを作成します。
                ResultSet rs = stmt.executeQuery (sqlstmt);

                // 結果セットから、すべての列と行を表示します。
                textArea.setVisible(false);
                dispResultSet (stmt,rs);
                textArea.setVisible(true);

                // 結果セットを閉じます。
                rs.close();

                // ステートメントを閉じます。
                stmt.close();
            }
            catch (SQLException ex) {
                printSQLException(ex);
            }
            catch (Exception e) {
                textArea.appendText("Exception:  " + e + "¥n");
            }
            textField.selectAll();
        }
        return true;
    }

      //---------------------------------------------------------------------
      // checkForWarning
      // 警告を確認し、表示します。警告が存在する場合、
      // true を返します。
//---------------------------------------------------------------------

      private static boolean checkForWarning (SQLWarning warn)
              throws SQLException
      {
          boolean rc = false;

          // SQLWarning オブジェクトが指定されている場合、
          // 警告メッセージを表示します。複数の警告が
          // チェーニングされている可能性があることに注意してください。

          if (warn != null) {
              textArea.appendText("¥n*** Warning ***¥n");
              rc = true;
              while (warn != null) {
                  textArea.appendText("SQLState: " +
                      warn.getSQLState () + "¥n");
                  textArea.appendText("Message:  " +
                      warn.getMessage () + "¥n");
                  textArea.appendText("Vendor:   " +
                      warn.getErrorCode () + "¥n");
```

```
            textArea.appendText("¥n");
            warn = warn.getNextWarning ();
        }
    }
    return rc;
}

//-------------------------------------------------------------------
// dispResultSet
// 指定された結果セット内のすべての列と行を表示します。
//-------------------------------------------------------------------

private static void dispResultSet (Statement sta, ResultSet rs)
    throws SQLException
{
    int i;

    // ResultSetMetaData を取得します。これは、
    // 列見出しに使用されます。
    ResultSetMetaData rsmd = rs.getMetaData ();

    // 結果セット内の列数を取得します。
    int numCols = rsmd.getColumnCount ();
    if (numCols == 0) {
        textArea.appendText("Updatecount is "+sta.getUpdateCount());
        return;
    }

    // 列見出しを表示します。
                for (i=1; i<=numCols; i++) {
        if (i > 1) {
            textArea.appendText("¥t");
        }
        try {
            textArea.appendText(rsmd.getColumnLabel(i));
        }
        catch(NullPointerException ex) {
            textArea.appendText("null");
        }
    }
    textArea.appendText("¥n");

    // 結果セットの終端までフェッチしながら、データを表示します。
    boolean more = rs.next ();
    while (more) {

        // 各列をループして、列データを取得し、
        // 表示します。
        for (i=1; i<=numCols; i++) {
            if (i > 1) {
                textArea.appendText("¥t");
            }
            try {
                textArea.appendText(rs.getString(i));
            }
            catch(NullPointerException ex) {
                textArea.appendText("null");
            }
        }
        textArea.appendText("¥n");

        // 次の結果セット行をフェッチします。
        more = rs.next ();
    }
}

private static void printSQLException(SQLException ex)
```

```
    {
            // SQLException が生成されています。それをキャッチし、
            // エラー情報を表示します。複数の
            // エラー・オブジェクトがチェーニングされて
            // いる可能性があることに注意してください。

            textArea.appendText("¥n*** SQLException caught ***¥n");

            while (ex != null) {
                textArea.appendText("SQLState: " +
                    ex.getSQLState () + "¥n");
                textArea.appendText("Message:  " +
                    ex.getMessage () + "¥n");
                textArea.appendText("Vendor:   " +
                    ex.getErrorCode () + "¥n");
                textArea.appendText("¥n");
                ex = ex.getNextException ();
            }
    }
}
```

Java コード例 3: sample3.java

```
/**
 *      JDBC サンプル・アプリケーション sample3
 *
 *
 *      この単純な JDBC アプリケーションは、Solid JDBC ドライバーを使用して
 *      以下の処理を行います。
 *
 *  1. JDBC ドライバー・マネージャー・サービスを使用してドライバーを登録します。
 *  2. ユーザーに対して、有効な JDBC 接続ストリングの入力を促すプロンプトを
 *     出します。
 *  3. ドライバーを使用して、Solid に接続します。
 *  4. プロシージャー sample3 をドロップして作成します。そのプロシージャーが
 *     存在しない場合、関連する例外をダンプします。
 *  5. java.sql.Statement を使用してそのプロシージャーを呼び出します。
 *  6. 結果セットのすべての行をフェッチおよびダンプします。
 *  7. 接続を閉じます。
 *
 *  アプリケーションを作成し実行するには、以下の手順を実行します。
 *
 *  1. 作業用の Java 開発環境を用意しておきます。
 *  2. 接続する Solid をインストールして開始します。サーバーが
 *     稼働中であることを確認します。
 *  3. 開発/稼働環境で使用する CLASSPATH 定義に
 *     SolidDriver2.0.jar を追加します。
 *  4. sample3.java ファイルに基づいて、Java プロジェクトを作成します。
 *  5. アプリケーションを作成し、実行します。
 *
*  詳しくは、IBM solidDB Development Kit パッケージに含まれる
*  readme.txt ファイルを参照してください。
*/

import java.io.*;
import java.sql.*;

public class sample3 {

    static Connection conn;
    public static void main (String args[]) throws Exception
    {
        System.out.println("JDBC sample application starts...");
        System.out.println("Application tries to register the driver.");

        // これは、ドライバーの登録に関して推奨されている方法です。
        Driver d = (Driver)Class.forName("solid.jdbc.SolidDriver").newInstance();
```

```
            System.out.println("Driver succesfully registered.");

            // ユーザーは、接続ストリングの入力を求められます。
            System.out.println(
                "Now sample application needs a connectstring in format:¥n"
                );
            System.out.println(
                "jdbc:solid://<host>:<port>/<user name>/<password>¥n"
                );
            System.out.print("¥nPlease enter the connect string >");
            BufferedReader reader =
            new BufferedReader(new InputStreamReader(System.in));
            String sCon = reader.readLine();

            // 次に、接続を試行します。
            System.out.println("Attempting to connect :" + sCon);
            conn = DriverManager.getConnection(sCon);

            System.out.println("SolidDriver succesfully connected.");

            DoIt();

            conn.close();
            // これで、すべて終了です。
            System.out.println(
                "¥nResult set dumped. Sample application finishes."
);
        }


        static void DoIt() {
            try {
                createprocs();
                PreparedStatement pstmt = conn.prepareStatement("call sample3(?)");
                // パラメーター値を設定します。
                pstmt.setInt(1,10);

                ResultSet rs = pstmt.executeQuery();
                if (rs != null) {
                    ResultSetMetaData md = rs.getMetaData();
                    int cols = md.getColumnCount();
                    int row = 0;
                    while (rs.next()) {
                        row++;
                        String ret = "row "+row+": ";
                        for (int i=1;i<=cols;i++) {
                            ret = ret + rs.getString(i) + " ";
                        }
                        System.out.println(ret);
                    }
                }
                conn.commit();
            }
            catch (SQLException ex) {
                printexp(ex);
            }
            catch (java.lang.Exception ex) {
                ex.printStackTrace ();
            }

        }

        static void createprocs() {
            Statement stmt = null;
            String proc = "create procedure sample3 (limit integer)" +
                          "returns (c1 integer, c2 integer) " +
```

```
                    "begin " +
                    "  c1 := 0;" +
                    "  while c1 < limit loop " +
                    "    c2 := 5 * c1;" +
                    "    return row;" +
                    "    c1 := c1 + 1;" +
                    "  end loop;" +
                    "end";

        try {
            stmt = conn.createStatement();
            stmt.execute("drop procedure sample3");
        } catch (SQLException ex) {
            printexp(ex);
        }

        try {
            stmt.execute(proc);
        } catch (SQLException ex) {
            printexp(ex);
            System.exit(-1);
        }
    }

    public static void printexp(SQLException ex) {
        System.out.println("¥n*** SQLException caught ***");
        while (ex != null) {
            System.out.println("SQLState: " + ex.getSQLState());
            System.out.println("Message:  " + ex.getMessage());
            System.out.println("Vendor:   " + ex.getErrorCode());
            ex = ex.getNextException ();
        }
    }

}
```

Java コード例 4: sample4.java

```
/**
 *      JDBC サンプル・アプリケーション sample4
 *
 *
 *      この単純な JDBC アプリケーションは、Solid JDBC ドライバーを使用して
 *      以下の処理を行います。
 *
 *  1. JDBC ドライバー・マネージャー・サービスを使用してドライバーを登録します。
 *  2. ユーザーに対して、有効な JDBC 接続ストリングの入力を促すプロンプトを
 *     出します。
 *  3. ドライバーを使用して、Solid に接続します。
 *  4. 表 sample4 をドロップして作成します。その表が
 *     存在しない場合、関連する例外をダンプします。
 *  5. 引数として指定されたファイルをデータベースに挿入します (Store メソッド)。
 *  6. この 'blob' を読み取り、out.tmp ファイルに戻します (Restore メソッド)。
 *  7. 接続を閉じます。
 *
 *  アプリケーションを作成し実行するには、以下の手順を実行します。
 *
 *  1. 作業用の Java 開発環境を用意しておきます。
 *  2. 接続する Solid をインストールして開始します。サーバーが
 *     稼働中であることを確認します。
 *  3. 開発/稼働環境で使用する CLASSPATH 定義に
 *     SolidDriver2.0.jar を追加します。
 *  4. sample4.java ファイルに基づいて、Java プロジェクトを作成します。
 *  5. アプリケーションを作成し、実行します。
 *
 *  詳しくは、solidDB Development Kit パッケージに含まれる
 *  readme.txt ファイルを参照してください。
```

```
*
 */

import java.io.*;
import java.sql.*;

public class sample4 {

    static Connection conn;
    public static void main (String args[]) throws Exception
    {
        String filename = null;
        String tmpfilename = null;

        if (args.length < 1) {
            System.out.println("usage: java sample4 <infile>");
            System.exit(0);
        }
        filename = args[0];
        tmpfilename = "out.tmp";
        System.out.println("JDBC sample application starts...");
        System.out.println("Application tries to register the driver.");

        // これは、ドライバーの登録に関して推奨されている方法です。
        Driver d = (Driver)Class.forName("solid.jdbc.SolidDriver").newInstance();

        System.out.println("Driver succesfully registered.");

        // ユーザーは、接続ストリングの入力を求められます。
        System.out.println(
            "Now sample application needs a connectstring in format:¥n"
            );
        System.out.println(
            "jdbc:solid://<host>:<port>/<user name>/<password>¥n"
            );
        System.out.print("¥nPlease enter the connect string >");
        BufferedReader reader =
        new BufferedReader(new InputStreamReader(System.in));
        String sCon = reader.readLine();

        // 次に、接続を試行します。
        System.out.println("Attempting to connect :" + sCon);
        conn = DriverManager.getConnection(sCon);

        System.out.println("SolidDriver succesfully connected.");

        // 表 sample4 をドロップし、作成します。
        createsample4();
        // その表にデータを挿入します。
        Store(filename);
        // そして、リストアします。
        Restore(tmpfilename);

        conn.close();
        // これで、すべて終了です。
        System.out.println("¥nSample application finishes.");
    }


    static void Store(String filename) {
        String sql = "insert into sample4 values(?,?)";
        FileInputStream inFileStream ;
        try {
            File f1 = new File(filename);
            int blobsize = (int)f1.length();
            System.out.println("Inputfile size is "+blobsize);
            inFileStream = new FileInputStream(f1);
```

```
            PreparedStatement stmt = conn.prepareStatement(sql);
            stmt.setLong(1, System.currentTimeMillis());
            stmt.setBinaryStream(2, inFileStream, blobsize);
            int rows = stmt.executeUpdate();
            stmt.close();
            System.out.println(""+rows+" inserted.");
            conn.commit();
        }
        catch (SQLException ex) {
            printexp(ex);
        }
        catch (java.lang.Exception ex) {
            ex.printStackTrace ();
        }

    }

    static void Restore(String filename) {
        String sql = "select id,blob from sample4";
        FileOutputStream outFileStream ;
        try {
            File f1 = new File(filename);
            outFileStream = new FileOutputStream(f1);

            PreparedStatement stmt = conn.prepareStatement(sql);
            ResultSet rs = stmt.executeQuery();
            int readsize = 0;
            while (rs.next()) {
                InputStream in = rs.getBinaryStream(2);
                byte bytes[] = new byte[8*1024];
                int nRead = in.read(bytes);
                while (nRead != -1) {
                    readsize = readsize + nRead;
                    outFileStream.write(bytes,0,nRead);
                    nRead = in.read(bytes);
                }

            }
            stmt.close();
            System.out.println("Read "+readsize+" bytes from database");
        }
        catch (SQLException ex) {
            printexp(ex);
        }
        catch (java.lang.Exception ex) {
            ex.printStackTrace ();
        }

    }


    static void createsample4() {
        Statement stmt = null;
        String proc = "create table sample4 (" +
                      "id numeric not null primary key,"+
                      "blob long varbinary)";

        try {
            stmt = conn.createStatement();
            stmt.execute("drop table sample4");
        } catch (SQLException ex) {
            printexp(ex);
        }

        try {
            stmt.execute(proc);
```

```
        } catch (SQLException ex) {
            printexp(ex);
            System.exit(-1);
        }
    }

    static void printexp(SQLException ex) {
        System.out.println("¥n*** SQLException caught ***");
        while (ex != null) {
            System.out.println("SQLState: " + ex.getSQLState());
            System.out.println("Message:  " + ex.getMessage());
            System.out.println("Vendor:   " + ex.getErrorCode());
            ex = ex.getNextException ();
        }
    }

}
```

solidDB JDBC ドライバー型変換マトリックス

このトピックの型変換マトリックスは、solidDB JDBC ドライバーがサポートする Java データ型から SQL データ型への変換方法を示します。

このマトリックスは、データを取得する ResultSet.getXXX メソッドとデータを設定する ResultSet.setXXX メソッドの両方に適用されます。X は、solidDB JDBC ドライバーがそのメソッドをサポートすることを意味します。

表 71. Java データ型から SQL データ型への変換

データの取得と設定に適用する Java データ型	**TINYINT**	**SMALLINT**	**INTEGER**	**REAL**	**FLOAT**	**DOUBLE**	**DECIMAL**	**NUMERIC**	**CHAR**	**VARCHAR**	**LONGVARCHAR**	**WCHAR**	**WVARCHAR**	**LONGWVARCHAR**	**BINARY**	**VARBINARY**	**LONGVARBINARY**	**DATE**	**TIME**	**TIMESTAMP**
getArray/setArray																				
getBlob/setBlob																				
getByte/setByte	X	X	X	X	X	X	X	X	X	X	X	X	X	X						
getCharacterStream/ setCharacterStream									X	X	X	X	X	X	X	X	X	X	X	X
getClob/setClob																				
getShort/setShort	X	X	X	X	X	X	X	X	X	X	X									
getInt/setInt	X	X	X	X	X	X	X	X	X	X	X									
getlong/setLong	X	X	X	X	X	X	X	X	X	X	X									

表 71. Java データ型から SQL データ型への変換 (続き)

データの取得と設定に適用する Java データ型	TINYINT	SMALLINT	INTEGER	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	CHAR	VARCHAR	LONGVARCHAR	WCHAR	WVARCHAR	LONGWVARCHAR	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
getfloat/setfloat	X	X	X	X	X	X	X	X	X	X	X									
getDouble/setDouble	X	X	X	X	X	X	X	X	X	X	X									
getBigDecimal/setBigDecimal	X	X	X	X	X	X	X	X	X	X	X									
getRef/setRef																				
getBoolean/setBoolean	X	X	X	X	X	X	X	X	X	X	X									
getString/setString	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
getBytes/setBytes									X	X	X	X	X	X	X	X	X			
getDate/setDate									X	X	X	X	X	X				X		X
getTime/setTime									X	X	X	X	X	X					X	X
getTimestamp/setTimestamp									X	X	X	X	X	X				X		X
getAsciiStream/setAsciiStream									X	X	X	X	X	X	X	X	X			
getUnicodeStream/setUnicodeStream									X	X	X	X	X	X	X	X	X			
getBinaryStream/setBinaryStream									X	X	X	X	X	X	X	X	X			
getObject/setObject	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

5 solidDB SA の使用

このセクションでは、*solidDB SA* とも呼ばれている solidDB アプリケーション・プログラミング・インターフェース (API) の使用方法について説明します。

solidDB SA は、solidDB データベース管理製品にアクセスするための、低レベルの C 言語クライアント・ライブラリーです。solidDB SA は、solidDB 製品内部にあるレイヤーです。通常は、ODBC または JDBC などの業界標準ベースのインターフェースの使用を推奨します。ただし、書き込みロードが多い環境では (BATCH INSERT および BATCH UPDATE)、solidDB SA によって、パフォーマンスを大幅に向上することができます。

solidDB SA とは

solidDB SA は solidDB データベース製品に接続するための C 言語クライアント・ライブラリーです。このライブラリーを solidDB 製品内部で使用して、solidDB データベース表内のデータにアクセスします。ライブラリーには 90 の関数が含まれており、データベースを接続し、カーソル・ベースの操作を実行するための低レベルのメカニズムを提供します。

ODBC や JDBC などの業界標準インターフェースと比べると、solidDB SA はデータベース・サーバーに送信するネットワーク・メッセージをより柔軟に構成することができます。読み取りまたは書き込みのパフォーマンス (例えばバッチ挿入や主キーの参照など) を最適化する必要のあるアプリケーションでは、solidDB SA を使用することで、パフォーマンスを大幅に向上できます。例えば、バッチ挿入中にサイトでのパフォーマンス・ボトルネックが発生する場合、solidDB SA はボトルネックを減少させることができます。これは、solidDB SA によって、単一のネットワーク・メッセージやリモート・プロシージャー・コール内で挿入する複数の行の受け渡しが行えるからです。

注: パフォーマンス・ボトルネックが読み取り操作で発生している場合は、solidDB SA を使用しても、パフォーマンスはわずかしか向上しません。

solidDB SA インターフェースは、SQL パーサー、インタープリター、オプティマイザーをバイパスします。solidDB SA 経由で SQL を使用していない限り、solidDB SA を使用して、結果セットにアクセスすることができます。SQL での結果セットのリトリーブが必要な場合は、ODBC などの業界標準 API か、ODBC ベースの solidDB Light Client を使用しなければなりません。

solidDB SA を使用するには、既存のインターフェースを変換する必要があります。そのため、以下のようなパフォーマンス向上のための他の方法を試した後でのみ (成果がほとんどなかった場合)、solidDB SA を使用することを推奨します。

- 最適な行順序になるように列を書き込むか、主キーによる列の索引付けを行います。そうしないと、solidDB は、データベースに挿入された順序で行をディスクに格納します。

- 不要な索引を除去します。例えば、表の行の 15% を超える選択を行う照会は、表のフル・スキャンによるほうが高速に行える可能性があります。
- 100 から 200 行を挿入するたびにトランザクションをコミットして、トランザクション・サイズを最適化します。
- ストアード・プロシージャーを使用します。

solidDB SA の概要

solidDB SA は、共有メモリー・アクセス (SMA) およびリンク・ライブラリー・アクセス (LLA) とともに使用することができます。このトピックでは、solidDB SA を使用する前に必要な手順を説明します。

solidDB SA の使用を開始する前に、以下を必ず実行しておいてください。

1. ローカル・アプリケーションを作成する場合、SMA または LLA ライブラリー・ファイルが必要です。SMA および LLA ライブラリー・ファイルは、solidDB のインストール中にインストールされます。これらのライブラリーには、solidDB SA 関数とともに、すべてのサーバー機能が含まれています。
2. リモート・ユーザー・アプリケーションを作成する場合は、solidDB SA ライブラリー (例えば、Windows オペレーティング・システムの場合 `solidimpsa.lib`) をアプリケーションにリンクできるようにしておく必要があります。
3. solidDB を始動しておきます。必要であれば、solidDB SA の使用前に新しいデータベースを作成しておいてください。

開発環境のセットアップとサンプル・プログラムの作成

リンク・ライブラリー・アクセスまたは SA クライアント・ライブラリーでの solidDB SA ライブラリーを使用したアプリケーション・プログラムの作成は、通常の C/C++ プログラムの作成と同じです。

1. リンク・ライブラリー・アクセス・ライブラリー・ファイルまたは SA クライアント・ライブラリーをプロジェクトに挿入します。正しいファイル名については、「*IBM solidDB 共有メモリー・アクセスおよびリンク・ライブラリー・アクセス・ユーザー・ガイド*」のセクション『*LLA アプリケーションの作成と実行*』を参照してください。
2. 以下の solidDB SA ヘッダー・ファイルをインクルードします。このファイルは、solidDB SA ライブラリーをリンク・ライブラリー・アクセスまたは solidDB SA クライアント・ライブラリーで使用するアプリケーションで必要です。

   ```
   #include "sa.h"
   ```

   必要なその他のすべての solidDB SA ヘッダーを含むディレクトリーを、開発環境のインクルード・ディレクトリー設定に挿入します。
3. ソース・コードをコンパイルします。
4. プログラムをリンクします。

開発環境セットアップの検証

solidDB SA サンプル・プログラムを使用して、開発セットアップを検証することができます。これによって、コードを書かずに開発環境を検証できます。

開発環境で、以下を検証してください。

- Windows 環境では、TCP/IP サービスは、標準 DLL である wsock32.dll によって提供されます。これらのサービスをプロジェクトにリンクするには、wsock32.lib をリンカーの lib ファイル・リストに追加します。

サンプル・アプリケーションを使用したデータベースへの接続

solidDB SA では、データベースへの接続は、SaConnectT 構造によって表現されます。この構造は、関数 SaConnect を呼び出すことで確立されます。以下のサンプル・コードは、ローカル・マシン・ポート 1313 で TCP/IP プロトコルを listen しているデータベースへの接続を確立します。パスワードが DBA のユーザー・アカウント DBA がデータベース内で定義されています。

```
SaConnectT* scon;

scon = SaConnect("tcp localhost 1313", "dba", "dba");
if (scon == NULL)
{
    /* 接続に失敗しました。接続エラー・テキストを表示します。*/
    char* errstr;
    SaErrorInfo(NULL, &errstr, NULL);
    printf("%s¥n", errstr);
    return(1);
}
```

SQL を使用せずに solidDB SA を使用したデータの書き込み

solidDB SA では、カーソルを使用してデータを書き込みます。

削除および更新の操作の場合は、カーソルの作成後、検索を行って、更新および削除する行をカーソルが指すようにします。挿入操作の場合は、カーソルの作成後、挿入行がカーソルにすぐに書き込まれます。solidDB SA では、単一のネットワーク・メッセージ内で挿入する複数の行を引き渡すこともできます。

挿入操作の実行

挿入操作に必要な solidDB SA 関数を以下の表にリストします。

solidDB が特定の表にカーソルを作成した後、変数が列にバインドされ、行がカーソルに書き込まれ、カーソルがクローズします。

注: SaArrayInsert を使用して単一メッセージに複数の行を挿入する場合、明示的なフラッシュを実行して、行をデータベースに送信する必要があります。

表 72. 挿入操作手順

手順	SA 関数	コメント
1. カーソルの作成	SaCursorCreate	

表 72. 挿入操作手順 (続き)

手順	SA 関数	コメント
2. カーソルへの変数のバインド	SaCursorColData、 SaCursorColDate、 SaCursorColDateFormat、 SaCursorColDFloat、 SaCursorColDouble、 SaCursorColDynData、 SaCursorColDynstr、 SaCursorColFloat、 SaCursorColInt、 SaCursorColLong、 SaCursorColStr、 SaCursorColTime、 SaCursorColTimestamp	
3. カーソルのオープン	SaCursorOpen	
4. カーソルへの行の書き込み	複数行の場合は SaArrayInsert、1 行の場合は SaCursorInsert	必要に応じてループ内で実行します。
5. カーソルの解放	SaCursorFree	
6. サーバーへのネットワーク・メッセージのフラッシュ	SaArrayFlush	SaArrayInsert の使用時のみ必要です。

以下のコード・サンプルの抜粋では、SaArrayInsert 関数を使用して、単一のネットワーク・メッセージに 4 行のデータを書き込む方法を示します。このコードでは、SaArrayFlush を呼び出すことで、サーバーにすべての行がフラッシュされるので、同じネットワーク・メッセージでこれらの行を渡すことができます。

```
scur = SaCursorCreate(scon, "SAEXAMPLE");

/* 変数を列にバインドします。*/
SaCursorColInt(scur, "INTC", &intc);
SaCursorColStr(scur, "CHARC", &str);

/* カーソルをオープンします。*/
SaCursorOpen(scur);

/* 表に値を挿入します。列の値は、列にバインドされたユーザー変数から
 * 取得します。
 */
for (intc = 2; intc <= 5; intc++) {
    switch (intc) {
        case 2:
            str = "B";
            break;
        case 3:
            str = "C";
            break;
        case 4:
            str = "D";
            break;
        case 5:
            str = "E";
            break;
    }
    SaArrayInsert(scur);
}
```

```
/* カーソルをクローズします。*/
SaCursorFree(scur);

/* 挿入内容をサーバーにフラッシュします。*/
SaArrayFlush(scon, NULL);
```

更新および削除の操作の実行

以下の表に、基本的な更新および削除の操作に必要な solidDB SA 関数をリストします。

solidDB が特定の表にカーソルを作成した後、変数が表の列にバインドされ、カーソルがオープンします。実際の検索を開始する前に、削除する行を検索するための制約を設定します。更新する行が複数ある場合は、更新や削除の前に、各行を個別にフェッチする必要があります。操作後、カーソルが解放されます。 cursor is freed.

表 73. 更新および削除の操作手順

手順	SA 関数	コメント
1. カーソルの作成	SaCursorCreate	
2. カーソルへの変数のバインド	SaCursorColData、 SaCursorColDate、 SaCursorColDateFormat、 SaCursorColDFloat、 SaCursorColDouble、 SaCursorColDynData、 SaCursorColDynstr、 SaCursorColFloat、 SaCursorColInt、 SaCursorColLong、 SaCursorColStr、 SaCursorColTime、 SaCursorColTimestamp	
3. カーソルのオープン	SaCursorOpen	
4. 更新または削除する行の検索制約の設定	SaCursorEqual、SaCursorAtleast、 SaCursorAtmost	
5. 更新または削除する行の検索の開始	SaCursorSearch	
6. 更新または削除する行のフェッチ	SaCursorNext	
7. 実際の更新または削除の実行	SaCursorUpdate または SaCursorDelete	更新の場合は、ステップ 2 でバインドした変数に新しい値が入っている必要があります。
8. カーソルの解放	SaCursorFree	

以下のコード・サンプルの抜粋では、SaCursorUpdate を使用して表の行を更新する方法を示します。このコードでは、カーソルの作成後に SaCursorColInt および SaCursorColStr を使用して表の列にバインドされた変数に、更新のための新しい値が入っている点に注意してください。

```
scur = SaCursorCreate(scon, "SAEXAMPLE");

/* テスト表の列 INTC と CHARC に変数をバインドします。*/
SaCursorColInt(scur, "INTC", &intc);
SaCursorColStr(scur, "CHARC", &str);

/* カーソルをオープンします。*/
SaCursorOpen(scur);

/* 検索制約を設定します。*/
str = "D";
SaCursorEqual(scur, "CHARC");

/* 検索を開始します。*/
SaCursorSearch(scur);

/* 列をフェッチします。*/
SaCursorNext(scur);

/* カーソル内の現在行を更新します。*/
intc = 1000;
str = "D Updated";
SaCursorUpdate(scur);

/* カーソルをクローズします。*/
SaCursorFree(scur);
```

SQL を使用せずに solidDB SA を使用したデータの読み取り

照会操作に必要な solidDB SA 関数をこのトピックにリストします。

solidDB SA では、カーソルを使用してデータを照会します。照会データは、更新や削除の操作と同様の方法で検索します。カーソルが特定の表に作成され、表の列に変数がバインドされたら、カーソルがオープンします。照会する行の検索の制約は、実際の検索開始前に設定されます。複数の行を検索する場合は、各行を個別にフェッチする必要があります。すべての行をフェッチした場合、カーソルを解放する必要があります。

基本的には、すべての solidDB SA 照会は SQL ベースの照会と同様の方法で、solidDB オプティマイザーを使用します。索引選択方法は SQL と同じです。唯一の例外は、solidDB SA の検索では、索引の選択に ORDER BY を使用する点です。つまり、ORDER BY に最適な索引が選択されます。2 つの索引の適性が同じ場合、コストの低いほうが選択されます。照会は、SaCursorSearch を呼び出すたびに最適化されます。

注: Solid SA を使用する場合は、オプティマイザーのヒント機能を使用することはできません。

表 74. 照会操作手順

手順	SA 関数	コメント
1. カーソルの作成	SaCursorCreate	

表 74. 照会操作手順 (続き)

手順	SA 関数	コメント
2. カーソルへの変数のバインド	SaCursorColInt、SACursorColStr、およびその他のデータ型用の他の関数	
3. カーソルのオープン	SaCursorOpen	
4. 照会する行の検索制約の設定	SaCursorEqual、SaCursorAtleast、SaCursorAtmost	
5. 照会する行の検索の開始	SaCursorSearch	
6. 指定した基準に一致する行のフェッチ	SaCursorNext	必要に応じてループ内で実行します。
7. カーソルの解放	SaCursorFree	

例

```
/* データベース表にカーソルを作成します。*/
scur = SaCursorCreate(scon, "SAEXAMPLE");

/* テスト表の列に変数をバインドします。*/
rc = SaCursorColInt(scur, "INTC", &intc);
rc = SaCursorColStr(scur, "CHARC", &str);

/* カーソルをオープンします。*/
rc = SaCursorOpen(scur);
assert(rc == SA_RC_SUCC);

/* 検索制約を設定します。*/
str = "A";
rc = SaCursorAtleast(scur, "CHARC");
str = "C";
rc = SaCursorAtmost(scur, "CHARC");

/* 順序付け基準を設定します。*/
rc = SaCursorAscending(scur, "CHARC");

/* 検索を開始します。*/
rc = SaCursorSearch(scur);

/* 行をフェッチします。*/
for (i = 1; i <= 3; i++) {
    rc = SaCursorNext(scur);
    switch (intc) {
        case 1:
            assert(strcmp(str, "A") == 0);
            break;
        case 2:
            assert(strcmp(str, "B") == 0);
            break;
        case 3:
            assert(strcmp(str, "C") == 0);
            break;
    }
```

```
    }

    /* カーソルをクローズします。*/
    SaCursorFree(scur);
}
```

solidDB SA を使用した SQL ステートメントの実行

SQL パーサーと SQL 変換処理をバイパスするだけでなく、solidDB SA では、直接 SaSQLExecDirect 関数を使用して、SQL ステートメントを限定的に実行することができます。

この関数は、CREATE TABLE などの単純 SQL ステートメントを実行するために設計されています。SQL 結果セットをリトリーブする必要がある場合は、ODBC や solidDB Light Client などの別のプログラミング・インターフェースを使用する必要があります。

例

```
/* テスト表と索引を作成します。*/
SaSQLExecDirect(scon,
        "CREATE TABLE SAEXAMPLE(INTC INTEGER, CHARC VARCHAR)");
SaSQLExecDirect(scon,
        "CREATE INDEX SAEXAMPLE_I1 ON SAEXAMPLE (CHARC)");
```

トランザクションと自動コミット・モード

デフォルトでは、solidDB SA は自動コミット・モードで実行されます。

トランザクションを明示的に開始する SaTransBegin 関数を呼び出すことで、自動コミット・モードをオフに切り替えることができます。このモードでは、トランザクションは、SaTransCommit 関数を使用してコミットするか、SaTransRollback を使用してロールバックします。

注: トランザクションのコミット後、solidDB SA は自動コミット・モードの設定に戻ります。

自動コミット・モードでは、挿入 (SaCursorInsert)、更新 (SaCursorUpdate)、または削除 (SaCursorDelete) の後すぐにトランザクションがコミットされます。SaArrayInsert を使用している場合でも、自動コミットの使用時は各レコードが別々のトランザクションに挿入されることに注意してください (詳しくは、 149 ページの『SaArrayInsert』を参照してください)。SaArrayInsert 関数を使用して複数の行を挿入する際にパフォーマンスを向上するには、SaTransBegin および SaTransCommit を使用して、複数の挿入を単一トランザクションにまとめます。

データベース・エラーの処理

このセクションには、データベース・エラーの処理に関する情報が含まれています。

solidDB SA は ODBC のようなエラー処理機能は提供していません。通常、solidDB SA 関数は、成功した場合は SA_RC_SUCC または要求したオブジェクトを指すポインターを返します。成功しなかった場合は、戻り値は solidDB SA エラー・コー

ド (以下のセクションの表を参照) のいずれか、または NULL になります。エラーがデータベース・エラーの場合は、エラー・テキストが SaErrorInfo 関数から返されます。

```
if (scon == NULL) {
        /* 接続に失敗しました。接続エラー・テキストを表示します。*/
        char* errstr;
        SaErrorInfo(NULL, &errstr, NULL);
        printf("%s¥n", errstr);
        return(1);
    }
```

関数 SaCursorErrorInfo は、最後のカーソル操作が失敗した場合は、エラー・テキストを返します。SaErrorInfo には接続パラメーターがあるので、その接続に該当する最後のエラーを返しますが、SaCursorErrorInfo にはカーソル・パラメーターがあるので、そのカーソルの最後のエラーを返すことに注意してください。

solidDB SA 関数のエラー・コードとメッセージ

以下に、solidDB SA 関数で考えられる戻りコードを示します。これらのエラー・コードはすべて、sa.h ファイルで定義されています。

表 75. solidDB SA 関数の戻りコード

エラー・コード	意味
SA_RC_SUCC	操作が成功しました。
SA_RC_END	操作が完了しました。
SA_ERR_FAILED	操作が失敗しました。
SA_ERR_CURNOTOPENED	カーソルがオープンしていません。
SA_ERR_CUROPENED	カーソルがオープンしています。
SA_ERR_CURNOSEARCH	カーソル内でアクティブな検索がありません。
SA_ERR_CURSEARCH	カーソル内でアクティブな検索があります。
SA_ERR_ORDERBYILL	「順序付け」指定が正しくありません。
SA_ERR_COLNAMEILL	列名が正しくありません。
SA_ERR_CONSTRILL	制約が正しくありません。
SA_ERR_TYPECONVILL	型変換が正しくありません。
SA_ERR_UNIQUE	ユニーク制約違反です。
SA_ERR_LOSTUPDATE	並行性競合。2 つのトランザクションで同じ行を更新または削除しました。
SA_ERR_SORTFAILED	検索結果セットのソートに失敗しました。

表 75. solidDB SA 関数の戻りコード (続き)

エラー・コード	意味
SA_ERR_CHSETUNSUPP	サポートされない文字セットです。
SA_ERR_CURNOROW	カーソル内に現在行がありません。
SA_ERR_COLISNOTNULL	NOT NULL 列に NULL 値が指定されました。
SA_ERR_LOCALSORT	結果セットがローカルにソートされています。行を更新または削除できません。
SA_ERR_COMERROR	通信エラーです。接続が失われました。
SA_ERR_NOSTRCONSTR	制約のストリングが欠落しています。
SA_ERR_ILLENUMVAL	数値が正しくありません。
SA_ERR_COLNOTBOUND	列がバインドされていません。
SA_ERR_CALLNOSUP	操作がサポートされていません。*
SA_ERR_RPCPARAM	RPC パラメーター・エラーです。
SA_ERR_TABLENOTFOUND	表が検出されません。
SA_ERR_READONLY	接続が読み取り専用です。
SA_ERR_ILLPARAMCOUNT	パラメーターの数が間違っています。
SA_ERR_INVARG	無効な引数です。
SA_ERR_INVCALLSEQ	無効な呼び出しシーケンスです。

注: SaArray* 関数はリンク・ライブラリー・アクセスではサポートされておらず、ネットワーク・クライアント・ライブラリーでのみ機能します。リンク・ライブラリー・アクセスでは、SA_ERR_CALLNOSUP を返します。

solidDB SA に関する特記事項

このトピックには、solidDB SA に関する重要な情報や制限事項が含まれています。

solidDB SA およびバイナリー・ラージ・オブジェクト (BLOB)

現在、solidDB SA は BLOB ストリームをサポートしておらず、属性値の最大サイズは 32K に制限されています。

SaCursorCol* 関数と solidDB SQL がサポートしているデータ型

SaCursorColXXX() 関数は、XXX 型の変数を、指定された列にバインドします。例えば、SaCursorColInt 関数は、int 型の変数を、指定された列にバインドします。変数を列にバインドする場合、通常、変数と列には対応する型が使用されます。例えば、通常、int C 変数は INT SQL 列にバインドします。ただし、列のデータ型とバインドする変数のデータ型は必ずしも同一である必要はありません。例えば、C int 変数を SQL FLOAT にバインドすることはできますが、データをやり取りする間に精度が失われる (あるいはオーバーフローやアンダーフローが発生する) リスクが生じます。

SaCursorCol* 関数は、以下の表にリストされた SQL データ型をサポートします。

表 76. サポートしている SQL データ型

SaCursolCol* 関数	TINYINT	SMALLINT	INTEGER	REAL	FLOAT	DOUBLE	DECIMAL	NUMERIC	CHAR	VARCHAR	LONGVARCHAR	WCHAR	WVARCHAR	LONGWVARC	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
SaCursorColInt	X	X	X	X	X	X	X	X	X	X	X	X	X	X						
SaCursor ColLong	X	X	X	X	X	X	X	X	X	X	X	X	X	X						
SaCursor ColFloat	X	X	X	X	X	X	X	X	X	X	X	X	X	X						
SaCursor ColDouble	X	X	X	X	X	X	X	X	X	X	X	X	X	X						
SaCursorColStr									X	X	X									
SaCursorCol Date									X	X	X	X	X	X				X		X
SaCursor ColTime									X	X	X	X	X	X					X	X
SaCursor ColTimestamp									X	X	X	X	X	X					X	X
SaCursor ColData															X	X	X			
SaCursor ColDynData		X	X	X	X	X	X	X	X	X	X				X	X	X			
SaCursor ColFixStr		X	X	X	X	X	X	X	X	X	X				X	X	X	X	X	X
SaCursor ColDynStr		X	X	X	X	X	X	X	X	X	X	X	X							

注: 他の API と同様に、solidDB SA における変換によっては、成功するかどうかが宣言した値で決まる場合があることに注意してください。例えば、フィールドの実際の値が (「123」のような) 整数の場合にのみ、SaCursorCollInt は (「foo」のような) SQL データ型 CHAR を扱うことができます。

solidDB SA 関数リファレンス

このトピックには、solidDB SA 関数のアルファベット順のリストが含まれています。

各説明には、目的、構文、パラメーター、戻り値、およびコメントが含まれています。

関数の構文

関数の宣言の構文は以下のとおりです。

```
SA_EXPORT_H function(modifier parameter [,...]);
```

ここで modifier には以下のいずれかを指定できます。

```
SaConnectT*
SaColSearchT*
SaCursorT*
SaDataTypeT*
SaDateT*
SaDfloatT*
SaDynDataT*
SaDynStrT*
SaChSetT
char*
char**
double*
long*
float*
int
int*
unsigned*
void
```

パラメーターはイタリックで記載し、以下で説明します。

パラメーターの説明

各関数の説明では、パラメーターを表形式で示します。表にはパラメーターの一般的な使用タイプ (次のセクションで説明) と、特定の関数におけるパラメーター変数の使用方法が記載されています。

パラメーター使用タイプ

以下の表には、solidDB SA パラメーターで考えられる使用タイプが示されています。パラメーターをポインターとして使用する場合は、呼び出し後にパラメーター変数の所有権を指定する 2 番目のカテゴリーの使用法を含むという点に注意してください。

表 77. solidDB SA パラメーター使用タイプ

使用タイプ	意味
in	パラメーターが入力であることを示します。
output	パラメーターが出力であることを示します。
in out	パラメーターが入出力であることを示します。
take	ポインター・パラメーターにのみ適用されます。関数がパラメーター値を取ることを意味します。関数呼び出し後は、呼び出し元は、パラメーターを参照できません。関数または関数で作成されたオブジェクトが、パラメーターが不要になったときにそれを解放する必要があります。
hold	ポインター・パラメーターにのみ適用されます。関数呼び出し後も、関数がパラメーター値を保持することを意味します。呼び出し元は、関数呼び出し後もパラメーター値を参照できます。また、パラメーターを解放する必要があります。通常は、この種のパラメーターは、ローカル・データ構造内にポインター値を保持するオブジェクトのコンストラクターに渡されます。呼び出し元はパラメーターを保持するオブジェクトが削除されるまで、パラメーターを解放することはできません。
use	ポインター・パラメーターにのみ適用されます。パラメーターが関数呼び出し中にのみ使用されることを意味します。関数呼び出し後は、呼び出し元はパラメーターを自由に処理することができます。これはパラメーター引き渡しの最も一般的なタイプです。
ref	out パラメーターにのみ適用されます。詳しくは、以下の『戻り値』を参照してください。
give	out パラメーターにのみ適用されます。詳しくは、以下の『戻り値』を参照してください。

戻り値

各関数の説明では、その関数が値を返すのかどうかと、返される値の型を示します。戻り値は以下のいずれかの値になります。

- Boolean (TRUE、FALSE)
- int (1、0 など)
- SA_RC_SUCC などの SaRetT (エラー戻りコード)。有効なエラー・コードのリストについては、 142 ページの『データベース・エラーの処理』を参照してください。
- Pointer (out パラメーター)

ポインターでは、以下の戻り使用タイプが考えられます。

表 78. ポインターの戻り使用タイプ

使用タイプ	意味
ref	呼び出し元が戻り値の参照しか行えず、解放はできないことを示します。戻り値を返したオブジェクトが、その戻り値を解放した後は、戻り値を使用しないようにしてください。
give	関数が呼び出し元に戻り値を提供することを示します。呼び出し元が戻り値を解放する必要があります。

SaArrayFlush

SaArrayFlush は、SaArrayInsert への一連の呼び出しによって配列操作バッファーが満杯になった場合、このバッファーをフラッシュ (つまり、サーバーにデータを送信) します。

デフォルトでは、SaArrayFlush 操作を含むすべての SA 操作は、自動コミット・モードで実行されます。自動コミット・モードでは、SaArrayFlush 関数は配列のすべてのレコードを単一トランザクションで自動的には挿入しません。その代わりに、SaArrayFlush を呼び出すと、各レコードの挿入が個別のトランザクションとして扱われます。パフォーマンスを最大化するには、SaArrayFlush の呼び出し前に明示的に SaTransBegin を実行し、SaArrayFlush の呼び出し後に明示的に SaTransCommit を実行します。

SaArray* 関数はリンク・ライブラリー・アクセスではサポートされておらず、ネットワーク・クライアント・ライブラリーでのみ機能します。リンク・ライブラリー・アクセスでは、SA_ERR_CALLNOSUP を返します。

構文

```
SaRetT SA_EXPORT_H SaArrayFlush(SaConnectT* scon, SaRetT* rctab)
```

SaArrayFlush 関数は以下のパラメーターを受け入れます。

表 79. SaArrayFlush のパラメーター

パラメーター	使用タイプ	説明
scon	use	接続オブジェクトを指すポインター
rctab	use	各配列操作に対する戻りコードの配列 このパラメーターが NULL 以外の場合、各配列操作の戻りコードは rctab[i] で返されます。ここで *i* は、前回の SaArrayFlush 以降の配列操作のオーダー番号です。

戻り値

SA_RC_SUCC または最初に失敗した配列操作のエラー・コード

『SaArrayInsert』も参照してください。

SaArrayInsert

SaArrayInsert は 1 つのネットワーク・メッセージに値の配列を挿入します。この関数は、挿入された値を、配列挿入バッファー内に配置します。SaArrayFlush 関数を使用して、バッファーをフラッシュ (つまりサーバーにデータを送信) することができます。

SaArrayInsert は、内部キャッシュが満杯になると暗黙的なフラッシュも行います。ただし、すべての行を確実にサーバーに送信するには、SaArrayInsert を使用して最後のレコードを挿入した後、SaArrayFlush を呼び出す必要があります。

注:

1. デフォルトでは、SaArrayInsert および SaArrayFlush の操作を含むすべての SA 操作は、自動コミット・モードで実行されます。パフォーマンスに関する重要な注記については、 148 ページの『SaArrayFlush』を参照してください。
2. SaArray* 関数はリンク・ライブラリー・アクセスではサポートされておらず、ネットワーク・クライアント・ライブラリーでのみ機能します。リンク・ライブラリー・アクセスでは、SA_ERR_CALLNOSUP を返します。

構文

```
SaRetT SA_EXPORT_H SaArrayInsert(SaCursorT* scur)
```

SaArrayInsert 関数は以下のパラメーターを受け入れます。

表 80. SaArrayInsert のパラメーター

パラメーター	使用タイプ	説明
scur	use	カーソル・オブジェクトを指すポインター

戻り値

SA_RC_SUCC またはエラー・コード

関連項目

148 ページの『SaArrayFlush』

SaColSearchCreate

SaColSearchCreate は、指定された表の列情報検索を開始します。

構文

```
SaColSearchT* SA_EXPORT_H SaColSearchCreate(
    SaConnectT* scon,
    char* tablename)
```

SaColSearchCreate 関数は以下のパラメーターを受け入れます。

表 81. SaColSearchCreate のパラメーター

パラメーター	使用タイプ	説明
scon	in	接続オブジェクトを指すポインター
tablename	in	表名

戻り値

列検索オブジェクトを指すポインター、または表が存在しない場合は NULL

SaColSearchFree

SaColSearchFree は、列検索オブジェクトを解放します。

構文

```
void SA_EXPORT_H SaColSearchFree(SaColSearchT* colsearch)
```

SaColSearchCreate 関数は以下のパラメーターを受け入れます。

表 82. SaColSearchCreate のパラメーター

パラメーター	使用タイプ	説明
colsearch	in、take	列検索ポインター

戻り値

なし

SaColSearchNext

SaColSearchNext は、表内の次の列の情報を返します。

構文

```
int SA_EXPORT_H SaColSearchNext(
    SaColSearchT* colsearch,
    char** p_colname,
    SaDataTypeT* p_coltype)
```

SaColSearchNext 関数は以下のパラメーターを受け入れます。

表 83. SaColSearchNext のパラメーター

パラメーター	使用タイプ	説明
colsearch	in、use	列検索ポインター
p_colname	out、ref	列名のローカル・コピーを指すポインターは * p_colname に格納されます。

表 83. SaColSearchNext のパラメーター (続き)

パラメーター	使用タイプ	説明
p_coltype	out	列の型は * p_coltype に格納されます。SaDataTypeT データ型および、この型で保持できる有効な値の説明は、sa.h ファイルを参照してください。

戻り値

表 84. SaColSearchNext の戻り値

値	説明
1	次の列が検出されたため、パラメーターは更新されます。
0	列がこれ以上ないため、パラメーターは更新されません。入力パラメーターが無効な場合も、関数は 0 を返します。

SaConnect

SaConnect は、solidDB サーバーへの接続を作成します。複数の接続を同時にアクティブにすることはできますが、異なる接続での操作は、個別のトランザクションで実行されます。

構文

```
SaConnectT* SA_EXPORT_H SaConnect(
    char* servername,
    char* username,
    char* password)
```

SaConnect 関数は以下のパラメーターを受け入れます。

表 85. SaConnect のパラメーター

パラメーター	使用タイプ	説明
servername	in、use	サーバー名。サーバー名が空の場合は、リンク・サーバーに接続されます。
username	in、use	ユーザー名
password	in、use	パスワード

戻り値

表 86. SaConnect の戻り値

戻り使用タイプ	説明
give	接続ポインター、または接続が失敗した場合は NULL

SaCursorAscending

SaCursorAscending は、列に昇順基準を指定します。

複数の列でソートするには、列ごとにこの関数を 1 回呼び出す必要があります。列にキー (主キーや索引) がない場合、行はサーバー側ではなく、ローカルに (クライアント上で) ソートされます。

構文

```
SaRetT SA_EXPORT_H SaCursorAscending(
   SaCursorT* scur,
   char* colname)
```

SaCursorAscending 関数は以下のパラメーターを受け入れます。

表 87. SaCursorAscending のパラメーター

パラメーター	使用タイプ	説明
scur	in、use	カーソル・オブジェクトを指すポインター
colname	in、use	列名

戻り値

SA_RC_SUCC またはエラー・コード

SaCursorAtleast

SaCursorAtleast は、列に Atleast 基準を指定します。Atleast 基準とは、列の値が Atleast の値以上でなければならないことを意味します。Atleast の値は、列に現在バインドされているユーザー変数から取得します。

構文

```
SaRetT SA_EXPORT_H SaCursorAtleast(
    SaCursorT* scur,
    char* colname)
```

SaCursorAtleast 関数は以下のパラメーターを受け入れます。

表 88. SaCursorAtleast のパラメーター

パラメーター	使用タイプ	説明
scur	in、use	カーソル・オブジェクトを指すポインター
colname	in、use	列名

戻り値

SA_RC_SUCC またはエラー・コード

SaCursorAtmost

SaCursorAtmost は、列に Atmost 基準を指定します。Atmost 基準とは、列の値が Atmost の値以下でなければならないことを意味します。Atmost の値は、列に現在バインドされているユーザー変数から取得します。

構文

```
SaRetT SA_EXPORT_H SaCursorAtmost(
    SaCursorT* scur,
    char* colname)
```

SaCursorAtmost 関数は以下のパラメーターを受け入れます。

表 89. SaCursorAtmost のパラメーター

パラメーター	使用タイプ	説明
scur	in、use	カーソル・オブジェクトを指すポインター
colname	in、use	列名

戻り値

SA_RC_SUCC またはエラー・コード

SaCursorBegin

SaCursorBegin は、セットの先頭にカーソルを位置付けます。以降の SaCursorNext 関数の呼び出しでは、最初の行が返されます。

構文

```
SaRetT SA_EXPORT_H SaCursorBegin(
    SaCursorT* scur)
```

SaCursorBegin 関数は以下のパラメーターを受け入れます。

表 90. SaCursorBegin のパラメーター

パラメーター	使用タイプ	説明
scur	use	カーソル・オブジェクトを指すポインター

戻り値

SA_RC_SUCC またはエラー・コード

SaCursorClearConstr

SaCursorClearConstr は、カーソルからすべての検索制約を消去します。

構文

```
SaRetT SA_EXPORT_H SaCursorClearConstr(
    SaCursorT* scur)
```

SaCursorClearConstr 関数は以下のパラメーターを受け入れます。

表 91. SaCursorClearConstr のパラメーター

パラメーター	使用タイプ	説明
scur	in、use	カーソル・オブジェクトを指すポインター

戻り値

SA_RC_SUCC またはエラー・コード

SaCursorColData

SaCursorColData は、ユーザー変数をデータベース列にバインドします。

バインド済みの変数は、「入力」または「出力」のいずれかのパラメーターとして使用できます。「入力」パラメーターは、挿入および更新などの操作や検索制約用に、クライアントからサーバーにデータ値を受け渡します。「出力」パラメーターは、検索中にサーバーが読み取った値を保持します。例えば、データの INSERT を実行するには、ユーザーは最初に変数をバインドし、実際の INSERT の前に値をバインド済みの変数に格納します。INSERT を実行すると、これらの値はデータベースにコピーされます。同様に、フェッチ操作中に、次の行をリトリーブすると、その行の列の値がバインド済みの変数にコピーされ、クライアント・プログラムで認識できるようにします。

1 回のバインディング後に、変数を複数回使用することができます。例えば、複数の行を挿入する場合は、バインド済みの変数に適切な値を格納し、INSERT 操作を実行するためのループを作成することができます。「バインド」操作は、ループ前に 1 回だけ行う必要があります。ループ内で、各 INSERT 操作に対して実行する

必要はありません。同様に、変数を一度バインドした後は、SaCursorNext 関数を使用して多数の行を (一度に 1 行ずつ) リトリーブすることができます。行をリトリーブするたびに、その値がバインド済みの変数にコピーされます。データ・バッファーのアドレスは、変更されない点に注意してください。SaCursorNext を呼び出すたびに、このバッファーに格納された値だけが変更されます。

列が (SELECT ステートメントで WHERE 節を使用する場合のように) 検索制約として設定されると、dataptr としてアドレスが渡されるユーザー・データ変数が指す値に、この制約の値が設定されます。例えば、関数 SaCursorEquals が列に対して呼び出された場合、サーバーはバインド済みの変数の現行値に完全に一致する値を持つ行だけをリトリーブします。検索制約は検索操作 (SaCursorSearch と、それに続く SaCursorNext の呼び出し) のためにセットアップされますが、実際には他の操作 (SaCursorUpdate または SaCursorDelete など) のためにカーソルを正しい位置に設定するために使用することができることに注意してください。通常、更新を検索と組み合わせて、行の一部だけを更新します。つまり、検索制約のある列の値を使用して影響のある行を定義し (実質的には SQL の「WHERE」節)、他のバインド済みの変数を使用して残りの列の新しい値を定義します。同じバインド済みの変数を、検索制約と更新/挿入操作の両方に使用できる点に注意してください (SQL UPDATE ステートメントの WHERE 節と「UPDATE ... SET col = value」節の両方で同じ列が使用できるのと同様です)。同じバインド済みの変数を検索制約と、クライアント/サーバー間でのデータの交換の両方に使用する場合は、バインド済みの変数内のデータが更新されるたびに検索制約が変更されることはありません。サーバーは検索制約が作成されたとき (例えば SaCursortAtmost() などの関数が呼び出されたとき) にバインド済みの変数内にあった値を使用します。

検索操作では、ユーザー変数は、現在行の値を含むように更新されます。また、検索基準が関係している場合、この関数はそれらの値を受け渡すために使用されます。挿入および更新の操作では、その列の新しい値はユーザー変数から取得します。

バインド済みの変数を「in」パラメーターとして (例えば INSERT や UPDATE の操作で) 使用する場合は、ユーザーはバッファーの割り振りと解放を行わなければなりません。バインド済みの変数を「out」パラメーターとして使用する場合は、SA レイヤーがバッファーの割り振りと解放を行います。変数を「out」パラメーターとして使用する場合は、ユーザー変数に格納される値は、列データのローカル・コピーを含むバッファーを指すポインターになります。各行をリトリーブした場合、その行の値がこのバッファーにコピーされます。このバッファーを指すポインターは、次回 SaCursorOpen または SaCursorFree を呼び出すまで有効です。その後はポインターを参照してはなりません。

構文

```
SaRetT SA_EXPORT_H SaCursorColData(
    SaCursorT* scur,
    char* colname,
    char** dataptr,
    unsigned* lenptr)
```

SaCursorColData 関数は以下のパラメーターを受け入れます。

表 92. SaCursorColData のパラメーター

パラメーター	使用タイプ	説明
scur	in、use	カーソル・オブジェクトを指すポインター
colname	in、use	列名
dataptr	in、hold	ユーザー変数を指すポインター
lenptr	in、hold	データの長さを保持するために使用する変数を指すポインター

戻り値

SA_RC_SUCC またはエラー・コード

SaCursorColDate

SaCursorColDate は、SaDateT 型のユーザー変数をデータベース列にバインドします。

検索操作では、ユーザー変数は、現在行の値を含むように更新されます。また、検索基準が関係している場合、この関数はそれらの値を受け渡すために使用されます。挿入および更新の操作では、その列の新しい値はユーザー変数から取得します。

構文

```
SaRetT SA_EXPORT_H SaCursorColDate(
SaCursorT* scur,
char* colname,
SaDateT* dateptr)
```

SaCursorColDate 関数は以下のパラメーターを受け入れます。

表 93. SaCursorColDate のパラメーター

パラメーター	使用タイプ	説明
scur	in、use	カーソル・オブジェクトを指すポインター
colname	in、use	列名
dateptr	in、hold	ユーザー変数を指すポインター

戻り値

SA_RC_SUCC またはエラー・コード

参照

変数のバインドについて詳しくは、 154 ページの『SaCursorColData』 を参照してください。

SaCursorColDateFormat

SaCursorColDateFormat は、日付形式のストリングをデータベース列にバインドします。

検索操作では、ユーザー変数は、現在行の値を含むように更新されます。また、検索基準が関係している場合、この関数はそれらの値を受け渡すために使用されます。列のデータ型によって、形式ストリングは日付、時刻、タイム・スタンプのいずれかの形式になります。

構文

```
SaRetT SA_EXPORT_H SaCursorColDateFormat(
    SaCursorT* scur,
    char* colname,
    char* dtformat)
```

SaCursorColDateFormat 関数は以下のパラメーターを受け入れます。

表 94. SaCursorColDateFormat のパラメーター

パラメーター	使用タイプ	説明
scur	in、use	カーソル・オブジェクトを指すポインター
colname	in、use	列名
dtformat	in、hold	日付/時刻/タイム・スタンプの形式ストリング

戻り値

SA_RC_SUCC またはエラー・コード

関連項目

変数のバインドについて詳しくは、 159 ページの『SaCursorColDynData』を参照してください。指定可能な日付/時刻/タイム・スタンプ形式についての説明は、 181 ページの『SaDateSetAsciiz』を参照してください。

SaCursorColDfloat

SaCursorColDfloat は、SaDfloatT 型のユーザー変数をデータベース列にバインドします。

検索操作では、ユーザー変数は、現在行の値を含むように更新されます。また、検索基準が関係している場合、この関数はそれらの値を受け渡すために使用されます。挿入および更新の操作では、その列の新しい値はユーザー変数から取得します。

注: SaDFloatT は SQL データ型の DECIMAL に相当します (FLOAT ではありません)。

構文

```
SaRetT SA_EXPORT_H SaCursorColDfloat(
    SaCursorT* scur,
    char* colname,
    SaDfloatT* dfloatptr)
```

SaCursorColDfloat 関数は以下のパラメーターを受け入れます。

表 95. SaCursorColDfloat のパラメーター

パラメーター	使用タイプ	説明
scur	in、use	カーソル・オブジェクトを指すポインター
colname	in、use	列名
dtfloatptr	in、hold	ユーザー変数を指すポインター

戻り値

SA_RC_SUCC またはエラー・コード

関連項目

『SaCursorColDouble』

161 ページの『SaCursorColFloat』

変数のバインドについて詳しくは、154 ページの『SaCursorColData』 を参照してください。

SaCursorColDouble

SaCursorColDouble は、double 型のユーザー変数をデータベース列にバインドします。

検索操作では、ユーザー変数は、現在行の値を含むように更新されます。また、検索基準が関係している場合、この関数はそれらの値を受け渡すために使用されます。挿入および更新の操作では、その列の新しい値はユーザー変数から取得します。

注: C 言語のデータ型である「double」は、SQL データ型の「FLOAT」に相当します。

構文

```
SaRetT SA_EXPORT_H SaCursorColDouble(
    SaCursorT* scur,
    char* colname,
    double* doubleptr)
```

SaCursorColDouble 関数は以下のパラメーターを受け入れます。

表 96. SaCursorColDouble のパラメーター

パラメーター	使用タイプ	説明
scur	in、use	カーソル・オブジェクトを指すポインター
colname	in、use	列名
doubleptr	in、hold	ユーザー変数を指すポインター

戻り値

SA_RC_SUCC またはエラー・コード

関連項目

161 ページの『SaCursorColFloat』.

157 ページの『SaCursorColDfloat』

変数のバインドについて詳しくは、『SaCursorColDynData』 を参照してください。

SaCursorColDynData

SaCursorColDynData は、SaDynDataT 型のユーザー変数をデータベース列にバインドします。

検索操作では、ユーザー変数は、現在行の値を含むように更新されます。また、検索基準が関係している場合、この関数はそれらの値を受け渡すために使用されます。挿入および更新の操作では、その列の新しい値はユーザー変数から取得します。

検索操作では、SaDynDataMove 関数を使用して列データが SaDynDataT 変数に格納され、これによって古いデータが上書きされます。検索の終了後、ユーザーは SaDynDataFree 関数を使用して、SaDynDataT 変数を解放する必要があります。

動的データ・オブジェクト (SaDynDataT) は、可変長データの処理を簡単にする抽象オブジェクトです。動的データはあらゆるデータ型で使用できますが、可変長データ (VARBINARY、LONG VARBINARY、VARCHAR、LONG VARCHAR など) に最適です。

データ・オブジェクトのメモリー管理は、外部からは認識されずにオブジェクト内で行われます。動的データ・オブジェクトには、データおよび長さという、外部か

ら見ることができる属性が 2 つあります。一般に、SaDynDataMove 関数および SaDynDataAppend 関数は、動的データ・オブジェクト内のデータ値の設定および変更に使用されます。必要に応じて追加のメモリーが自動的に割り振られ、SaDynDataFree を使用して動的データ・オブジェクトが破棄されると、関連付けられたすべてのメモリーが自動的に割り振り解除されます。ユーザーは SaDynDataGetData 関数および SaDynDataGetLen 関数を使用して、それぞれデータまたは長さにアクセスすることができます。

SaDynDataMove および SaDynDataAppend は、既にデータがメモリー・バッファー内に完全に存在している場合には使用できません。同じデータのコピーを 2 つ保持することによってメモリー使用量が増大するとともに、バッファーが大容量の場合、メモリー・コピーのオーバーヘッドが大きくなる可能性があります。したがって、 (SaDynDataMove を使用してコピーするよりも) SaDynDataMoveRef を使用してデータ・ポインターを直接割り当てる方が賢明な場合があります。この場合、ユーザーは動的データ・オブジェクト自体を解放した後でのみ、メモリー・バッファーを変更または割り振り解除することができます。

構文

```
SaRetT SA_EXPORT_H SaCursorColDynData(
    SaCursorT* scur,
    char* colname,
    SaDynDataT* dd)
```

SaCursorColDynData 関数は以下のパラメーターを受け入れます。

表 97. SaCursorColDynData のパラメーター

パラメーター	使用タイプ	説明
scur	in、use	カーソル・オブジェクトを指すポインター
colname	in、use	列名
dd	in、hold	ユーザー変数を指すポインター

戻り値

SA_RC_SUCC またはエラー・コード

SaCursorColDynStr

SaCursorColDynStr は、SaDynStrT 型のユーザー変数をデータベース列にバインドします。

検索操作では、ユーザー変数は、現在行の値を含むように更新されます。また、検索基準が関係している場合、この関数はそれらの値を受け渡すために使用されます。挿入および更新の操作では、その列の新しい値はユーザー変数から取得します。

検索操作では、SaDynStrMove 関数を使用して列データが SaDynStrT 変数に格納され、これによって古いデータが上書きされます。検索の終了後、ユーザーは SaDynStrFree 関数を使用して、SaDynStrT 変数を解放する必要があります。

ユーザーは SaDynStrT 変数を (文字の列だけではなく) 不特定型の列にバインドできます。データは列型と動的ストリング型の間で双方向に変換されます。

動的ストリング・オブジェクト (SaDynStrT) は、可変長ストリングの処理を簡単にする抽象オブジェクトです。一般に、SaDynStrMove 関数および SaDynStrAppend 関数は、動的ストリング・オブジェクト内のデータ値の設定および変更に使用されます。必要に応じて追加のメモリーが自動的に割り振られ、SaDynStrFree を使用して動的データ・オブジェクトが破棄されると、関連付けられたすべてのメモリーが自動的に割り振り解除されます。

構文

```
SaRetT SA_EXPORT_H SaCursorColDynStr(
    SaCursorT* scur,
    char* colname,
    SaDynStrT* ds)
```

SaCursorColDynStr 関数は以下のパラメーターを受け入れます。

表 98. SaCursorColDynStr のパラメーター

パラメーター	使用タイプ	説明
scur	in、use	カーソル・オブジェクトを指すポインター
colname	in、use	列名
ds	in、hold	ユーザー変数を指すポインター

戻り値

SA_RC_SUCC またはエラー・コード

関連項目

変数のバインドについて詳しくは、 159 ページの『SaCursorColDynData』 を参照してください。

SaCursorColFloat

SaCursorColFloat は、float 型のユーザー変数をデータベース列にバインドします。

バインド後の変数は、列に書き込むか列から読み取る値、あるいは検索操作を制約するために使用する値を (例えば SQL 内の WHERE 節に相当する部分の一部として) 保持するために使用できます。検索操作では、ユーザー変数は、リトリーブされた現在行から読み取った値を含むように更新されます。また、検索基準が関係し

ている場合、この関数はそれらの値を受け渡すために使用できます。更新および挿入の操作では、バインド済みのユーザー変数から新しい値が取得され、データベース内の列に書き込まれます。

注: C 言語の「float」データ型は、SQL の「FLOAT」データ型ではなく、SQL の「SMALLFLOAT」データ型に相当します。

構文

```
SaRetT SA_EXPORT_H SaCursorColFloat(
    SaCursorT* scur,
    char* colname,
    float* floatptr)
```

SaCursorColFloat 関数は以下のパラメーターを受け入れます。

表 99. SaCursorColFloat のパラメーター

パラメーター	使用タイプ	説明
scur	in、use	カーソル・オブジェクトを指すポインター
colname	in、use	列名
floatptr	in、hold	ユーザー変数を指すポインター

戻り値

SA_RC_SUCC またはエラー・コード

関連項目

158 ページの『SaCursorColDouble』

157 ページの『SaCursorColDfloat』.

変数のバインドについて詳しくは、159 ページの『SaCursorColDynData』 を参照してください。

SaCursorColInt

SaCursorColInt は、int 型のユーザー変数をデータベース列にバインドします。

バインド後の変数は、列に書き込むか列から読み取る値、あるいは検索操作を制約するために使用する値を (例えば SQL 内の WHERE 節に相当する部分の一部として) 保持するために使用できます。検索操作では、ユーザー変数は、リトリーブされた現在行から読み取った値を含むように更新されます。また、検索基準が関係している場合、この関数はそれらの値を受け渡すために使用できます。更新および挿入の操作では、バインド済みのユーザー変数から新しい値が取得され、データベース内の列に書き込まれます。

注: C 言語の「int」データ型はプラットフォームに依存しますが、SQL データ型 (TINYINT、 SMALLINT、INT、および BIGINT) はプラットフォームに依存しません。適切な C 言語データ型と値を、対応する SQL データ型にマップするように注意しなければなりません。

構文

```
SaRetT SA_EXPORT_H SaCursorColInt(
SaCursorT* scur,
char* colname,
int* intptr)
```

SaCursorColInt 関数は以下のパラメーターを受け入れます。

表 100. SaCursorColInt のパラメーター

パラメーター	使用タイプ	説明
scur	in、use	カーソル・オブジェクトを指すポインター
colname	in、use	列名
intptr	in、hold	ユーザー変数を指すポインター

戻り値

SA_RC_SUCC またはエラー・コード

関連項目

変数のバインドについて詳しくは、 159 ページの『SaCursorColDynData』 を参照してください。

SaCursorColLong

SaCursorColLong は、ユーザー変数をデータベース列にバインドします。

バインド後の変数は、列に書き込むか列から読み取る値、あるいは検索操作を制約するために使用する値を (例えば SQL 内の WHERE 節に相当する部分の一部として) 保持するために使用できます。検索操作では、ユーザー変数は、リトリーブされた現在行から読み取った値を含むように更新されます。また、検索基準が関係している場合、この関数はそれらの値を受け渡すために使用できます。更新および挿入の操作では、バインド済みのユーザー変数から新しい値が取得され、データベース内の列に書き込まれます。

注: C 言語の「long」データ型はプラットフォームに依存しますが、SQL データ型 (TINYINT、SMALLINT、INT、および BIGINT) はプラットフォームに依存しません。適切な C 言語データ型と値を、対応する SQL データ型にマップするように注意しなければなりません。

構文

```
SaRetT SA_EXPORT_H SaCursorColLong(
    SaCursorT* scur,
    char* colname,
    long* longptr)
```

SaCursorColLong 関数は以下のパラメーターを受け入れます。

表 101. SaCursorColLong のパラメーター

パラメーター	使用タイプ	説明
scur	in、use	カーソル・オブジェクトを指すポインター
colname	in、use	列名
longptr	in、hold	ユーザー変数を指すポインター

戻り値

SA_RC_SUCC またはエラー・コード

関連項目

変数のバインドについて詳しくは、 159 ページの『SaCursorColDynData』 を参照してください。

SaCursorColNullFlag

SaCursorColNullFlag は、NULL 値フラグを列にバインドします。

列値が NULL の場合、* p_isnullflag の値は 1 で、それ以外の場合、値は 0 です。* p_isnullflag の値は、フェッチ操作中に自動的に更新されます。検索操作では、ユーザー変数は、現在行の値を含むように更新されます。また、検索基準が関係している場合、この関数はそれらの値を受け渡すために使用されます。挿入および更新中は、*p _isnullflag がゼロ以外の場合に NULL 値がデータベースに挿入されます。

構文

```
SaRetT SA_EXPORT_H SaCursorColNullFlag(
SaCursorT* scur,
char* colname,
int* p_isnullflag)
```

SaCursorColNullFlag 関数は以下のパラメーターを受け入れます。

表 102. SaCursorColNullFlag パラメーター

パラメーター	使用タイプ	説明
scur	in、use	カーソル・オブジェクトを指すポインター

表 102. SaCursorColNullFlag パラメーター (続き)

パラメーター	使用タイプ	説明
colname	in、use	列名
p_isnullflag	in、hold	フェッチ操作中に NULL の状況が格納され、挿入および更新の操作中に NULL の状況の取得元となる整数変数を指すポインター

戻り値

SA_RC_SUCC またはエラー・コード

関連項目

変数のバインドについて詳しくは、 159 ページの『SaCursorColDynData』 を参照してください。

SaCursorColStr

SaCursorColStr は、ユーザー変数をデータベース列にバインドします。

検索操作では、ユーザー変数は、現在行の値を含むように更新されます。また、検索基準が関係している場合、この関数はそれらの値を受け渡すために使用されます。挿入および更新の操作では、その列の新しい値はユーザー変数から取得します。

検索操作では、ユーザー変数に格納される値は、列データのローカル・コピーを指すポインターになります。データ・ポインターは、次回 SaCursorOpen または SaCursorFree を呼び出すまで有効です。その後はポインターを参照してはなりません。

構文

```
SaRetT SA_EXPORT_H SaCursorColStr(
SaCursorT* scur,
char* colname,
char** strptr)
```

SaCursorColStr 関数は以下のパラメーターを受け入れます。

表 103. SaCursorColStr のパラメーター

パラメーター	使用タイプ	説明
scur	in、use	カーソル・オブジェクトを指すポインター
colname	in、use	列名
strptr	in、hold	ユーザー変数を指すポインター

戻り値

SA_RC_SUCC またはエラー・コード

関連項目

変数のバインドについて詳しくは、 159 ページの『SaCursorColDynData』 を参照してください。

SaCursorColTime

SaCursorColTime は、SaDateT 型のユーザー変数をデータベース列にバインドします。

検索操作では、ユーザー変数は、現在行の値を含むように更新されます。また、検索基準が関係している場合、この関数はそれらの値を受け渡すために使用されます。挿入および更新の操作では、その列の新しい値はユーザー変数から取得します。

構文

```
SaRetT SA_EXPORT_H SaCursorColTime(
    SaCursorT* scur,
    char* colname,
    SaDateT* timeptr)
```

注: timeptr のデータ型は実際には SaDateT です。時間データ用の別の SaTimeT はありません。

SaCursorColTime 関数は以下のパラメーターを受け入れます。

表 104. SaCursorColTime のパラメーター

パラメーター	使用タイプ	説明
scur	in、use	カーソル・オブジェクトを指すポインター
colname	in、use	列名
timeptr	in、hold	ユーザー変数を指すポインター

戻り値

SA_RC_SUCC またはエラー・コード

関連項目

変数のバインドについて詳しくは、 159 ページの『SaCursorColDynData』 を参照してください。

SaCursorColTimestamp

SaCursorColTimestamp は、SaDateT 型のユーザー変数をデータベース列にバインドします。

検索操作では、ユーザー変数は、現在行の値を含むように更新されます。また、検索基準が関係している場合、この関数はそれらの値を受け渡すために使用されます。挿入および更新の操作では、その列の新しい値はユーザー変数から取得します。

構文

```
SaRetT SA_EXPORT_H SaCursorColTimestamp(
    SaCursorT* scur,
    char* colname,
    SaDateT* timestampptr)
```

注: timeptr のデータ型は実際には SaDateT です。タイム・スタンプ・データ用の別の SaTimestampT はありません。

SaCursorColTimestamp 関数は以下のパラメーターを受け入れます。

表 105. SaCursorColTimestamp のパラメーター

パラメーター	使用タイプ	説明
scur	in、use	カーソル・オブジェクトを指すポインター
colname	in、use	列名
timestampptr	in、hold	ユーザー変数を指すポインター

戻り値

SA_RC_SUCC またはエラー・コード

関連項目

変数のバインドについて詳しくは、 159 ページの『SaCursorColDynData』 を参照してください。

SaCursorCreate

SaCursorCreate は、表名で指定された表にカーソルを作成します。表が存在していない場合は操作が失敗します。

構文

```
SaCursorT* SA_EXPORT_H SaCursorCreate(
    SaConnectT* scon,
    char* tablename)
```

SaCursorCreate 関数は以下のパラメーターを受け入れます。

表 106. SaCursorCreate のパラメーター

パラメーター	使用タイプ	説明
scon	in、hold	接続オブジェクトを指すポインター
tablename	in、use	表名

戻り値

パラメーター *scon* の使用タイプは「hold」です。これは、作成されたカーソル・オブジェクトが、関数呼び出しが戻った後も、scon オブジェクトを参照し続けるためです。

表 107. 戻り値

戻り使用タイプ	説明
give	カーソル・オブジェクトを指すポインター、または表が存在しない場合は NULL

SaCursorDelete

SaCursorDelete は、カーソル内の現在行をデータベースから削除します。カーソルは行に位置付ける必要があります。

構文

```
SaRetT SA_EXPORT_H SaCursorDelete(SaCursorT* scur)
```

SaCursorDelete 関数は以下のパラメーターを受け入れます。

表 108. SaCursorDelete のパラメーター

パラメーター	使用タイプ	説明
scur	in、use	カーソル・オブジェクトを指すポインター

戻り値

SA_RC_SUCC またはエラー・コード

SaCursorDescending

SaCursorDescending は、列に降順ソート基準を指定します。

複数の列でソートするには、列ごとにこの関数を 1 回呼び出す必要があります。

列にキー (主キーや索引) がない場合、行はサーバー側ではなく、ローカルに (クライアント上で) ソートされます。

構文

```
SaRetT SA_EXPORT_H SaCursorDescending(
    SaCursorT* scur,
    char* colname)
```

SaCursorDescending 関数は以下のパラメーターを受け入れます。

表 109. SaCursorDescending のパラメーター

パラメーター	使用タイプ	説明
scur	in、use	カーソル・オブジェクトを指すポインター
colname	in、use	列名

戻り値

SA_RC_SUCC またはエラー・コード

SaCursorEnd

SaCursorEnd は、セットの最後にカーソルを位置付けます。それ以降 SaCursorPrev を呼び出すと、セットの最後の行にカーソルが位置付けられます。

構文

```
SaRetT SA_EXPORT_H SaCursorEnd(
    SaCursorT* scur)
```

SaCursorEnd 関数は以下のパラメーターを受け入れます。

表 110. SaCursorEnd のパラメーター

パラメーター	使用タイプ	説明
scur	use	カーソル・オブジェクトを指すポインター

戻り値

SA_RC_SUCC またはエラー・コード

SaCursorEqual

SaCursorEqual は、列に等価検索基準を指定します。

構文

```
SaRetT SA_EXPORT_H SaCursorEqual(
    SaCursorT* scur,
    char* colname)
```

SaCursorEqual 関数は以下のパラメーターを受け入れます。

表 111. SaCursorEqual のパラメーター

パラメーター	使用タイプ	説明
scur	in、use	カーソル・オブジェクトを指すポインター
colname	in、use	列名

戻り値

SA_RC_SUCC またはエラー・コード

SaCursorErrorInfo

SaCursorErrorInfo は、カーソル内の前回の操作からエラー情報を返します。

構文

```
bool SA_EXPORT_H SaCursorErrorInfo(
SaCursorT* scur,
char** errstr,
int* errcode)
```

SaCursorErrorInfo 関数は以下のパラメーターを受け入れます。

表 112. SaCursorErrorInfo のパラメーター

パラメーター	使用タイプ	説明
scur	in、use	カーソル・オブジェクトを指すポインター
errstr	out、ref	NULL 以外の場合は、エラー・ストリングのローカル・コピーを指すポインターが * errstr に格納されます。
errcode	out	NULL 以外の場合は、エラー・コードが * errcode に格納されます。

戻り値

エラーがある場合は TRUE が返され、errstr および errcode が更新されます。

エラーがない場合は FALSE が返され、errstr および errcode が更新されません。

SaCursorFree

SaCursorFree は、カーソルを解放します。この呼び出し以降、カーソル・ポインターが無効になります。

構文

```
void SA_EXPORT_H SaCursorFree(SaCursorT* scur)
```

SaCursorFree 関数は以下のパラメーターを受け入れます。

表 113. SaCursorFree のパラメーター

パラメーター	使用タイプ	説明
scur	in、take	カーソル・オブジェクトを指すポインター

戻り値

なし

SaCursorInsert

SaCursorInsert は、新しい行をデータベースに挿入します。新しい行の列の値は、列にバインドされたユーザー変数から取得されます。新しい行を挿入するには、カーソルをオープンしておく必要があります。

構文

```
SaRetT SA_EXPORT_H SaCursorInsert(SaCursorT* scur)
```

SaCursorInsert 関数は以下のパラメーターを受け入れます。

表 114. SaCursorInsert のパラメーター

パラメーター	使用タイプ	説明
scur	in、use	カーソル・オブジェクトを指すポインター

戻り値

SA_RC_SUCC またはエラー・コード

SaCursorLike

SaCursorLike は、列に類似基準を指定します。

値には SQL の「_」または「%」のようなワイルドカード文字は含むことはできません。列の値にこれらの文字が含まれていると、システムがエスケープ文字により引用符でこれらの文字を囲みます。したがって、類似値は「%」文字で終わるワイルドカード文字のない SQL の類似値と実質的には同じです。例えば、列内の「MARK」を検索するようにエンジンに指定した場合、エンジンは「MARK」、「MARK SMITH」、および「MARKETING」など、「MARK」で始まるすべての値を検索します。

類似値は、列にバインドされたユーザー変数から取得します。

構文

```
SaRetT SA_EXPORT_H SaCursorLike(
    SaCursorT* scur,
    char* colname,
    int likelen)
```

SaCursorLike 関数は以下のパラメーターを受け入れます。

表 115. SaCursorLike のパラメーター

パラメーター	使用タイプ	説明
scur	in、use	カーソル・オブジェクトを指すポインター
colname	in、use	列名
likelen	in	類似部分の長さ (ストリング終止符を除く)

戻り値

SA_RC_SUCC またはエラー・コード

SaCursorNext

SaCursorNext は、データベースから次の行をフェッチします。列にバインドされたすべてのユーザー変数が更新されます。

構文

```
SaRetT SA_EXPORT_H SaCursorNext(SaCursorT* scur)
```

SaCursorNext 関数は以下のパラメーターを受け入れます。

表 116. SaCursorNext のパラメーター

パラメーター	使用タイプ	説明
scur	in、use	カーソル・オブジェクトを指すポインター

戻り値

SA_RC_SUCC: 検出された次の行

SA_RC_END: 検索終了

SaCursorOpen

SaCursorOpen は、カーソルをオープンします。

カーソルをオープンする前に、すべての SaCursorColXXX 操作を実行しておく必要があります。カーソルをオープンすると、実行している可能性がある既存の検索が終了します。また、カーソルに指定されたすべての検索基準が消去されます。

カーソルをオープンすると、ユーザーは新しい行をカーソルに挿入するか、検索基準を指定することができます。検索を開始するには、カーソルをオープンしておく必要があります。

構文

```
SaRetT SA_EXPORT_H SaCursorOpen(SaCursorT* scur)
```

SaCursorOpen 関数は以下のパラメーターを受け入れます。

表 117. SaCursorOpen のパラメーター

パラメーター	使用タイプ	説明
scur	in、use	カーソル・オブジェクトを指すポインター

戻り値

SA_RC_SUCC またはエラー・コード

SaCursorOrderbyVector

SaCursorOrderbyVector を使用して、検索で使用する列の順序を指定します。

初期値を値のベクトルとして使用して、キー内の検索開始位置を指定します。初期値はキー内の開始点の選択だけに使用されます。その後、初期値は列の値と照合されることはありません。複数の基準を指定すると、指定された順序で処理されます。順序付けのための正しいキーが必要です。

初期値は、列にバインドされたユーザー変数から取得します。

構文

```
SaRetT SA_EXPORT_H SaCursorOrderbyVector(
    SaCursorT* scur,
    char* colname)
```

SaCursorOrderbyVector 関数は以下のパラメーターを受け入れます。

表 118. SaCursorOrderbyVector のパラメーター

パラメーター	使用タイプ	説明
scur	in、use	カーソル・オブジェクトを指すポインター
colname	in、use	列名

戻り値

SA_RC_SUCC またはエラー・コード

SaCursorOrderbyVector の例

```
/* これらの変数は、「I」と「J」の名前が付いた列にバインドされます。*/
int i, j;
/* このカーソル内の列に変数をバインドします。*/
SaCursorColStr(scur, "I", 'i);
SaCursorColStr(scur, "J", 'j);
/* 検索に使用する値を設定します。*/
i = 2;
j = 1;
/* 列の順序を指定します。*/
SaCursorOrderByVector(scur, "I");
SaCursorOrderByVector(scur, "J");
/* 一致する値がないか、カーソルを検索します。*/
SaCursorSearch(scur);
```

上記は、以下の SQL WHERE 節と同等です。

```
...WHERE (i,j) >= (2,1)
```

SaCursorPrev

SaCursorPrev は、データベースから前の行をフェッチします。列に現在バインドされているすべてのユーザー変数が更新されます。

構文

```
SaRetT SA_EXPORT_H SaCursorPrev(SaCursorT* scur)
```

SaCursorPrev 関数は以下のパラメーターを受け入れます。

表 119. SaCursorPrev のパラメーター

パラメーター	使用タイプ	説明
scur	in、use	カーソル・オブジェクトを指すポインター

戻り値

SA_RC_SUCC: 検出された前の行

SA_RC_END: 検出の冒頭 (既に先頭行にいるので、前の行はありません)

注: SA_RC_END はカーソルのどちらの端 (先頭または最後) にでも適用できます。

SaCursorReSearch

SaCursorReSearch は、古い検索基準を使用して、新しい検索を開始します。

構文

```
SaRetT SA_EXPORT_H SaCursorReSearch(SaCursorT* scur)
```

SaCursorReSearch 関数は以下のパラメーターを受け入れます。

表 120. SaCursorReSearch のパラメーター

パラメーター	使用タイプ	説明
scur	use	カーソル・オブジェクトを指すポインター

戻り値

SA_RC_SUCC、SA_RC_END、またはエラー・コード。エラー・コードのリストについては、142 ページの『データベース・エラーの処理』を参照してください。

SaCursorSearch

SaCursorSearch は、カーソル内の検索を開始します。検索開始後、ユーザーはデータベースから行をフェッチすることができます。各検索は個別のトランザクションとして実行され、検索開始後は現行ユーザーやその他のユーザーによる変更を認識しません。

構文

```
SaRetT SA_EXPORT_H SaCursorSearch(SaCursorT* scur)
```

SaCursorSearch 関数は以下のパラメーターを受け入れます。

表 121. SaCursorSearch のパラメーター

パラメーター	使用タイプ	説明
scur	in、use	カーソル・オブジェクトを指すポインター

戻り値

SA_RC_SUCC、SA_RC_END、またはエラー・コード

SaCursorSearchByRowid

SaCursorSearchByRowid は、rowid で指定された行が検索セットに含まれる新しい検索を開始します。

SaCursorSearchByRowid は rowid に従って検索を行うだけなので、1 行またはゼロ行を返します。以前の検索制約は解除されず、次回 SaCursorReSearch を呼び出す際にも有効になります。

特定のレコードの rowid を取得するには、rowid 列の値を読み取ります。各表には rowid 列があります。CREATE TABLE または ALTER TABLE ステートメント内で rowid 列を明示的に作成する必要はありません。

構文

```
SaRetT SA_EXPORT_H SaCursorSearchByRowid(
    SaCursorT* scur,
    void* rowid,
    int rowidlen)
```

SaCursorSearchByRowid 関数は以下のパラメーターを受け入れます。

表 122. SaCursorSearchByRowid のパラメーター

パラメーター	使用タイプ	説明
scur	in、use	カーソル・オブジェクトを指すポインター
rowid	in、use	rowid を含むデータ域を指すポインター。rowid は「void *」として宣言されますが、ストリング (char *) の形式でなければなりません。
rowidlen	in	rowid パラメーターが指すデータ (ストリング) の長さ

戻り値

SA_RC_SUCC、SA_RC_END、またはエラー・コード

SaCursorSearchReset

SaCursorSearchReset は、検索カーソルをリセットします。

古い検索制約が使用されますが、その値はユーザー・バッファー (すなわち、パラメーター) から再度読み取られます。使用する特定の値以外は同一の照会を使用して繰り返し検索を行う場合は、これによってパフォーマンスを向上することができます。

例えば、特定のユーザーや接続が、特定の表の ID 列に基づいて常にその表を検索し、検索のたびに異なる ID 値を使用すると仮定します。次の ID を検索するための「新しい」照会を作成する代わりに、既存の照会をリセットして、新しい値を使用することができます。

例えば、以下のような既存のコードがあると仮定します。

```
...
/* 変数を列にバインドします。*/
SaCursorColInt(scur, "MY_COL_NAME", &search_parameter1);

/* 毎回異なる値を使用して照会を繰り返します。*/
while (there_are_more_values_to_look_for)  {
    /* 検索する値のパラメーターを設定します。*/
    search_parameter1 = some_value;
    /* 検索基準を指定します。*/
    rc = SaCursorEqual(scur, "MY_COL_NAME");
    /* 検索基準とパラメーター値を使用する新しい照会を作成します。*/
    rc = SaCursorSearch(scur);
    /* 検索基準に一致する行 (複数可) を取得します。*/
    rc = SaCursorNext(scur);
    /* リトリーブしたデータを処理します... */
    foo();
    ...
```

```
    /* 次のループを反復する前に、古い照会を削除します。*/
    rc = SaCursorClearConstr(scur);
    }
...
```

以下の例のようにコードを変更することで、ほとんどの場合にパフォーマンスを向上できます。

```
...
/* 変数を列にバインドします。*/
SaCursorColInt(scur, "MY_COL_NAME", &search_parameter1);

/* 新しい照会を作成します。*/
rc = SaCursorEqual(scur, "MY_COL_NAME");
rc = SaCursorSearch(scur);
/* 検索する値のパラメーターを設定します。*/
search_parameter1 = some_value;

/* 毎回異なる値を使用して照会を繰り返します。*/
while (there_are_more_values_to_look_for)  {
    /* 検索基準に一致する行 (複数可) を取得します。*/
    rc = SaCursorNext(scur);
    /* リトリーブしたデータを処理します... */
    foo();
    ...
    /* 次に検索する値のパラメーターを設定します。*/
    search_parameter1 = some_value;
    /* 既存の照会をリセットして、パラメーター内の最新の値を使用します。*/
    rc = SaCursorSearchReset(scur);
    }
...
```

SaCursorSearchReset() を使用すると、制約条件 (上記の例では「Equal」) の再指定や、SaCursorSearch() の呼び出しを毎回行う必要がなくなります。

SaCursorSearchReset は新しい結果セットの先頭にカーソルをリセットします。例えば、制約がまったくない検索をリセットすると、カーソルの位置は表の先頭に変更されます。

注: この関数を呼び出す前に、バッファー内の検索パラメーター値を必ず更新するようにしてください。この関数呼び出し中に、新しい値が読み取られます。

制限

1. SaCursorSearchReset() は以下のシナリオでは使用できません。
   - 検索でローカル・ソートが行われます。つまり、検索で使用される索引では処理できないソート基準があります。
   - SaCursorSearchByRowid で rowid による検索が行われます。

   このようなケースでは、SaCursorSearchReset は SA_ERR_NORESETSEARCH を返します。
2. 制約内で使用する各「類似」値は同じ長さでなければなりません。これは、SaCursorLike() が「類似」制約の長さを引数として使用するが、SaCursorSearchReset() を呼び出すとこの長さを変更できないからです。例えば、以下のシーケンスの「類似」値を使用すると、すべてが同じ長さなので、関数は正しく機能します。

   ```
   "SMITH"
   "JONES"
   ```

ただし、以下のようなシーケンスの「類似」値を使用すると、この関数は正しく機能しません。

```
"SMITH"

"JOHNSON"
```

3. 同じ列のバインディングを使用して複数の制約を設定する場合は、SaCursorSearchReset を使用することは、通常、実用的ではありません。例えば、1 から 10 までの範囲 (1 と 10 を含む) で「col」の値を検索するとします。以下のような例になります。

```
SaCursorColInt(scur, "col", &i);
i = 1;
SaCursorAtleast(scur, "col");
i = 10;
SaCursorAtmost(scur, "col");
```

このような検索をリセットすると、変数 i から一度だけ列の新しい値が読み取られます。したがって、サーバーは 1 つの値を読み取り、それを上限および下限の両方として使用します。例えば、以下のようなコードを使用すると仮定します。

```
i = 5;
SaCursorSearchReset(scur);
```

このコードでは 5 <= i <= 5 という検索を行いますが、これは望ましい結果ではありません。

構文

```
SaRetT SA_EXPORT_H SaCursorSearchReset(
    SaCursorT* scur
```

SaCursorSearchReset 関数は以下のパラメーターを受け入れます。

表 123. SaCursorSearchReset のパラメーター

パラメーター	使用タイプ	説明
scur	in、use	カーソル・オブジェクトを指すポインター

戻り値

SA_RC_SUCC またはエラー・コード

SaCursorSetLockMode

SaCursorSetLockMode は、カーソルの検索モードを設定します。

この設定は、サーバーで可能なロック方式モードに影響を及ぼします。検索が既にアクティブな場合は、設定は同じカーソルで行われる次の検索でのみ有効です。デフォルトの検索モードは SA_LOCK_SHARE です。

構文

```
SaRetT SA_EXPORT_H SaCursorSetLockMode(
    SaCursorT* scur,
    sa_lockmode_t lockmode)
```

SaCursorSetLockMode 関数は以下のパラメーターを受け入れます。

表 124. SaCursorSetLockMode のパラメーター

パラメーター	使用タイプ	説明
scur	in、use	カーソル・オブジェクトを指すポインター
lockmode	in	以下のいずれかの検索モードです。 • SA_LOCK_SHARE • SA_LOCK_FORUPDATE • SA_LOCK_EXCLUSIVE

各モードの意味は以下のとおりです。

- SA_LOCK_SHARE: デフォルトのオプティミスティック並行性制御。
- SA_LOCK_FORUPDATE: 更新する行をロックします。他のユーザーは読み取りだけが行え、書き込みは行えません。
- SA_LOCK_EXCLUSIVE: 行を排他的にロックします。他のユーザーはこのレコードを読み取ることも、書き込むこともできません。

注: この関数は任意の表に適用されます。この関数を適用するために、表を特定のロック・モードにする必要はありません。

戻り値

SA_RC_SUCC

SA_ERR_ILLENUMVAL

SaCursorSetPosition

SaCursorSetPosition は、キー値で指定された行にカーソルを位置付けます。キー値は、ユーザーがバインド済みの、制約指定がある列変数から取得されます。

構文

```
SaRetT SA_EXPORT_H SaCursorSetPosition(
    SaCursorT* scur)
```

SaCursorSetPosition 関数は以下のパラメーターを受け入れます。

表 125. SaCursorSetPosition のパラメーター

パラメーター	使用タイプ	説明
scur	in、use	カーソル・オブジェクトを指すポインター

戻り値

SA_RC_SUCC またはエラー・コード

SaCursorSetRowsPerMessage

SaCursorSetRowsPerMessage は、1 つのネットワーク・メッセージでサーバーからクライアントに送信する行の数を設定します。

関数 SaCursorSearch によって検索が開始された後は、この設定は無効です。

構文

```
SaRetT SA_EXPORT_H
    SaCursorSetRowsPerMessage(
    SaCursorT* scur,
    int rows_per_message)
```

SaCursorSetRowsPerMessage 関数は以下のパラメーターを受け入れます。

表 126. SaCursorSetRowsPerMessage のパラメーター

パラメーター	使用タイプ	説明
scur	in、use	カーソル・オブジェクトを指すポインター
rows_per_message	in	1 つのネットワーク・メッセージで送信する行の数

戻り値

SA_RC_SUCC: 成功

SA_ERR_FAILED: エラー、rows_per_message < 1

SaCursorUpdate

SaCursorUpdate は、データベースのカーソルの現在行を更新します。

カーソルは行に位置付ける必要があります。新しい行の列の値は、列にバインドされたユーザー変数から取得されます。

構文

```
SaRetT SA_EXPORT_H SaCursorUpdate(SaCursorT* scur)
```

SaCursorUpdate 関数は以下のパラメーターを受け入れます。

表 127. SaCursorUpdate のパラメーター

パラメーター	使用タイプ	説明
scur	in、use	カーソル・オブジェクトを指すポインター

戻り値

SA_RC_SUCC またはエラー・コード

SaDateCreate

SaDateCreate は、新しい日付オブジェクトを作成します。

日付オブジェクトに格納される日付は未定義です。

構文

```
SaDateT* SA_EXPORT_H SaDateCreate(void)
```

SaDateCreate 関数はパラメーターを受け入れません。

戻り値

表 128. SaDateCreate の戻り値

戻り使用タイプ	説明
give	新しい日付オブジェクト

SaDateFree

SaDateFree は、日付オブジェクトを解放します。

この呼び出し後は、その日付オブジェクトが無効となり使用できなくなります。

構文

```
void SA_EXPORT_H SaDateFree(SaDateT* date)
```

SaDateFree 関数は以下のパラメーターを受け入れます。

表 129. SaDateFree のパラメーター

パラメーター	使用タイプ	説明
date	in、take	日付オブジェクト

戻り値

なし

SaDateSetAsciiz

SaDateSetAsciiz は、ASCII ゼロ・ストリング日付を日付オブジェクトに設定します。

形式ストリングでは、以下の特殊文字が認識されます。

```
YYYY        世紀を含む年
YY          デフォルトの世紀 1900 での年
MM          月
M           月
```

```
DD          日
D           日
HH          時
H           時
NN          分
N           分
SS          秒
S           秒
FFF         秒の小数部、1/1000 秒
```

すべてのフィールドはオプションです。フィールドは形式ストリングに従ってスキャンされ、一致すると、そのフィールドが正しい値で置き換えられます。フォーマット内のその他のすべての文字は、そのまま処理されます。

2 文字 (例えば、「MM」、「DD」など) の場合、値が 2 桁で表現されることを意味しています (1 から 9 の値は、前に文字 0 を付けて、例えば 01 と表現されます)。単一の文字は、可能な場合には値を 1 桁で表現することを意味しています。例えば、日付形式を ″YY-M-D″ と定義した場合、1999 年 1 月 2 日は ″99-1-2″ のようになります。日付形式を ″YY-MM-DD″ と定義した場合、この日付は ″99-01-02″ のようになります。

以下の例に、日付形式の使用法を示します。

```
SaDateSetAsciiz(date, "YY-MM-DD", "94-09-13");

SaDateSetAsciiz(date, "MM/DD/YY HH.NN", "09/13/94 19.20");
```

デフォルトの日付形式は YYYY-MM-DD HH:NN:SS であり、ここで時刻フィールドはオプションです。

構文

```
SaRetT SA_EXPORT_H SaDateSetAsciiz(
    SaDateT* date,
    char* format,
    char* asciiz)
```

SaDateSetAsciiz 関数は以下のパラメーターを受け入れます。

表 130. SaDateSetAsciiz のパラメーター

パラメーター	使用タイプ	説明
date	in、out	日付オブジェクト
format	in、use	asciiz (ゼロ終了 ASCII) バッファーの日付形式、またはデフォルトのフォーマットを使用する場合には NULL
asciiz	in、use	asciiz (ゼロ終了 ASCII) ストリング・フォーマットのデータを含むバッファー

戻り値

SA_RC_SUCC

SA_ERR_FAILED

SaDateSetTimet

SaDateSetTimet は、入力値を「timet」という名前の変数から「date」という名前の変数にコピーします。この値は、time_t フォーマット (C ライブラリー関数 time() から返されるフォーマット) から SaDateT フォーマットに自動的に変換されます。

構文

```
SaRetT SA_EXPORT_H SaDateSetTimet(
SaDateT* date,
long timet)
```

SaDateSetTimet 関数は以下のパラメーターを受け入れます。

表 131. SaDateSetTimet のパラメーター

パラメーター	使用タイプ	説明
date	use	日付オブジェクト
timet	in	time_t フォーマットの新しい日付値

戻り値

SA_RC_SUCC

SA_ERR_FAILED

SaDateToAsciiz

SaDateToAsciiz は、ASCII ゼロ終了ストリング・フォーマットで日付を格納します。

別の日付形式については、 181 ページの『SaDateSetAsciiz』を参照してください。

構文

```
SaRetT SA_EXPORT_H SaDateToAsciiz(
SaDateT* date,
char* format,
char* asciiz)
```

SaDateToAsciiz 関数は以下のパラメーターを受け入れます。

表 132. SaDateToAsciiz のパラメーター

パラメーター	使用タイプ	説明
date	in、use	日付オブジェクト
format	in、use	asciiz (ゼロ終了 ASCII) バッファーでの日付形式、またはデフォルトのフォーマットを使用する場合には NULL

表 132. *SaDateToAsciiz* のパラメーター (続き)

パラメーター	使用タイプ	説明
asciiz	out	日付を格納するバッファー **注:** 呼び出し元は、この関数を呼び出す前に十分に大きいバッファーを割り振る必要があること、またバッファーの処理の終了時にこのバッファーを割り振り解除する必要があることに注意してください。

戻り値

SA_RC_SUCC

SA_ERR_FAILED

SaDateToTimet

SaDateToTimet は、日付を time_t フォーマットで格納します。time_t の日付は、C ライブラリー関数 time() で返される値と同じです。

構文

```
SaRetT SA_EXPORT_H SaDateToTimet(
SaDateT* date,
long* p_timet)
```

SaDateToTimet 関数は以下のパラメーターを受け入れます。

表 133. *SaDateToTimet* のパラメーター

パラメーター	使用タイプ	説明
date	in、use	日付オブジェクト
timet	out	time_t フォーマットで日付を格納する long 変数を指すポインター

戻り値

SA_RC_SUCC

SA_ERR_FAILED

SaDefineChSet

SaDefineChSet は、クライアント文字セットを定義します。

構文

```
SaRetT SA_EXPORT_H SaDefineChSet(
SaConnectT* scon,
SaChSetT chset)
```

SaDefineChSet 関数は以下のパラメーターを受け入れます。

表 134. SaDefineChSet のパラメーター

パラメーター	使用タイプ	説明
scon	in out	接続オブジェクトを指すポインター
chset	in	列挙型文字指定。有効な文字セットは、sa.h ファイルにリストされており、SA_CHSET_DEFAULT、SA_CHSET_ANSI などが含まれています。

注: scon パラメーターはこの関数呼び出しで変更されるため、scon の使用タイプには「out」が含まれます。

戻り値

OK の場合には SA_RC_SUCC、または指定した文字セットがサポートされない場合には SA_ERR_CHSETUNSUPP

SaDfloatCmp

SaDfloatCmp は、2 つの dfloat 値を比較します。

構文

```
int SA_EXPORT_H SaDfloatCmp(
SaDfloatT* p_dfl1,
SaDfloatT* p_dfl2)
```

SaDfloatCmp 関数は以下のパラメーターを受け入れます。

表 135. SaDfloatCmp のパラメーター

パラメーター	使用タイプ	説明
p_dfl1	in、use	dfloat 変数を指すポインター
p_dfl2	in、use	dfloat 変数を指すポインター

戻り値

```
p_dfl1 < p_dfl2 の場合 <   -1
p_dfl1 = p_dfl2 の場合 =   0
p_dfl1 > p_dfl2 の場合 >   1
```

これは、C の strcmp() 関数と同等のものであり、この C 関数は、最初のパラメーターが 2 番目のパラメーターより小さい場合には負数、2 つが等しい場合にはゼロ、最初のパラメーターが 2 番目のパラメーターより大きい場合には正数 (ゼロより大きい) を返します。

SaDfloatDiff

SaDfloatDiff は、2 つの dfloat 値の差 (つまり、p_dfl1 - p_dfl2) を計算します。その結果は、* p_result_dfl に格納されます。

構文

```
SaRetT SA_EXPORT_H SaDfloatDiff(
    SaDfloatT* p_result_dfl,
    SaDfloatT* p_dfl1,
    SaDfloatT* p_dfl2)
```

SaDfloatDiff 関数は以下のパラメーターを受け入れます。

表 136. SaDfloatDiff のパラメーター

パラメーター	使用タイプ	説明
p_result_dfl	out	結果を格納する dfloat 変数を指すポインター
p_dfl1	in、use	dfloat 変数を指すポインター
p_dfl2	in、use	dfloat 変数を指すポインター

戻り値

SA_RC_SUCC

SA_ERR_FAILED

SaDfloatOverflow

SaDfloatOverflow は、dfloat にオーバーフロー値が含まれるか検査します。

構文

```
int SA_EXPORT_H SaDfloatOverflow(
    SaDfloatT* p_dfl)
```

SaDfloatOverflow 関数は以下のパラメーターを受け入れます。

表 137. SaDfloatOverflow のパラメーター

パラメーター	使用タイプ	説明
p_dfl	in、use	dfloat 変数を指すポインター

戻り値

1: dfloat 値がオーバーフロー値の場合

0: dfloat 値がオーバーフロー値ではない場合

SaDfloatProd

SaDfloatProd は、2 つの dfloat 値の積を計算します。その結果は、* p_result_dfl に格納されます。

構文

```
SaRetT SA_EXPORT_H SaDfloatProd(
SaDfloatT* p_result_dfl,
SaDfloatT* p_dfl1,
SaDfloatT* p_dfl2)
```

SaDfloatProd 関数は以下のパラメーターを受け入れます。

表 138. SaDfloatProd のパラメーター

パラメーター	使用タイプ	説明
p_result_dfl	out	結果を格納する dfloat 変数を指すポインター
p_dfl1	in	dfloat 変数を指すポインター
p_dfl2	in	dfloat 変数を指すポインター

戻り値

SA_RC_SUCC

SA_ERR_FAILED

SaDfloatQuot

SaDfloatQuot は、2 つの dfloat 値の商 (つまり、p_dfl1/p_dfl2) を計算します。その結果は、* p_result_dfl に格納されます。

構文

```
SaRetT SA_EXPORT_H SaDfloatQuot(
    SaDfloatT* p_result_dfl,
    SaDfloatT* p_dfl1,
    SaDfloatT* p_dfl2)
```

SaDfloatQuot 関数は以下のパラメーターを受け入れます。

表 139. SaDfloatQuot のパラメーター

パラメーター	使用タイプ	説明
p_result_dfl	out	結果を格納する dfloat 変数を指すポインター
p_dfl1	in	dfloat 変数を指すポインター
p_dfl2	in	dfloat 変数を指すポインター

戻り値

SA_RC_SUCC

SA_ERR_FAILED

SaDfloatSetAsciiz

SaDfloatSetAsciiz は、ゼロ終了 ASCII ストリングから dfloat 値を設定します。

構文

```
SaRetT SA_EXPORT_H SaDfloatSetAsciiz(
    SaDfloatT* p_dfl,
    char* asciiz)
```

SaDfloatSetAsciiz 関数は以下のパラメーターを受け入れます。

表 140. SaDfloatSetAsciiz のパラメーター

パラメーター	使用タイプ	説明
p_dfl1	out	結果を格納する dfloat 変数を指すポインター
asciiz	in	ゼロ終了 ASCII ストリングとして dfloat 値を読み取るバッファー

戻り値

SA_RC_SUCC

SA_ERR_FAILED

SaDfloatSum

SaDfloatSum は、2 つの dfloat 値の和を計算します。その結果は、* p_result_dfl に格納されます。

構文

```
SaRetT SA_EXPORT_H SaDfloatSum(
    SaDfloatT* p_result_dfl,
    SaDfloatT* p_dfl1,
    SaDfloatT* p_dfl2)
```

SaDfloatSum 関数は以下のパラメーターを受け入れます。

表 141. SaDfloatSum のパラメーター

パラメーター	使用タイプ	説明
p_result_dfl	out	結果を格納する dfloat 変数を指すポインター
p_dfl1	in	dfloat 変数を指すポインター

表 141. *SaDfloatSum* のパラメーター *(続き)*

パラメーター	使用タイプ	説明
p_dfl2	in	dfloat 変数を指すポインター

戻り値

SA_RC_SUCC またはエラー・コード

SaDfloatToAsciiz

SaDfloatToAsciiz は、dfloat 値を asciiz (ゼロ終了 ASCII) ストリングとして格納します。

構文

```
SaRetT SA_EXPORT_H SaDfloatToAsciiz(
    SaDfloatT* p_dfl,
    char* asciiz)
```

SaDfloatToAsciiz 関数は以下のパラメーターを受け入れます。

表 142. *SaDfloatToAsciiz* のパラメーター

パラメーター	使用タイプ	説明
p_dfl	in	dfloat 変数を指すポインター
asciiz	out	dfloat を asciiz (ゼロ終了 ASCII) ストリング・フォーマットで格納するバッファー。そのメモリーは、呼び出し元が事前に割り振る必要があります。

戻り値

SA_RC_SUCC

SA_ERR_FAILED

SaDfloatUnderflow

SaDfloatUnderflow は、dfloat にアンダーフロー値が含まれるか検査します。

構文

```
int SA_EXPORT_H SaDfloatUnderflow(
    SaDfloatT* p_dfl)
```

SaDfloatUnderflow 関数は以下のパラメーターを受け入れます。

表 143. *SaDfloatUnderflow* のパラメーター

パラメーター	使用タイプ	説明
p_dfl	in、use	dfloat 変数を指すポインター

戻り値

1: dfloat 値がアンダーフロー値の場合

0: dfloat 値がアンダーフロー値ではない場合

SaDisconnect

SaDisconnect は、solidDB サーバーからユーザーを切断します。

構文

```
void SA_EXPORT_H SaDisconnect(SaConnectT* scon)
```

SaDisconnect 関数は以下のパラメーターを受け入れます。

表 144. SaDisconnect のパラメーター

パラメーター	使用タイプ	説明
scon	in、take	接続オブジェクトを指すポインター

戻り値

なし

SaDynDataAppend

SaDynDataAppend は、動的データ・オブジェクトにデータを付加します。

構文

```
void SA_EXPORT_H SaDynDataAppend(
    SaDynDataT* dd,
    char* data,
    unsigned len)
```

SaDynDataAppend 関数は以下のパラメーターを受け入れます。

表 145. SaDynDataAppend のパラメーター

パラメーター	使用タイプ	説明
dd	use	動的データ・オブジェクト
data	in out、use	dd に付加するデータ
len	in	付加するデータの長さ

戻り値

なし

関連項目

変数のバインドについて詳しくは、 159 ページの『SaCursorColDynData』を参照してください。

SaDynDataChLen

SaDynDataChLen は、動的データ・オブジェクトのデータ域の長さを変更します。必要に応じて、メモリーの割り振りと割り振り解除を行います。

新しい長さが現在の長さよりも小さい場合、データ域が切り捨てられます。新しい長さが現在の長さよりも大きい場合、新しいデータ域の内容はスペース文字で初期化されます。

構文

```
void SA_EXPORT_H SaDynDataChLen(
    SaDynDataT* dd,
    unsigned len)
```

SaDynDataChLen 関数は以下のパラメーターを受け入れます。

表 146. SaDynDataChLen のパラメーター

パラメーター	使用タイプ	説明
dd	in out、use	動的データ・オブジェクト
len	in	動的データ・オブジェクトの新しいデータ域の長さ

戻り値

なし

関連項目

「動的データ」(SaDynDataT) について詳しくは、 159 ページの『SaCursorColDynData』を参照してください。

SaDynDataClear

SaDynDataClear は、SaDynDataT オブジェクトからメモリーの割り振りを解除します。

SaDynDataClear は、データを割り振り解除しますが、SaDynDataT オブジェクト自体は割り振り解除しません。SaDynDataClear の結果として、SaDynDataCreate から返された「空」の動的データ・オブジェクトが残されます。SaDynDataT オブジェクト自体は、SaDynDataFree 関数を使用して別途割り振り解除する必要があります。

構文

```
void SA_EXPORT_H SaDynDataClear(
    SaDynDataT* dd)
```

SaDynDataClear 関数は以下のパラメーターを受け入れます。

表 147. SaDynDataClear のパラメーター

パラメーター	使用タイプ	説明
dd	in out、use	動的データ・オブジェクト

戻り値

なし

関連項目

「動的データ」(SaDynDataT) について詳しくは、 159 ページの
『SaCursorColDynData』を参照してください。

SaDynDataCreate

SaDynDataCreate は、新しい動的データ・オブジェクトを作成します。動的データ・オブジェクトは、可変量の不特定型データを保持できるオブジェクトです。

動的データ・オブジェクトは、他の SaDynDataXXX 関数を使用して操作できます。

構文

```
SaDynDataT* SA_EXPORT_H SaDynDataCreate(void)
```

SaDynDataCreate はパラメーターを受け入れません。

戻り値

表 148. SaDynDataCreate の戻り値

戻り使用タイプ	説明
give	新しい空の動的データ・オブジェクト。エラーの場合には、NULL を返します。

関連項目

「動的データ」(SaDynDataT) について詳しくは、 159 ページの
『SaCursorColDynData』を参照してください。

SaDynDataFree

SaDynDataFree は、動的データ・オブジェクトを解放します。この呼び出し後は、動的データ・オブジェクト・ポインターが無効になり使用できなくなります。

構文

```
void SA_EXPORT_H SaDynDataFree(
    SaDynDataT* dd)
```

SaDynDataFree 関数は以下のパラメーターを受け入れます。

表 149. SaDynDataFree のパラメーター

パラメーター	使用タイプ	説明
dd	in、take	動的データ・オブジェクト

戻り値

なし

関連項目

「動的データ」(SaDynDataT) について詳しくは、 159 ページの
『SaCursorColDynData』を参照してください。

SaDynDataGetData

SaDynDataGetData は、動的データ・オブジェクトのデータ域を指すポインターを返します。

構文

```
char* SA_EXPORT_H SaDynDataGetData(
    SaDynDataT* dd)
```

SaDynDataGetData 関数は以下のパラメーターを受け入れます。

表 150. SaDynDataGetData のパラメーター

パラメーター	使用タイプ	説明
dd	in、use	動的データ・オブジェクト

戻り値

動的データ・オブジェクトのローカル・データ域への参照

関連項目

「動的データ」(SaDynDataT) について詳しくは、 159 ページの
『SaCursorColDynData』を参照してください。

SaDynDataGetLen

SaDynDataGetLen は、動的データ・オブジェクトのデータ域の長さを返します。

構文

```
unsigned SA_EXPORT_H SaDynDataGetLen(
    SaDynDataT* dd)
```

SaDynDataGetData 関数は以下のパラメーターを受け入れます。

表 151. SaDynDataGetData のパラメーター

パラメーター	使用タイプ	説明
dd	in、use	動的データ・オブジェクト

戻り値

データ域の長さ。この関数は、エラーが発生した場合、またはデータ域の実際の長さが 0 の場合、0 を返します。

関連項目

「動的データ」(SaDynDataT) について詳しくは、 159 ページの『SaCursorColDynData』を参照してください。

SaDynDataMove

SaDynDataMove は、「data」という名前のパラメーターから動的データ・オブジェクト (名前は dd) にデータをコピーします。この関数は、既存のデータが存在する場合には上書きします。

パラメーター dd は、SaDynDataCreate 関数で事前に作成した動的データ・オブジェクトを指している必要があります。

注: SaDynDataMove は、データをコピーします。データではなく、単なる参照をコピーする場合には、 195 ページの『SaDynDataMoveRef』を参照してください。

一般に、SaDynDataMove 関数および SaDynDataAppend 関数は、動的データ・オブジェクト内のデータ値の設定および変更に使用されます。必要に応じて追加のメモリーが自動的に割り振られ、SaDynDataFree を使用して動的データ・オブジェクトが破棄されると、関連付けられたすべてのメモリーが自動的に割り振り解除されます。ユーザーは SaDynDataGetData 関数および SaDynDataGetLen 関数を使用して、それぞれデータまたは長さにアクセスすることができます。

SaDynDataMove および SaDynDataAppend は、既にデータがメモリー・バッファー内に完全に存在している場合には使用できません。同じデータのコピーを 2 つ保持することによってメモリー使用量が増大するとともに、バッファーが大容量の場合、メモリー・コピーのオーバーヘッドが大きくなる可能性があります。したがって、(SaDynDataMove を使用してコピーするよりも) SaDynDataMoveRef を使用してデータ・ポインターを直接割り当てる方が賢明な場合があります。この場合、ユーザーは動的データ・オブジェクト自体を解放した後でのみ、メモリー・バッファーを変更または割り振り解除することができます。

構文

```
void SA_EXPORT_H SaDynDataMove(
SaDynDataT* dd,
char* data,
unsigned len)
```

SaDynDataMove 関数は以下のパラメーターを受け入れます。

表 152. SaDynDataMove のパラメーター

パラメーター	使用タイプ	説明
dd	in out、use	動的データ・オブジェクト
data	in、use	新しいデータ
len	in	データの長さ (データがストリングの場合、この長さにはストリング終止符が含まれる)

戻り値

なし

関連項目

『SaDynDataMoveRef』

「動的データ」(SaDynDataT) について詳しくは、 159 ページの『SaCursorColDynData』を参照してください。

SaDynDataMoveRef

SaDynDataMoveRef は、動的データ・オブジェクトに対するデータ参照を移動します。

SaDynDataMoveRef は、「data」という名前のパラメーターから「dd」という名前のパラメーターの該当フィールドにポインター (アドレス) をコピーします。呼び出し元は、動的データ・オブジェクトが入力データを参照している限り、そのデータが存続することを保証する必要があります。

注: この関数は、データではなく、参照のみをコピーします。単なる参照ではなく、データをコピーする場合には、 194 ページの『SaDynDataMove』を参照してください。

一般に、SaDynDataMove 関数および SaDynDataAppend 関数は、動的データ・オブジェクト内のデータ値の設定および変更に使用されます。必要に応じて追加のメモリーが自動的に割り振られ、SaDynDataFree を使用して動的データ・オブジェクトが破棄されると、関連付けられたすべてのメモリーが自動的に割り振り解除されます。ユーザーは SaDynDataGetData 関数および SaDynDataGetLen 関数を使用して、それぞれデータまたは長さにアクセスすることができます。

SaDynDataMove および SaDynDataAppend は、既にデータがメモリー・バッファー内に完全に存在している場合には使用できません。同じデータのコピーを 2 つ保持することによってメモリー使用量が増大するとともに、バッファーが大容量の場合、メモリー・コピーのオーバーヘッドが大きくなる可能性があります。したがって、(SaDynDataMove を使用してコピーするよりも) SaDynDataMoveRef を使用してデータ・ポインターを直接割り当てる方が優れている場合があります。この場合、

ユーザーは動的データ・オブジェクト自体を解放した後でのみ、メモリー・バッファーを変更または割り振り解除することができます。

構文

```
void SA_EXPORT_H SaDynDataMoveRef(
    SaDynDataT* dd,
    char* data,
    unsigned len)
```

SaDynDataMoveRef 関数は以下のパラメーターを受け入れます。

表 153. SaDynDataMoveRef のパラメーター

パラメーター	使用タイプ	説明
dd	in out、use	動的データ・オブジェクト
data	in、hold	データ
len	in	データの長さ (データがストリングの場合、この長さにはストリング終止符が含まれる)

戻り値

なし

関連項目

194 ページの『SaDynDataMove』

「動的データ」(SaDynDataT) について詳しくは、 159 ページの『SaCursorColDynData』を参照してください。

SaDynStrAppend

SaDynStrAppend は、動的ストリングの終端に別のストリングを付加します。

構文

```
void SA_EXPORT_H SaDynStrAppend(
    SaDynStrT* p_ds,
    char* str)
```

SaDynStrAppend 関数は以下のパラメーターを受け入れます。

表 154. SaDynStrAppend のパラメーター

パラメーター	使用タイプ	説明
p_ds	out	動的ストリング
str	in、use	p_ds に付加されるストリング

戻り値

なし

SaDynStrCreate

SaDynStrCreate は、新しい動的ストリング・オブジェクトを作成 (初期化) します。

構文

```
SaDynStrT SA_EXPORT_H SaDynStrCreate(void)
```

SaDynStrCreate はパラメーターを受け入れません。

戻り値

表 155. SaDynStrCreate の戻り値

戻り使用タイプ	説明
give	空データで初期化された動的ストリング・オブジェクト。メモリー不足の場合は NULL を返します。

SaDynStrFree

SaDynStrFree は、SaDynStrT 変数を解放します。

検索操作では、SaDynStrMove 関数を使用して列データが SaDynStrT 変数に格納され、これによって古いデータが上書きされます。検索の終了後、ユーザーは SaDynStrFree 関数を使用して、SaDynStrT 変数を解放する必要があります。

構文

```
void SA_EXPORT_H SaDynStrFree(
    SaDynStrT* p_ds)
```

SaDynStrFree 関数は以下のパラメーターを受け入れます。

表 156. SaDynStrFree のパラメーター

パラメーター	使用タイプ	説明
p_ds	in、take	動的ストリング

注: この関数はメモリーの割り振りを解除するため、関数呼び出し後は p_ds ポインターが無効となります。したがって、使用タイプは「take」です。

戻り値

なし

SaDynStrMove

SaDynStrMove は、ストリングの値 (2 番目のパラメーター) を SaDynStrT (最初のパラメーター) にコピーします。

SaDynStrMove は、ポインターではなく、ストリングをコピーします。

SaDynStrT は、SaDynStrMove で設定する前に、SaDynStrCreate で初期化する必要があります。

注意:
SaDynStrT を (例えば memcpy を使用して) 別の SaDynStrT にコピーしないでください。2 つの SaDynStrT ポインターが同じ割り振り領域を指すことになります。

構文

```
void SA_EXPORT_H SaDynStrMove(
    SaDynStrT* p_ds,
    char* str)
```

SaDynStrMove 関数は以下のパラメーターを受け入れます。

表 157. SaDynStrMove のパラメーター

パラメーター	使用タイプ	説明
p_ds	out	動的ストリング変数を指すポインター
str	in、use	動的ストリングの新しい値

戻り値

なし

SaErrorInfo

SaErrorInfo は、サーバー接続での最後の操作からエラー情報を返します。

この関数では、カーソル・エラーは検査できません。代わりに、SaCursorErrorInfo 関数を使用する必要があります。

構文

```
bool SA_EXPORT_H SaErrorInfo(
SaConnectT* scon,
char** errstr,
int* errcode)
```

SaErrorInfo 関数は以下のパラメーターを受け入れます。

表 158. SaErrorInfo のパラメーター

パラメーター	使用タイプ	説明
scon	use	接続オブジェクトを指すポインター
errstr	out、ref	エラーが発生した場合、およびこのパラメーターが NULL 以外の場合には、*errstr にエラー・ストリングのローカル・コピーを指すポインターが格納されます。
errcode	out	エラーが発生した場合、およびこのパラメーターが NULL 以外の場合には、*errcode にエラー・コードが格納されます。

戻り値

TRUE: エラーが発生したため、errstr および errcode が更新されています。

FALSE: エラーが発生していないので、errstr および errcode が更新されていません。

SaGlobalInit

SaGlobalInit は、SA システム内でいくつかのグローバルな初期化を行います。

この関数は、SaConnect を除く他の SA 関数の前に呼び出す必要があります。他の SA 関数の前に SaConnect 関数を呼び出した場合、SaGlobalInit は SaConnect によって呼び出されるため、ユーザーが呼び出す必要はありません。

構文

```
void SA_EXPORT_H SaGlobalInit(void)
```

SaGlobalInit はパラメーターを受け入れません。

戻り値

なし

SaSetDateFormat

SaSetDateFormat デフォルトの日付形式を定義します。

使用可能な日付形式については、 183 ページの『SaDateToAsciiz』を参照してください。

構文

```
SaRetT SA_EXPORT_H SaSetDateFormat(
SaConnectT* scon,
char* dateformat)
```

SaSetDateFormat 関数は以下のパラメーターを受け入れます。

表 159. SaSetDateFormat のパラメーター

パラメーター	使用タイプ	説明
scon	in、out、use	接続オブジェクトを指すポインター
dateformat	in、use	接続のデフォルトの日付形式

注: scon パラメーターはこの関数呼び出しで変更されるため、使用タイプには「out」が含まれます。

戻り値

成功した場合には SA_RC_SUCC

サーバーへの接続が切断された場合には SA_ERR_COMERROR

関連項目

使用可能な日付、時刻、およびタイム・スタンプのフォーマットについては、 183 ページの『SaDateToAsciiz』を参照してください。

SaSetSortBufSize

SaSetSortBufSize は、接続がローカル・ソート (SA ライブラリーによってクライアント・サイドで実行されるソート) に使用するメモリーの容量を設定します。

構文

```
SaRetT SA_EXPORT_H SaSetSortBufSize(
    SaConnectT* scon,
    unsigned long size)
```

SaSetSortBufSize 関数は以下のパラメーターを受け入れます。

表 160. SaSetSortBufSize のパラメーター

パラメーター	使用タイプ	説明
scon	in、out、use	接続オブジェクトを指すポインター
size	in	バイト単位のメモリー・バッファー・サイズ

注: scon パラメーターはこの関数呼び出しで変更されるため、使用タイプには「out」が含まれます。

戻り値

OK の場合には SA_RC_SUCC、または指定されたメモリー・サイズが小さすぎる (< 10 KB) 場合には SA_ERR_FAILED

SaSetSortMaxFiles

SaSetSortMaxFiles は、接続がローカル・ソート (SA ライブラリーによってクライアント・サイドで実行されるソート) に使用するファイルの最大数を設定します。

構文

```
SaRetT SA_EXPORT_H SaSetSortMaxFiles(
SaConnectT* scon,
unsigned int nfiles)
```

SaSetSortMaxFiles 関数は以下のパラメーターを受け入れます。

表 161. SaSetSortMaxFiles のパラメーター

パラメーター	使用タイプ	説明
scon	in、out、use	接続オブジェクトを指すポインター
nfiles	in	ファイルの最大数

注: scon パラメーターはこの関数呼び出しで変更されるため、使用タイプには「out」が含まれます。

戻り値

OK の場合には SA_RC_SUCC、指定されたファイル数が少なすぎる (< 3) 場合には SA_ERR_FAILED

SaSetTimeFormat

SaSetTimeFormat は、デフォルトの時刻形式を定義します。

使用可能なフォーマットについては、 181 ページの『SaDateSetAsciiz』で SaDateSetAsciiz の時刻部分の説明を参照してください。

構文

```
SaRetT SA_EXPORT_H SaSetTimeFormat(
    SaConnectT* scon,
    char* timeformat)
```

SaSetTimeFormat 関数は以下のパラメーターを受け入れます。

表 162. SaSetTimeFormat のパラメーター

パラメーター	使用タイプ	説明
scon	in、out、use	接続オブジェクトを指すポインター

表 162. SaSetTimeFormat のパラメーター (続き)

パラメーター	使用タイプ	説明
timeformat	in	接続のデフォルトの時刻形式

注: scon パラメーターはこの関数呼び出しで変更されるため、scon の使用タイプには「out」が含まれます。

戻り値

SA_RC_SUCC

サーバーへの接続が切断された場合には SA_ERR_COMERROR

関連項目

使用可能な日付、時刻、およびタイム・スタンプのフォーマットについては、 181 ページの『SaDateSetAsciiz』を参照してください。

SaSetTimestampFormat

SaSetTimestampFormat は、デフォルトのタイム・スタンプ・フォーマットを定義します。

使用可能な日付、時刻、およびタイム・スタンプのフォーマットについては、 181 ページの『SaDateSetAsciiz』を参照してください。

構文

```
SaRetT SA_EXPORT_H SaSetTimestampFormat(
    SaConnectT* scon,
    char* timestampformat)
```

SaSetTimestampFormat 関数は以下のパラメーターを受け入れます。

表 163. SaSetTimestampFormat のパラメーター

パラメーター	使用タイプ	説明
scon	in、out、use	接続オブジェクトを指すポインター
timestampformat	in	接続のデフォルトのタイム・スタンプ・フォーマット

戻り値

SA_RC_SUCC

関連項目

使用可能な日付、時刻、およびタイム・スタンプのフォーマットについては、 181 ページの『SaDateSetAsciiz』を参照してください。

SaSQLExecDirect

SaSQLExecDirect では、CREATE TABLE、DROP TABLE、INSERT、および DELETE などの単純 SQL ステートメントを実行できます。

データをフェッチできないため、SELECT 操作は実行できません。

構文

```
SaRetT SA_EXPORT_H SaSQLExecDirect(SaConnectT* scon,
char *sqlstr)
```

SaSQLExecDirect 関数は以下のパラメーターを受け入れます。

表 164. SaSQLExecDirect のパラメーター

パラメーター	使用タイプ	説明
scon	in、use	接続オブジェクトを指すポインター
sqlstr	in、use	実行する SQL ステートメントを含むストリングを指すポインター

戻り値

SA_RC_SUCC

以下に、可能性のあるエラー・コードを示します。

- 15001: SAP_ERR_SYNTAXERROR_SD. 構文エラー: <error>, <line>。
- 15002: SAP_ERR_ILLCOLNAME_S. 正しくない列名 <name>。
- 15003: SAP_ERR_TOOMANYPARAMS. ストリング制約に対してパラメーターが多すぎる。
- 15004: SAP_ERR_TOOFEWPARAMS. ストリング制約に対してパラメーターが少なすぎる。

SaTransBegin

SaTransBegin は、新しいトランザクションを開始します。この呼び出し後は、すべての選択、挿入、更新、および削除の操作が同じトランザクション内で実行されますが、SaTransCommit を呼び出すまで、データベース内で変更が可視になりません。

SaTransBegin 呼び出しを実行しないと、サーバーはデフォルトで自動コミット・モードとなり、したがって各選択、挿入、更新、および削除の操作は、別々のトランザクションで実行されます。自動コミット・モードでは、明示的なコミット (SaTransCommit) は必要ありません。

トランザクションは、失われた更新やユニーク・エラーに関して、書き込み操作を検証するモードで実行されます。

構文

```
void SA_EXPORT_H SaTransBegin(SaConnectT* scon)
```

SaTransBegin 関数は以下のパラメーターを受け入れます。

表 165. SaTransBegin のパラメーター

パラメーター	使用タイプ	説明
scon	in、use	接続オブジェクトを指すポインター

戻り値

なし

SaTransCommit

SaTransCommit は、SaTransBegin で開始された現行トランザクションをコミットします。

この関数を呼び出すと、すべての変更がデータベース内で永続的になります。現行トランザクションが完了すると、データベース・サーバーは、次回 SaTransBegin を呼び出すまで自動コミット・モードに戻ります。

構文

```
SaRetT SA_EXPORT_H SaTransCommit(SaConnectT* scon)
```

SaTransCommit 関数は以下のパラメーターを受け入れます。

表 166. SaTransCommit のパラメーター

パラメーター	使用タイプ	説明
scon	in、use	接続オブジェクトを指すポインター

戻り値

SA_RC_SUCC またはエラー・コード

SaTransRollback

SaTransRollback は、SaTransBegin によって開始された現行トランザクションをロールバックします。データベースには、変更が行われません。

現行トランザクションが完了すると、データベース・サーバーは、次回 SaTransBegin を呼び出すまで自動コミット・モードに戻ります。

構文

```
SaRetT SA_EXPORT_H SaTransRollback(SaConnectT* scon)
```

SaTransRollback 関数は以下のパラメーターを受け入れます。

表 167. SaTransRollback のパラメーター

パラメーター	使用タイプ	説明
scon	in、use	接続オブジェクトを指すポインター

戻り値

SA_RC_SUCC またはエラー・コード

SaUserId

SaUserID は、接続の現行ユーザー ID を返します。

構文

```
int SA_EXPORT_H SaUserId(SaConnectT* scon)
```

SaUserId 関数は以下のパラメーターを受け入れます。

表 168. SaUserId のパラメーター

パラメーター	使用タイプ	説明
scon	in、use	接続オブジェクトを指すポインター

戻り値

サーバー内のユーザー ID

6 Unicode の使用

solidDB は Unicode 標準をサポートして、世界中の主要言語で使用される文字をエンコードする機能を提供しています。 Unicode エンコード・データを使用するために、アプリケーション開発において、非標準または solidDB 固有の実装環境を使用する必要はありません。つまり、標準 ODBC API または JDBC API、および solidDB ツールを使用することができます。また、solidDB は、各アプリケーションが異なるエンコードを使用するように設定されている可能性のある異機種混合のマルチ・クライアント環境もサポートしています。

Unicode データベース・モード

バージョン 6.5 から、solidDB データベースを、*Unicode* モードまたは*部分的 Unicode* モードの 2 つのモードで作成することができるようになりました。このデータベース・モードは、solidDB サーバーの文字データ型 (CHAR、VARCHAR など) のエンコードを基にしています。ワイド文字データ型 (WCHAR、WVARCHAR など) は、両方のモードで Unicode にエンコードされます。

- Unicode モード

  Unicode モードでは、文字データ型の内部表記は UTF-8 です。

  ワイド文字データ型の内部表記は、UTF-16 です。

- 部分的 Unicode モード

  部分的 Unicode モードでは、文字データ型の内部表記は特定のエンコードを使用しません。その代わりに、データはバイト・ストリングで保管されますが、この場合、ユーザー・アプリケーションがこのことを認識し、必要に応じて変換を処理するものと想定されます。

  ワイド文字データ型の内部表記は、UTF-16 です。

  solidDB バージョン 6.3 およびそれより前のバージョンで作成されたデータベースは、部分的 Unicode タイプです。

  重要: デフォルトのデータベース・モードは、部分的 Unicode です。

注: Unicode アプリケーションは、Unicode データベースと部分的 Unicode データベースの両方で作成可能です。ただし、このセクションの説明では、Unicode サポートは Unicode データベース・モードを基にしていることを想定しています。

solidDB Unicode データベースの主な機能

- **Unicode データの保管およびリトリーブ**

  Unicode データの内部表記は、UTF-8 および UTF-16 のエンコードを基にしています。ワイド文字列型のデータは、内部で UTF-16 で表記され、文字列型のデータは UTF-8 で表記されます。

つまり、単一およびマルチバイトの両方のデータを文字列型で保管することができます。主にマルチバイトのデータが予想される場合、そのマルチバイトのデータをワイド文字列型で保管するように選択して、スペースの使用効率を最適化することができます。

- **アプリケーションで使用されるエンコードへの制限がない**

  solidDB ODBC/JDBC ドライバーは、solidDB サーバーで、アプリケーション・エンコードと UTF-8/UTF-16 フォーマット間の変換を処理します。

- **アプリケーション開発に使用可能な標準 ODBC API および JDBC API**

  アプリケーション開発に対する、非標準、または solidDB 固有の要件はありません。標準 ODBC API または JDBC API を使用できます。

Unicode とは

Unicode 標準は、コンピューター処理でテキストに使用される汎用文字表現標準です。 Unicode では、一貫性のある方法で、マルチリンガルのプレーン・テキストをエンコードして、多国間でテキスト・ファイルを簡単に交換できるようにします。

Unicode 標準は、今日の主な書き言葉で使用される文字用のコード・ポイント (固有番号) を定義しています。これには、句読記号、弁別発音記号、数学記号、技術記号、矢印、および絵記号などが含まれます。合計すると、Unicode 標準は、多数の書き言葉のクラシカル・テキストおよびヒストリカル・テキストを含む、世界中のアルファベット、表意文字セット、および記号集合の 100,000 文字を超える文字に、コードを提供しています。文字は、UTF-8 および UTF-16 などさまざまなエンコード・フォームで表すことができます。

Unicode 標準は、国際標準 ISO/IEC 10646 と完全な互換性を持ち、ISO/IEC 10646 と同じ文字およびコード・ポイントがすべて含まれています。このようなコードごとの一致は、東アジアの表意文字 (漢字) も含めて、 2 つの標準のすべてのエンコード文字に当てはまります。 Unicode 標準は、文字とその使用法に関する追加情報も提供しています。Unicode に準拠するあらゆるインプリメンテーションは、ISO/IEC 10646 にも準拠しています。

エンコード形式

Unicode 文字は、32 ビット形式 (UTF-32)、16 ビット形式 (UTF-16)、および 8 ビット形式 (UTF-8) という 3 つのエンコード形式の中の 1 つで表されます。これらの文字エンコード標準は、各文字の識別情報と数値 (コード位置) を定義するだけでなく、その値のビット表現も定義します。

バージョン 6.5 から、solidDB は、文字データを表すために UTF-8 エンコードを使用し、ワイド文字データには UTF-16 エンコードを使用するように構成することができます。データベース・モードは、パラメーター **General.InternalCharEncoding** を使用して制御されます。

データベースが Unicode モード (**General.InternalCharEncoding=UTF8**) で作成される場合、以下が適用されます。

- ワイド文字列型のデータは、内部で UTF-16 で表記されます。
- 文字列型のデータは、UTF-8 で表記されます。

データベースが部分的 Unicode モード (**General.InternalCharEncoding=Raw**) で作成される場合、以下が適用されます。

- ワイド文字列型のデータは、内部で UTF-16 で表記されます。
- 文字列型のデータは、特定のエンコードでエンコードされません。その代わりに、データはバイト・ストリングで保管されますが、この場合、ユーザー・アプリケーションがこのことを認識し、必要に応じて変換を処理するものと想定されます。

UTF-8 および UTF-16 エンコードは、基本的にエンコード方式を、実装で使用される実際のビットに変換する方法です。つまり、UTF-8 エンコードと UTF-16 エンコードは同じ文字セットを共有しますが、各文字のデータ・サイズは異なります。

- UTF-16

  UTF 16 は、16 ビット文字を想定し、特定の文字範囲を拡張メカニズムとして使用することで、16 ビット文字のペアを使用して、さらに 100 万の文字にアクセスできるようにします。

- UTF-8

  UTF-8 は、すべての Unicode 文字を可変長のバイト・エンコード方式にトランスフォームする方法です。一般的な ASCII セットに対応する Unicode 文字が、ASCII と同じバイト値を持つことになるという利点と、ソフトウェアを大きく書き換えなくても、多くの既存のソフトウェアで、UTF-8 にトランスフォームされた Unicode 文字を使用できるという利点があります。

  Unicode コンソーシアムも、Unicode 標準を実装する方法として UTF-8 の使用を承認しています。16 ビットの UTF-16 形式で表現されたあらゆる Unicode 文字は、情報を失うことなく UTF-8 形式に変換することができ、その逆の変換もできます。

Unicode データベースの設計

このセクションでは、Unicode で使用するための solidDB データベースのセットアップ方法について説明します。

注: Unicode アプリケーションは、Unicode データベースと部分的 Unicode データベースの両方で作成可能です。ただし、このセクションの説明では、Unicode サポートは Unicode データベース・モードを基にしていることを想定しています。

Unicode データベースの作成

solidDB データベース・モードは、パラメーター **General.InternalCharEncoding** を使用して制御されます。

- Unicode モード: **General.InternalCharEncoding=UTF8**

  InternalCharEncoding が「UTF8」に設定されている場合、文字データ型の内部表記は UTF-8 です。文字データ型とワイド文字データ型の両方が、solidDB サーバーおよびアプリケーション間で変換されます。

- 部分的 Unicode モード: **General.InternalCharEncoding=Raw**

InternalCharEncoding が「Raw」に設定されている場合、文字データ型の内部表記は特定のエンコードを使用しません。その代わりに、データはバイト・ストリングで保管されますが、この場合、ユーザー・アプリケーションがこのことを認識し、必要に応じて変換を処理したものと想定されます。ワイド文字データ型は、solidDB サーバーおよびアプリケーション間で変換されます。

solidDB バージョン 6.3 およびそれより前のバージョンで作成されたデータベースは、部分的 Unicode タイプです。

重要: データベース・モードは、データベースの作成時に定義する必要があり、後で変更することはできません。

データベースがいずれかのモードで既に存在し、そのデータベース・モードがパラメーターの値と矛盾する場合、サーバーの始動は、`solerr.out` に以下のエラー・メッセージを出して失敗します。

```
Parameter General.InternalCharEncoding contradicts the existing database mode
```

Unicode データベースで使用するデータ型の決定

文字データ型とワイド文字データ型は両方とも、Unicode データベースに Unicode データを保管するために使用することができます。主にマルチバイトのデータが予想される場合、そのマルチバイトのデータをワイド文字列型で保管するように選択して、スペースの使用効率を最適化することができます。これは、UTF-8 エンコードと UTF-16 エンコードは同じ文字セットを共有していても、各文字のデータ・サイズが異なるためです。

- ワイド文字列型のデータ (WCHAR/WVARCHAR/LONG WVARCHAR) は、内部で UTF-16 で表記され、各文字は、2 バイトまたは 4 バイトで表記されます。
  - 基本多言語面 (BMP) の文字: 2 バイト
  - BMP 以外の文字 (代理文字): 4 バイト
- 文字列型のデータ (CHAR/VARCHAR/LONG VARCHAR) は UTF-8 で表記され、各文字は、1 バイトから 4 バイトで表記されます。

  サイズは、コード・ポイントによって異なります。

  簡単な例
  - ASCII 文字: 1 バイト
  - キリル文字、アラビア語、ヘブライ語、Latin 1 補足などの文字: 2 バイト
  - アジア言語の文字/残りの BMP 文字: 3 バイト
  - BMP 以外の文字 (代理文字): 4 バイト

例えばアジア言語では、大部分の文字が BMP に属し、2 バイトが必要になるため、ワイド文字データ型 (UTF-16) で保管したほうが効率的です。ヨーロッパ言語の場合、大部分の共通文字が 1 バイトで表記されるため、文字データ型 (UTF-8) で保管したほうが効率的です。

ワイド文字データの場合、必要な処理も少なくなります。つまり、ワイド文字データ型を使用すると、パフォーマンスが向上する場合があります。

Unicode データ型は相互運用が可能です。これは、UTF-16 と UTF-8 が同じ文字セットを共有し、どちらのデータ型を使用してもデータ損失のリスクがないからです。暗黙的な型変換を使用して、文字データ型とワイド文字データ型間で、すべてのストリング操作を行うことができます。

Unicode データ格納用列の作成

Unicode データの Unicode データベースへの保管を開始するには、以下のようにして、Unicode データ列を含む表を最初に作成する必要があります。

```
CREATE TABLE customer1 (c_id INTEGER, c_name VARCHAR,...)
CREATE TABLE customer2 (c_id INTEGER, c_name WVARCHAR,...)
```

データ列の順序付け (照合)

文字データ列は、UTF-8 のバイナリー値と、UTF-16 フォーマットのワイド文字データ列に基づいて順序付けされます (最上位バイト順を使用)。バイナリー順序が、各国語ユーザーが希望するものと異なる場合、別個の列を作成して、正しい順序付け情報を保管する必要があります。

データベース・エンティティー名での Unicode の使用

すべての SQL ステートメントで Unicode 名を二重引用符で囲むだけで、Unicode ストリングを使用して表、列、プロシージャーなどのデータベース・エンティティーに名前を付けることができます。

solidDB ツールは、環境のデフォルト・ロケール、または指定されたロケールに従って、Unicode ストリングを処理することができます。

詳しくは、『Unicode での solidDB ツールの使用』を参照してください。

ユーザー名およびパスワードでの Unicode の使用

ユーザー名とパスワードにも Unicode ストリングを使用することができます。ただし、さまざまなツールからのアクセスの問題を回避するために、元のデータベース管理者のアカウント情報を純粋な ASCII ストリングとして提供する必要があります。

ファイル名での Unicode の使用

Unicode ストリングは、いずれのファイル名にも使用できません。

Unicode での solidDB ツールの使用

このセクションでは、Unicode データベースおよび部分的 Unicode データベースでの solidDB ツールの使用方法について説明します。

以下の solidDB ツールを使用して、Unicode データベースと部分的 Unicode データベースの両方において、システムのデフォルト・ロケール、または指定されたロケールで、データを出力およびインポートすることができます。

- solidDB SQL エディター (solsql)
- solidDB データ・ディクショナリー (soldd)

- solidDB エクスポート (solexp)
- solidDB Speed Loader (solload)

solidDB リモート制御 (solcon) は、UTF-8 へのデータの変換をサポートしていません。例えば、solcon に出力されたエラー・メッセージに Unicode でエンコードされたデータが含まれている場合、そのメッセージは、コンソールで正しく表示されません。

変換に使用されるロケールは、ツールの開始時に、コマンド行オプションで定義されます。

重要:

- solidDB ツールは solidDB ODBC API 3.5.1 を使用します。つまり、文字データ型のバインディング方式がサーバー・サイドの **Srv.ODBCDefaultCharBinding** パラメーターまたはクライアント・サイドの client-side **Client.ODBCCharBinding** パラメーターで定義されている場合、この設定は solidDB ツールに対しても有効になります。
- Unicode データベースと部分的 Unicode データベースは、CHAR および WCHAR のデータ型の変換に関して、動作が異なります。
  - **Unicode データベース**

    CHAR と WCHAR のデータ型は、solidDB の UTF-8/UTF-16 フォーマットと、選択されたバインディング方式で定義されたロケール/コードページ間で変換されます。
  - **部分的 Unicode データベース**

    CHAR データ型は変換されません。その代わり、部分的 Unicode データベースに CHAR データを保管するために使用される raw (バイナリー) フォーマットで処理されます。

    WCHAR データ型は、solidDB の UTF-16 フォーマットと、選択されたバインディング方式で定義されたロケール/コードページ間で変換されます。

表 169. 部分的 Unicode および Unicode データベース用の solidDB ツールのコマンド行オプション

オプション	説明
オプションなし/工場出荷時設定	`solid.ini` ファイルのサーバー・サイド・パラメーターまたはクライアント・サイド・パラメーターでオーバーライドされない限り、コンソールのロケール設定が使用されます。
`-m`	`solid.ini` ファイルのサーバー・サイド・パラメーターまたはクライアント・サイド・パラメーターに関わらず、コンソールのロケール設定が使用されます。

表 169. 部分的 Unicode および Unicode データベース用の solidDB ツールのコマンド行オプション (続き)

オプション	説明
-M<locale_name>	ロケール・コンソール設定は、<locale_name> で定義されたロケールによってオーバーライドされます。<locale_name> は、オペレーティング・システムによって異なります。 例えば、Linux 環境では、中国語 (簡体字)/中国のコード・ページ GB18030 のロケール名は zh_CN.gb18030 です。 Windows 環境では、フィンランド語/フィンランドの Latin1 コード・ページのロケール名は fin_fin.1252 です。
-u	入出力は強制的に UTF-8 になります。

注: solid.ini ファイルのサーバー・サイド・パラメーターまたはクライアント・サイド・パラメーターが「Raw」バインディングを使用するように設定されている場合、常に -m、-M、または -u オプションを使用して solid.ini の設定をオーバーライドする必要があります。

Unicode データベースと部分的 Unicode データベース間の互換性

データベースが Unicode モードで作成された場合、それを部分的 Unicode モードに変更することはできません。その逆の変更も行えません。部分的 Unicode データベースを Unicode に変換 (またはその逆に変換) する必要がある場合、solidDB ツールを使用して、データベースをエクスポートおよび再ロードすることができます。

データベースがいずれかのモードで既に存在し、そのデータベース・モードがパラメーターの値と矛盾する場合、サーバーの始動は、solerr.out に以下のエラー・メッセージを出して失敗します。

```
Parameter General.InternalCharEncoding contradicts the existing database mode
```

部分的 Unicode データベースの Unicode への変換

部分的 Unicode データベースを Unicode データベースに変換するには、solidDB ツールを使用して、データベースをエクスポートおよび再ロードします。

始める前に

- データベースのバックアップを作成します。
- 部分的 Unicode データベースの CHAR データ型列のデータをエンコードするためにアプリケーション・サイドで使用されるロケール/コードページを検証します。

  エクスポート・フェーズ中に、CHAR データ型列にあるデータは solidDB ツールによって変換されずに、そのまま出力されます。つまり、CHAR データ型の基本となるロケール/コードページが、出力ファイルのロケール/コードページ・フォーマットになります。出力ファイルに単一のロケール/コードページでデータが含まれるようにするには、solidDB が、WCHAR データ型列のデータを、UTF-16 から、CHAR データと正確に同じロケール/コードページ・フォーマットに変換できる必要があります。

インポート・フェーズで、solidDB ツールは、出力ファイルのロケール/コードページ・フォーマットから、Unicode データベースで使用される UTF-8 (CHAR) および UTF-16 (WCHAR) エンコードに、データを変換します。

このタスクについて

この手順では、例として以下のセットアップが使用されます。

- サーバー名は solidDB、接続に使用されるプロトコルは TCP/IP でポート 1964 を使用します (ネットワーク名は「tcpip 1964」)。
- 部分的 Unicode データベースは、ユーザー名「dbadmin」、パスワード「password」を使用して作成されています。
- 部分的 Unicode データベースの CHAR データ型は、ロケール zh_CN.gb18030 (中国語 (簡体字)/中国、およびコードページ GB18030) を使用して、アプリケーション・サイドでエンコードされます。

ヒント: 独自のデータベース作成スクリプトを使用できる場合は、それらを使用して、新規データベース表定義を作成することができます。この場合、それらをエクスポートおよびインポートするために、soldd および solsql を使用する必要はありません。

手順

1. **solidDB データ・ディクショナリー (soldd) を使用して、データ定義を抽出します。**

   以下のコマンドを使用して、すべての表、ビュー、トリガー、索引、プロシージャー、シーケンス、およびイベントの定義を含む SQL スクリプトを抽出します。

   ```
   soldd -Mzh_CN.gb18030 "tcpip 1964" dbadmin password
   ```

   デフォルトのファイル名 `soldd.sql` が使用されます。

   注: セキュリティー上の理由から、ユーザー定義とロール定義はリストされていません。データベースにユーザーまたはロールが含まれている場合、それらの `CREATE` ステートメントを、抽出した SQL ファイルに手動で追加します。

   重要: 参照整合性を保持するために、表定義ステートメントを再編成して、参照される表が、参照を行う表より前に作成されるようにしなければならない場合があります。

2. **solidDB エクスポート (solexp) を使用して、データベースからデータを抽出します。**

   以下のコマンドを使用して、すべての表の制御ファイルおよびデータ・ファイルを抽出してください。

   ```
   solexp -Mzh_CN.gb18030 "tcpip 1964" dbadmin password *
   ```

   このエクスポートにより、各表に対して、制御ファイル (`<table_name>.ctr`) およびデータ・ファイル (`<table_name>.dat`) が作成されます。デフォルトのファイル名は、エクスポートした表の名前と同じです。

3. **新規 Unicode データベースを作成します。**

a. **General.InternalCharEncoding** パラメーターを「UTF8」に設定します。

```
[General]
InternalCharEncoding=UTF8
```

b. 新規 Unicode データベース用の作業ディレクトリーで solidDB を始動して、新規データベースを作成します。

4. **solidDB SQL エディター (solsql) を使用して、新規データベースにデータ定義をインポートします。**

   以下のコマンドを使用して、solidDB データ・ディクショナリー (soldd) によって作成された SQL スクリプトを実行します。

   ```
   solsql -fsoldd.sql -Mzh_CN.gb18030 "tcpip 1964" dbadmin password
   ```

5. **solidDB Speed Loader (solload) を使用して、新規データベースにデータをロードします。**

   各表に対して、以下のコマンドを使用して、新規データベースにデータをロードします。

   ```
   solload -Mzh_CN.gb18030 "tcpip 1964" dbadmin password <table_name>.ctr
   ```

関連トピック

- 「*IBM solidDB 管理者ガイド*」の『*solidDB データ管理ツールの使用*』

Unicode に対応したアプリケーションの開発

このセクションでは、solidDB で、アプリケーションを Unicode データで使用するように設計する方法を説明します。

サポートされるインターフェース

- **ODBC**

  solidDB ODBC ドライバーは Unicode 準拠であり、Microsoft ODBC 3.51 標準に準拠しています。

  注: solidDB は、Unicode 用および ASCII 用として、2 つのバージョンの ODBC ドライバーを提供しています。Unicode バージョンは ASCII バージョンのスーパーセットであり、Unicode 文字セットと ASCII 文字セットで使用できます。

- **JDBC**

  Unicode は solidDB JDBC ドライバーでサポートされています。これは、JDBC 2.0 標準の solidDB 実装です。

  Java はネイティブで Unicode ストリングを使用するため、Unicode のサポートとは、主に、solidDB で文字データにアクセスするときに、データ型の変換が不要であることを意味します。さらに、JDBC ResultSet Class メソッドである getUnicodeStream および setUnicodeStream が、solidDB に保管された大容量の Unicode テキストの処理用にサポートされます。

- **solidDB Light Client**

solidDB Light Client は、ODBC 3.5 以降の API 機能をサポートしていないため、Unicode をサポートしません。

異なるロケール設定を持つマルチ・クライアント環境 (Unicode データベースのみ)

Unicode データベースでは、solidDB ODBC ドライバーおよび JDBC ドライバーが、solidDB サーバーで、アプリケーション・エンコードと UTF-8/UTF-16 フォーマット間の変換を処理します。

ODBC 環境では、アプリケーション・バッファーでのエンコード用に、アプリケーションのデフォルト・ロケール、またはユーザー定義のロケールが使用されるように変換を設定することができます。これは、サーバー・サイドの **Srv.ODBCDefaultCharBinding** 構成パラメーターおよびクライアント・サイドの **Client.ODBCCharBinding** 構成パラメーターで制御されます。

SQL ストリング関数

SQL ストリング関数は、予想通りに機能します。変換は必要に応じて暗黙に行われます。いずれかのオペランドがワイド文字型である場合、結果は、常にワイド文字型になります。

関数 UPPER() および LOWER() は、文字が Latin 1 コード・ページの一部である場合にのみ、Unicode ストリングで、大文字または小文字の変換を実行します。Unicode 文字を大文字または小文字に変換できない場合、入力ストリングはそのまま返されます。

ODBC アプリケーション・データベースおよび Unicode データベース

ODBC 環境では、solidDB ODBC ドライバーが、アプリケーション (クライアント) で使用されるエンコードと、solidDB Unicode データベースの UTF-8/UTF-16 フォーマット間のデータ変換を処理します。文字データのバインディングは、サーバー・サイドのパラメーター **Srv.ODBCDefaultCharBinding** を使用して、すべてのクライアントに対して設定することも、クライアント・サイドのパラメーター **Client.ODBCCharBinding** を使用して、クライアントごとに設定することもできます。いずれの場合でも、標準 C 型識別子の SQL_C_CHAR が使用されます。

文字データのバインディングでは、以下の方式の 1 つを使用するように ODBC ドライバーを設定することができます。

- 現在のクライアント・ロケール・エンコード
- ロケール名で定義された特定のエンコード
- エンコードなし
- UTF-8 エンコード

すべての方式に対して、以下の 2 つのユース・ケースがサポートされます。

- サーバー・サイド・パラメーター **ODBCDefaultCharBinding** を使用して、すべてのクライアントに対して同じバインディング方式を設定する。

  ```
  [Srv]ODBCDefaultCharBinding=raw|locale|locale:|locale:<locale name>|UTF8
  ```

- クライアント・サイド・パラメーター **ODBCCharBinding** を使用して、クライアントごとにバインディング方式を設定する。

  ```
  [Client]
  ODBCCharBinding=raw|locale|locale:<locale name>
  ```

  ODBCCharBinding パラメーターは、**ODBCDefaultCharBinding** で設定されるサーバー・サイド設定をオーバーライドします。

ファクトリー値は、両方とも `locale:` です。

- `raw` — solidDB サーバーとクライアント間でデータ変換は行われません

  値「raw」は、バージョン 6.3 またはそれ以前の solidDB で使用しているバインディングをデータベースで使用する場合に利用できます。
- `locale` — クライアント・システムで設定される場合も、現在のクライアント・ロケール設定が使用されます
- `locale:` — 現在のクライアント設定が、クライアント・システムのデフォルト・ロケール・セットでオーバーライドされます

  ドライバーは、空のストリングを使用して setlocale() を呼び出します。これにより、システムに設定されたロケール設定が効率的に検索されます。

  例えば Linux 環境では、環境変数の LC_CTYPE が最初に検査され、それが定義されていなければ、環境変数 LANG が検索されます。
- `locale:<locale name>` — 現在のクライアント・システム設定がオーバーライドされ、指定されたロケールが使用されます

  `<locale name>` の規則は、オペレーティング・システムにより異なります。

  例えば、Linux 環境では、中国語 (簡体字)/中国のコード・ページ GB18030 のロケール名は `zh_CN.gb18030` です。 Windows 環境では、フィンランド語/フィンランドの Latin1 コード・ページのロケール名は `fin_fin.1252` です。
- `UTF8` — クライアント・サイド・システムに設定されたロケールに関わらず、UTF-8 バインディングが強制されます。

注:

- **Srv.ODBCDefaultCharBinding** の値が `locale` 以外の場合、すべてのクライアントに対して、現在のすべてのシステム・ロケール設定をオーバーライドします。
- **Client.ODBCCharBinding** の値が `locale` 以外の場合、サーバー・サイド値 (設定されている場合) と、現在のシステム・ロケール設定の両方をオーバーライドします。

現在のクライアント・ロケール・エンコードを使用する場合 (locale)

現在のクライアント・ロケール・エンコードを使用するには、以下のようにします。

1. **次のように、パラメーター設定を構成します。**
   - **すべてのクライアントが同じバインディング方式を使用する場合 (サーバー・サイド・パラメーター)**

サーバー・サイド solid.ini のセクション [Server] に、
ODBCDefaultCharBinding パラメーターを設定します。

```
[Srv]ODBCDefaultCharBinding=locale
```

- **一部またはすべてのクライアントが異なるバインディング方式を必要とする場合 (クライアント・サイド・パラメーター)**

  クライアント・サイド solid.ini のセクション [Client] に、
  ODBCCharBinding パラメーターを設定します。

  ```
  [Client]
  ODBCCharBinding=locale
  ```

  クライアント・サイド・パラメーターは、サーバー・サイド設定をオーバーライドします。

2. **アプリケーションが setlocale() を呼び出すように設定します。**

特定のロケール・エンコードを使用する場合 (locale:<locale_name>)

特定のロケール・エンコードを使用するには、以下のようにします。

solid.ini にロケールを定義します。

- **すべてのクライアントが同じバインディング方式を使用する場合 (サーバー・サイド・パラメーター)**

  サーバー・サイド solid.ini のセクション [Server] に、
  ODBCDefaultCharBinding パラメーターを設定します。

  ```
  [Srv]ODBCDefaultCharBinding=locace:<locale name>
  ```

  例えば Linux 環境では、以下のようにします。

  ```
  [Srv]ODBCDefaultCharBinding=locale:zh_CN.gb18030
  ```

- **一部またはすべてのクライアントが異なるバインディング方式を必要とする場合 (クライアント・サイド・パラメーター)**

  クライアント・サイド solid.ini のセクション [Client] に、
  ODBCCharBinding パラメーターを設定します。

  ```
  [Client]
  ODBCCharBinding=locale:<locale name>
  ```

  クライアント・サイド・パラメーターは、サーバー・サイド設定をオーバーライドします。

  例えば Linux 環境では、以下のようにします。

  ```
  [Client]
  ODBCCharBinding=locale:zh_CN.gb18030
  ```

注: 特定のロケールを設定すると、setlocale() で定義されたアプリケーション設定がオーバーライドされます。

例 1

すべてのクライアントが、クライアントの現在のロケールを使用します。各クライアントが、異なるコード・ページを使用することができます。

以下のように、サーバー・サイドの solid.ini が使用されます。

```
[Srv]ODBCDefaultCharBinding=locale
```

例 2

クライアントの現在のロケールを使用するクライアントもありますが、Latin1 コード・ページを使用するクライアントもあります。

以下のように、サーバー・サイドの solid.ini が使用されます。

```
[Srv]ODBCDefaultCharBinding=locale
```

Latin1 コード・ページが必要なクライアントでは、以下のように、クライアント・サイドの solid.ini が使用されます。

```
[Client]
ODBCCharBinding=locale:fin_fin.1252
```

JDBC アプリケーション・データベースおよび Unicode データベース

JDBC 環境では、solidDB JDBC ドライバーが、アプリケーション (クライアント) で使用されるエンコードと、Unicode データベースの UTF-8/UTF-16 フォーマット間のデータ変換を処理します。 JDBC で Unicode を使用するために、solidDB 固有の設定を作成する必要はありません。

7 トランザクション・ログ・リーダーの使用

solidDB トランザクション・ログ・リーダーは、solidDB トランザクション・ログからトランザクションごとにログ・レコードを読み取ることができるソリューションです。ログ・リーダーは、SYS_LOG と呼ばれる読み取り専用の仮想表を基にしています。この仮想表では、各行が単一のログ項目に対応しています。 SYS_LOG 表には、SQL ステートメントを使用して、ODBC および JDBC ドライバーでアクセスすることができます。

SYS_LOG 表は仮想表です。ログ・リーダーが SYS_LOG 表に対して SQL 要求を受け取ると、内部ログ構造から、該当する結果セットが動的に生成されます。各ログ読み取りは、異なるログ・レコードから開始することができます。

トランザクション・ログにある各項目ごとに、SYS_LOG 表には、ログ・レコード、実行されたトランザクションおよびステートメントの型、および変更されたデータ自体を持つ行を識別するデータが含まれています。

SYS_LOG 表は、ローカルおよびリモートの両方から、ODBC および JDBC が使用可能なアプリケーションによって読み取ることができます。複数のアプリケーションが、干渉なしで同時に SYS_LOG 表を読み取ることができます。

SYS_LOG 表に関して詳しくは、「*IBM solidDB SQL ガイド*」の付録『*データベース仮想表*』のセクション『*SYS_LOG*』を参照してください。

ログ・リーダーを使用したアプリケーション開発に関する考慮事項

サポートされている表タイプ

- インメモリー表とディスク・ベース表の両方がサポートされています。
- トランジエント表およびテンポラリー表はサポートされていません。

  トランジエント表およびテンポラリー表はログ記録されないため、これらの表内のデータはログ・リーダーを介して返されません。

サポートされるデータベース操作

- コミットされたトランザクションのみがログ・リーダーによって返されます。

  1 つのトランザクションに対するすべてのイベントは、一度に返されます。各トランザクションは、コミット時にすべて返されます。ログ・リーダーは、コミットされたすべてのトランザクションを、ログにコミットされた順番で返します。重複するトランザクションは、トランザクションごとに返されます。
- トリガーはサポートされています。

  トリガー・アクション部分の操作は、通常のユーザー操作としてログに記録されます。トリガーでの書き込み操作は、操作が実行されるとログに記録されます。つまり、トリガー前の操作はユーザー・データ操作の前にログに記録され、トリガー後の操作はユーザー・データ操作の後にログに記録されます。

- カスケード・アクションはサポートされています。

  参照アクションのカスケードによる操作は、通常のユーザー操作としてログに記録されます。カスケード操作は、実際のユーザー・データ操作の後にログに記録されます。
- DDL 操作はサポートされています。

  DDL 操作の場合、ログ・リーダーは、元の SQL ステートメントを含む特別な DBE_LOGREADER_LOG_REC_DDL レコードを返します。

キャッチアップ・モードおよびライブ・データ・モード

ログ読み取りが SYS_LOG から始まる場合、読み取りは最初にキャッチアップ・モードに入ります。キャッチアップ・モードでは、ログ読み取りの開始位置がログから検索され、その位置から読み取りが開始されます。ログ読み取りが現在のログの最後に達すると、ライブ・データの読み取りを開始します。ライブ・モードでは、トランザクションは、トランザクションが実行されると返されます。

複数のログ・リーダーが使用されている場合、ログ・リーダーごとにいずれかのデータ・モードになります。

ライブ・データ・モードでは、使用可能なデータがない場合でも、毎秒、カーソルが返されます。この場合、SYS_LOG 表の FLAGS フィールドはゼロです。

主キー

主キーは、表に対して必須ではありません。システムで生成された内部および非表示の主キー値は、ログ・リーダーを介して返されません。

データベースを設計する際に、主キーが定義されていない場合に行を識別する方法を決定する必要があります。

高可用性

高可用性 (HotStandby) がサポートされているので、1 次サーバーと 2 次サーバーの間でログ・ファイルの内容とログ・アドレスの互換性が保たれます。ログが 1 次サーバーから読み取られ、フェイルオーバーが発生した場合、SYS_LOG からの新規読み取りは、古い 1 次サーバーから受け取った最後の LOGADDR を使用して開始することができます。

ログは、2 次サーバーから読み取ることもできます。例えばロード・バランシングに関して、この機能は便利です。

スロットル

クライアントがログ・レコードを読み取るスピードより、サーバーがログ・レコードを生成するスピードの方が速い場合、スロットルが生じる場合があります。つまり、ログ・リーダーがライブ・データから遅れ過ぎないようにするために、サーバーに書き込むユーザー・トランザクションがスローダウンされます。
LogReader.MaxSpace パラメーターを使用して、スロットルが生じる前に行われるバッファリングを制御することができます。

読み取りを開始したがその後で読み取りを停止したアプリケーションが原因で、サーバーが停止することもあります。

ログの最大サイズ

ログ・リーダーを使用しているアプリケーションが長時間にわたって停止または終了した場合、ログの最大サイズに到達することがあります。このような場合、エラー・メッセージは生成されません。また、キャッチアップ用にアプリケーションによって保管された位置は使用できなくなり、キャッチアップ操作は失敗します。

アクセス権限

SYS_LOG 表へのアクセス、パーティションへの表の追加、またはパーティションからの表の除去を行うには、管理者権限が必要です。

ログ・リーダーの構成

ログ・リーダーは、solid.ini 構成ファイル内の LogReader セクションにあるサーバー・サイド構成パラメーターで構成されています。

このタスクについて

重要: LogReaderEnabled、**MaxSpace**、および **MaxLogSize** の各パラメーターも、solidDB Universal Cache および CDC Replication で使用されます。

手順

- **LogReaderEnabled** を「yes」に設定して、ログ・リーダーを有効にします。

  これにより、ログ・リーダーは SYS_LOG からの読み取りが可能になります。トランザクションのロギング・モードも、より詳細になります。
- ご使用の環境に応じて、以下のパラメーターを設定します。
  - **MaxSpace** 値を設定して、スロットルが生じる前にメモリーのバッファーに入れられるログ・レコードの最大数を定義します。
  - **MaxLogSize** 値を設定して、キャッチアップに使用可能なログの最大サイズを定義します。

    定義したサイズにログが到達すると、古いログ・データが削除され、それより古い LOGADDR ログ位置からのキャッチアップはできなくなります。
  - **MaxMemLogSize** 値を設定して、ロギングが無効な場合の (**Logging.LogEnabled=No**)、メモリー内のログ・リーダー・ログ・ファイルの最大サイズを定義します。

例

```
[LogReader]
LogReaderEnabled=yes   ;デフォルト: no
;
;MaxLogSize=100000       ;デフォルト: 10240 (MB)
; 考えられるキャッチアップのために常に保持されるログ・ファイルの
; 量 (MB 単位)。このサイズは、妥当なキャッチアップの最大サイズ
; に応じて調整される必要があります。宣言されるスペースは、
; 常に完全に占有されます。
;
```

```
MaxSpace=500000          ;デフォルト:100000
; レコードのスロットルに使用されるインメモリー・ログ・リーダー・バッファーの
; サイズ。バッファーがいっぱいになると、スロットル (スローダウン)
; が行われます。バッファーが使用されると、サイズは、solidDB サーバー・プロセスの
: フットプリントにまでなります。
```

ログ・リーダーを使用したログ・データの読み取り

solidDB トランザクション・ログは、ログ・リーダー固有の SELECT ステートメントを使用して読み取ることができます。特定のログ位置 (LOGADDR) から、ログ読み取りを開始することもできます。

このタスクについて

同時に複数のアクティブな SELECT ステートメントが一度に存在することが可能でで、それぞれのステートメントは (LOGADDR で定義される) 異なるログ・レコードから開始できます。各 SELECT は、互いに独立しています。

手順

- **ログ・データの読み取り**

  ログを読み取るための基本構文は、以下のとおりです。

  ```
  SELECT RECID, RELID, FLAGS, LOGADDR, DATA FROM SYS_LOG WHERE LOGADDR > ?;
  ```

  WHERE 条件は、LOGADDR フィールドに対してのみ許可されます。許可される制約は、より大きい (>) のみです。他のフィールドに制約を付けると、エラーが生じます。

- **指定したログ位置からのログ読み取りの開始**

  1. 現在の LOGADDR 値をリトリーブします。

     ```
     SELECT LOGADDR FROM SYS_LOG LIMIT 1;
     ```

  2. SELECT ステートメントの LOGADDR を定義します。

タスクの結果

読み取りが開始すると、フェッチ呼び出しにより、行が返され始めます。使用可能なデータがない場合、サーバーは 1 秒間隔で空のデータを返します。このような場合、新規フェッチ呼び出しを発行して、新規データが使用可能かどうかを確認することができます。

例

1. 以下のようにしてテスト表を作成します。

   ```
   CREATE TABLE TEST(I INTEGER NOT NULL PRIMARY KEY, C VARCHAR);
   COMMIT WORK;
   ```

2. 以下のようにして、現在のログ・アドレスを取得します。

   ```
   SELECT LOGADDR FROM SYS_LOG LIMIT 1;
   LOGADDR
   -------
   0000000000000001FFFFFFFF0000029500000295
   1 rows fetched.
   ```

3. 以下のようにして、表にデータを挿入します。

```
INSERT INTO TEST VALUES(1,'test1');
COMMIT WORK;
```

4. 以下のようにして、SYS_LOG 表から行を読み取ります。

```
SELECT RECID,RELID,FLAGS,LOGADDR,DATA FROM SYS_LOG WHERE LOGADDR > '00000000000000001FFFFFFFF0000029500000295';
RECID RELID     FLAGS LOGADDR                                 DATA
----- -----     ----- -------                                 ----
    7     0         1 00000000000000002000000010000029F0000029F NULL
    1 10000         1 00000000000000002000000020000029F0000029F 00000004000000010000000574657374 31FFFFFFFE
   12     0         1 00000000000000002000000030000029F0000029F NULL
 NULL  NULL         0 NULL                                      NULL
```

ログ・レコードのパーティショニングおよびフィルタリング

デフォルトでは、アプリケーションによって生成されるすべてのログ・レコードが返されます。データベースのサブセットのログ・レコードにのみアクセスする場合は、ログ・リーダーのパーティションを指定することができます。ログ・リーダーのパーティションは、名前付きの表の集合です。パーティションは重複する場合があります。

パーティションに関する情報は、SYS_FEDT_DB_PARTITION システム表および SYS_FEDT_TABLE_PARTITION システム表に保管されています。

パーティションの作成および削除

ログ・リーダーのパーティションは、SQL ステートメントを使用して、作成、変更、および削除されます。

このタスクについて

パーティション設定は、トランザクション用でパーシスタントです。

手順

- **パーティションの作成**

  以下のコマンドを使用して、パーティションを作成します。

  ```
  CREATE LOGREADER PARTITION <partition-name>
  ```

- **パーティションの削除**

  以下のコマンドを使用して、パーティションを削除 (ドロップ) します。

  ```
  DROP LOGREADER PARTITION <partition-name>
  ```

- **パーティションの変更**

  以下のコマンドを使用して、パーティションに表を追加したり、パーティションから表を除去したりします。

  ```
  ALTER LOGREADER PARTITION <partition-name> {ADD | DROP} TABLE <table-name>
  ```

  注: 表がパーティションの一部である場合、その表をドロップしたり、変更したりすることはできません。使用できる ALTER ステートメントは、ALTER LOGREADER PARTITION ... DROP TABLE のみです。

パーティション・フィルターの使用

セッション固有のパーティション・フィルターを設定して、特定のパーティションからのみ、ログ・レコードを読み取ることができます。

このタスクについて

パーティション・フィルターに対する設定は、現在のセッションのみに適用され、その設定後に開始された SYS_LOG からのすべての読み取りに対して有効になります。パーティション・データは、SYS_FEDT_DB_PARTITION システム表に保管されます。

手順

- 以下のステートメントを使用して、パーティション・フィルターを設定します。

  ```
  SET LOGREADER PARTITION { <partition-name> | NONE }
  ```

  NONE に設定すると (デフォルト)、すべてのログ・レコードが読み取られます。
- SELECT ステートメントを使用して、SYS_FEDT_DB_PARTITION システム表にある既存のパーティションを表示します。

  例えば、以下のようにします。

  ```
  SELECT * FROM sys_fedt_db_partition
  ```

トランザクション・バッチの設定

トランザクション・バッチを使用すると、ログからの複数のトランザクションを単一のトランザクションのように返すことができます。トランザクション・バッチ・サイズは、セッションに応じて設定されます。ログ記録されたトランザクションが別のデータベースに送られる場合、トランザクション・バッチによって、ネットワーク全体の読み取りパフォーマンスを向上させることができます。例えば、多数の挿入を 1 つのトランザクションでバッチ処理できるため、別のデータベースでトランザクションを実行する必要がある場合に、1 つのトランザクションしか実行しなくて済みます。

このタスクについて

トランザクション・バッチ設定は、現在のセッションにのみ適用され、その設定後に開始された SYS_LOG からのすべての読み取りに対して有効になります。

手順

以下のコマンドを使用して、トランザクション・バッチ・サイズを設定します。

```
SET LOGREADER BATCH <size>
```

デフォルトのバッチ・サイズは 1 で、トランザクションのバッチ処理は行われません。

タスクの結果

トランザクションのバッチ・サイズを設定しても、キャッチアップ位置は変更されません。つまり、同じキャッチアップ位置を使用して、もう一度すべてのトランザクションをバッチ処理で読み取ることができます。

付録 A. solidDB がサポートする ODBC 関数

このトピックでは、solidDB がサポートする ODBC 関数について説明します。

表 170. solidDB がサポートする ODBC 関数

関数名/導入されたバージョン [1]	目的	ODBC の使用時の可用性	規格適合 [2]
データ・ソースへの接続			
SQLAllocEnv (1.0)	N/A	推奨されない (SQLAllocHandle で置き換え)	N/A
SQLAllocConnect (1.0)	N/A	推奨されない (SQLAllocHandle で置き換え)	N/A
SQLAllocHandle (3.0)	サポートされているデータ・ソース属性のリストを返します。 インストールされているドライバーとその属性のリストを返します。	サポートされる サポートされる	ISO 92 ODBC
SQLConnect (1.0)	ドライバーおよびデータ・ソースへの接続を確立します。接続ハンドルは、状況、トランザクション状態、およびエラー情報を含めて、データ・ソースへの接続に関するすべての情報のストレージを参照します。	サポートされる	ISO 92
SQLDriverConnect (1.0)	この関数は、SQLConnect の代替となります。ユーザーに対してすべての接続情報の入力を促すプロンプトを出すダイアログ・ボックスや、システム情報に定義されていないデータ・ソースを含めて、SQLConnect の 3 つの引数より多くの接続情報を必要とするデータ・ソースをサポートします。	サポートされる (この関数の Unicode バージョンを含む)	ODBC
SQLBrowseConnect (1.0)	属性および属性値の連続レベルを返します。すべてのレベルを列挙し終わると、データ・ソースへの接続が完了し、完全な接続ストリングが返されます。 SQL_SUCCESS_WITH_INFO が返された場合、すべての接続情報が指定され、この時点でアプリケーションがデータ・ソースに接続されていることを示しています。	サポートされない	ISO 92
SQLGetInfo (1.0)	接続に関連付けられたドライバーおよびデータ・ソースに関する一般情報を返します。	サポートされる	ISO 92

表 170. solidDB がサポートする ODBC 関数 (続き)

関数名/導入されたバージョン [1]	目的	ODBC の使用時の可用性	規格適合 [2]
SQLGetFunctions (1.0)	ドライバーが特定の ODBC 関数をサポートするかどうかに関する情報を返します。	サポートされる。この関数は、ODBC ドライバー・マネージャー内にインプリメントされています。また、ドライバー内にもインプリメントできます。ドライバーに SQLGetFunctions がインプリメントされている場合、ドライバー・マネージャーは、ドライバー内でこの関数を呼び出します。それ以外の場合、ドライバー・マネージャー自体でこの関数を実行します。solidDB の場合、この関数はドライバー内にインプリメントされているため、ドライバーにリンクしたアプリケーションは、アプリケーションからこの関数を呼び出すことができます。	ISO 92
SQLGetTypeInfo (1.0)	データ・ソースでサポートされているデータ型に関する情報を返します。ドライバーは、SQL 結果セットの形式で情報を返します。このデータ型は、データ定義言語 (DDL) ステートメントで使用するためのものです。	サポートされる	ISO 92
ドライバーとデータ・ソースに関する情報の取得			
SQLDataSources (1.0)	データ・ソースに関する情報を返します。	サポートされる。この関数は、ODBC ドライバー・マネージャー内にインプリメントされています。 Microsoft ODBC ドライバー・マネージャーを持たない Microsoft Windows 以外のプラットフォームでは、この関数はサポートされません。	ISO 92

表 170. solidDB がサポートする ODBC 関数 (続き)

関数名/導入されたバージョン [1]	目的	ODBC の使用時の可用性	規格適合 [2]
SQLDrivers (2.0)	ドライバーの説明とドライバー属性キーワードをリストします。	サポートされる。この関数は、ODBC ドライバー・マネージャー内にインプリメントされています。 Microsoft Windows では、solidDB に接続するアプリケーションが OLE DB または ADO API を使用する場合、または Microsoft Access、FoxPro、Crystal Reports など、ドライバー・マネージャーを必要とするデータベース・ツールを使用する場合、ドライバー・マネージャーが必要になります。 Microsoft Windows 以外のプラットフォームでは、ドライバー・マネージャーは、iODBC、Merant、および UnixODBC などのベンダーから提供されます。	ODBC
SQLGetConnectAttr (3.0) SQLSetConnectAttr (3.0)	接続属性の値を返します。 接続属性を設定します。	サポートされる サポートされる	ISO 92 ISO 92
SQLGetEnvAttr (3.0) SQLSetEnvAttr (3.0)	環境属性の値を返します。 環境属性を設定します。	サポートされる サポートされる	ISO 92 ISO 92
SQLGetStmtAttr (3.0) SQLSetStmtAttr (3.0)	ステートメント属性の値を返します。 ステートメント属性を設定します。	サポートされる サポートされる	ISO 92 ISO 92
SQLSetConnectOption (1.0) SQLGetConnectOption (1.0)	N/A	推奨されない (SQLSetConnectAttr で置き換え) 推奨されない (SQLGetConnectAttr で置き換え)	N/A N/A
SQLGetStmtOption (1.0) SQLSetStmtOption (1.0)	N/A N/A	推奨されない (SQLGetStmtAttr で置き換え) 推奨されない (SQLSetStmtAttr で置き換え)	N/A N/A
記述子フィールドの設定とリトリーブ			

表 170. solidDB がサポートする ODBC 関数 (続き)

関数名/導入されたバージョン [1]	目的	ODBC の使用時の可用性	規格適合 [2]
SQLGetDescField (3.0) SQLSetDescField (3.0)	単一記述子フィールドの現行の設定または値を返します。 記述子レコードの単一フィールドの値を設定します。	サポートされる サポートされる	ISO 92 ISO 92
SQLGetDescRec (3.0) SQLSetDescRec (3.0)	記述子レコードの複数フィールドの現行の設定または値を返します。返されたフィールドは、列またはパラメーターのデータの名前、データ型、およびストレージを記述します。 列またはパラメーターのデータにバインドされているデータ型とバッファーに影響を与える複数の記述子フィールドを設定します。	サポートされる サポートされる	ISO 92 ISO 92
SQLCopyDesc (3.0)	1 つの記述子ハンドルから別の記述子ハンドルに記述子情報をコピーします。	サポートされる	ISO 92
SQL 要求の準備			
SQLAllocStmt (1.0)	N/A	推奨されない (SQLAllocHandle で置き換え)	N/A
SQLPrepare (1.0)	後で実行するために SQL ステートメントを準備します。	サポートされる	ISO 92
SQLBindParameter (2.0)	SQL ステートメントのパラメーター用のストレージを割り当てます。	サポートされる 注: この関数は、X/Open 標準および ISO 標準にはあるが、ODBC 2.x に存在しなかった SQLBindParam に代わるものです。	ODBC
SQLGetCursorName (1.0) SQLSetCursorName (1.0)	ステートメント・ハンドルに関連したカーソル名を返します。 アクティブ・ステートメントでカーソル名を指定します。アプリケーションが SQLSetCursorName を呼び出さない場合、ドライバーは、SQL ステートメント処理の必要に応じてカーソル名を生成します。	サポートされる サポートされる	ISO 92 ISO 92
SQLParamOptions (1.0)	N/A	推奨されない (SQLSetStmtAttr で置き換え)	N/A
SQLSetParam (1.0)	N/A	推奨されない (SQLBindParameter で置き換え)	N/A
SQLSetScrollOptions (1.0)	カーソル動作を制御するオプションを設定します。	推奨されない (SQLGetInfo および SQLSetStmtAttr で置き換え)	ODBC

表 170. solidDB がサポートする ODBC 関数 (続き)

関数名/導入されたバージョン [1]	目的	ODBC の使用時の可用性	規格適合 [2]
要求のサブミット			
SQLExecute (1.0)	ステートメント内にいずれかのパラメーター・マーカーが存在する場合、そのパラメーター・マーカー変数の現行値を使用して準備済みステートメントを実行します。	サポートされる	ISO 92
SQLExecDirect (1.0)	ステートメント内にいずれかのパラメーターが存在する場合、そのパラメーター・マーカー変数の現行値を使用して準備可能ステートメントを実行します。SQLExecDirect は、一回限りの実行のために SQL ステートメントをサブミットする最も迅速な方法です。	サポートされる	ISO 92
SQLNativeSQL (1.0)	ドライバーによって変更された SQL ストリングを返します。SQLNativeSQL は、SQL ステートメントを実行しません。	インプリメントされない。solidDB は、この機能をサポートしません。	N/A
SQLDescribeParam (1.0)	ドライバーによって変換された SQL ステートメントのテキストを返します。この情報は、IPD のフィールドでも入手できます。	サポートされる	ODBC
SQLNumParams (1.0)	SQL ステートメント内のパラメーター数を返します。	サポートされる	ISO 92
SQLParamData (1.0)	SQLPutData と組み合わせて使用され、実行時にパラメーター・データを提供します。(長いデータ値の場合に便利です。)	サポートされる	ISO 92
SQLPutData (1.0)	アプリケーションは、ステートメント実行時に、パラメーターまたは列のデータをドライバーに送信できます。この関数を使用して、文字、バイナリー、またはデータ・ソース固有のデータ型 (例えば、SQL_LONGVARBINARY 型または SQL_LONGVARCHAR 型のパラメーター) の列に文字データ値またはバイナリー・データ値を部分単位で送信できます。	サポートされる	ISO 92
結果および結果に関する情報のリトリーブ			
SQLRowCount (1.0)	UPDATE、INSERT、または DELETE のステートメントによって影響を受ける行の数を返します。	サポートされる	ISO 92
SQLNumResultCols (1.0)	結果セット内の列数を返します。	サポートされる	ISO 92

表 170. solidDB がサポートする ODBC 関数 (続き)

関数名/導入されたバージョン [1]	目的	ODBC の使用時の可用性	規格適合 [2]
SQLDescribeCol (1.0)	結果セット内の 1 つの列に対する結果記述子 (列名、型、列サイズ、小数桁数、および NULL 可能性) を返します。この情報は、IRD のフィールドでも入手できます。 注: ドライバーは、現在、SQL_DESC_LABEL、 SQL_DESC_NAME、SQL_DESC_SCHEMA_NAME、SQL_DESC_CATALOG_NAME、SQL_DESC_BASE_COLUMN_NAME、 および SQL_DESC_BASE_TABLE_NAME の各属性に関して、 バイト数の代わりに文字数を返します。 これは、ODBC 標準により厳密に準拠しており、ADO、VB、OLE-DB、および ODBC の呼び出しを使用して正確に機能します。ただし、これは Microsoft Visual DataBase Project の障害の原因となることに注意してください。レコードの更新/挿入後、そのレコードが保存されず、「the table does not exist」というエラーが表示されます。	サポートされる	ISO 92
SQLColAttributes (1.0)	N/A	推奨されない (SQLColAttribute で置き換え)	N/A
SQLColAttribute (3.0)	結果セット内の列の属性を記述します。 **注:** ドライバーは、現在、SQL_DESC_LABEL、 SQL_DESC_NAME、SQL_DESC_SCHEMA_NAME、SQL_DESC_CATALOG_NAME、SQL_DESC_BASE_COLUMN_NAME、 および SQL_DESC_BASE_TABLE_NAME の各属性に関して、 バイト数の代わりに文字数を返します。 これは、ODBC 標準により厳密に準拠しており、ADO、VB、OLE-DB、および ODBC の呼び出しを使用して正確に機能します。ただし、これは Microsoft Visual DataBase Project の障害の原因となることに注意してください。レコードの更新/挿入後、そのレコードが保存されず、「the table does not exist」というエラーが表示されます。	サポートされる	ISO 92
SQLBindCol (1.0)	結果列用にストレージを割り当て、データ型を指定します。	サポートされる	ISO 92
SQLFetch (1.0)	複数の結果行を返します。その際、結果セットから次のデータ行セットをフェッチし、すべてのバインド済み列のデータを返します。	サポートされる	ISO 92
SQLExtendedFetch (2.0)	N/A	SQLFetchScroll で置き換え	N/A

表 170. solidDB がサポートする ODBC 関数 (続き)

関数名/導入されたバージョン [1]	目的	ODBC の使用時の可用性	規格適合 [2]
SQLFetchScroll (3.0)	スクロール可能な結果行を返します。その際、結果セットから指定されたデータ行セットをフェッチし、すべてのバインド済み列のデータを返します。ブロック・カーソルのサポートにより、アプリケーションは、単一フェッチで複数の行をアプリケーション・バッファーにフェッチできます。 ODBC 2.x ドライバーで作業する場合、ドライバー・マネージャーは、この関数を SQLExtendedFetch にマップします。	サポートされる **注:** 現在、solidDB ODBC ドライバーはブックマークをサポートしていないため、SQLFetchScroll で SQL_FETCH_BOOKMARK オプションをサポートすることはできません。	ISO 92
SQLGetData (1.0)	結果セットの 1 行の 1 列の一部または全部を返します。複数回呼び出して可変長データを部分単位でリトリーブすることができるため、長いデータ値の場合に便利です。	サポートされる	ISO 92
SQLSetPos (1.0)	フェッチしたデータ・ブロック内にカーソルを位置付け、アプリケーションが行セット内のデータのリフレッシュや、結果セット内のデータの更新または削除を行えるようにします。	サポートされる。以下のすべてのオプションと一緒にサポートされます。 SQL_POSITION、 SQL_DELETE、および SQL_UPDATE	ODBC
SQLBulkOperations (3.0)	バルク挿入、およびブックマークによる更新、削除、フェッチを含むバルク・ブックマーク操作を実行します。	solidDB は、SQL_ADD オプションを使用する場合に限ってサポートします。	ODBC
SQLMoreResults (1.0)	SELECT、UPDATE、INSERT、または DELETE のステートメントを含むステートメントで使用できる結果がまだあるかどうかを判別し、もしあれば、それらの結果の処理を初期化します。	サポートされない solidDB は、複数の結果をサポートしません。	ODBC
SQLGetDiagField (3.0)	追加の診断情報 (指定したハンドルに関連付けられた診断データ構造の単一フィールド) を返します。この情報には、エラー、警告、および状況情報が含まれます。	サポートされる	ISO 92
SQLGetDiagRec (3.0)	追加の診断情報 (診断データ構造の複数フィールド) を返します。1 回の呼び出しに対して 1 つの診断フィールドを返す SQLGetDiagField とは異なり、SQLGetDiagRec は、SQLSTATE、ネイティブ・エラー・コード、および診断メッセージ・テキストなど、診断レコードで一般的に使用されている複数のフィールドを返します。	サポートされる	ISO 92
SQLError (1.0)	N/A	推奨されない (SQLGetDiagRec で置き換え)	N/A

表 170. solidDB がサポートする ODBC 関数 (続き)

関数名/導入されたバージョン [1]	目的	ODBC の使用時の可用性	規格適合 [2]
データ・ソースのシステム表に関する情報の取得			
SQLColumnPrivileges (1.0)	指定された表の列とそれに関連した特権のリストを返します。ドライバーは、指定された StatementHandle に対する結果セットとして、情報を返します。この関数は、該当する SQL の実行を通してサポートされます。	サポートされる	ODBC
SQLColumns (1.0)	指定された表の列とそれに関連した特権のリストを返します。ドライバーは、指定された StatementHandle に対する結果セットとして、情報を返します。この関数は、該当する SQL の実行を通してサポートされます。	サポートされる	X/Open
SQLForeignKeys (1.0)	以下の 2 つのタイプのリストを返します。 • 指定された表内の外部キー (他の表の主キーを参照する、指定された表内の列) • 指定された表の主キーを参照する、他の表内の外部キー ドライバーは、指定されたステートメントに対する結果セットとして、各リストを返します。	サポートされる	ODBC
SQLPrimaryKeys (1.0)	表の主キーを構成する列名のリストを返します。ドライバーは、結果セットとして情報を返します。この関数では、単一の呼び出しで複数の表から主キーを返すことはサポートされていません。	サポートされる	ODBC
SQLProcedureColumns (1.0)	指定したプロシージャーの入出力パラメーターのリスト、および結果セットを構成する列を返します。ドライバーは、指定されたステートメントに対する結果セットとして、情報を返します。	サポートされる	ODBC
SQLProcedures (1.0)	特定のデータ・ソースに保管されたプロシージャー名のリストを返します。プロシージャーは、実行可能オブジェクト、または入出力パラメーターを使用して実行できる名前付きエンティティーを示す総称です。	サポートされる	ODBC

表 170. solidDB がサポートする ODBC 関数 (続き)

関数名/導入されたバージョン [1]	目的	ODBC の使用時の可用性	規格適合 [2]
SQLSpecialColumns (1.0)	指定した表内の列に関して、以下の情報を返します。 • 表内の行をユニークに識別する最適な列セット • トランザクションによって行内の値が更新された場合に自動的に更新される列	サポートされる	X/Open
SQLStatistics (1.0)	単一の表およびその表に関連した索引のリストに関する統計を返します。ドライバーは、結果セットとして情報を返します。	サポートされる	ISO 92
SQLTablePrivileges (1.0)	表のリストおよび各表に関連した特権のリストを返します。ドライバーは、指定されたステートメントに対する結果セットとして、情報を返します。	サポートされる	ODBC
SQLTables (1.0)	特定のデータ・ソースに保管された表名、カタログ名、またはスキーマ名、および表タイプのリストを返します。	サポートされる	X/Open
ステートメントの終了			
SQLFreeStmt (1.0)	ステートメント処理を終了し、保留中の結果を破棄し、またオプションでステートメント・ハンドルに関連付けられたすべてのリソースを解放します。	サポートされる 注: SQL_DROP オプション付きの SQLFreeStmt は、SQLFreeHandle で置き換えられています。	ISO 92
SQLCloseCursor (3.0)	ステートメントに対してオープンされていたカーソルをクローズし、保留中の結果を破棄します。	サポートされる	ISO 92
SQLCancel (1.0)	SQL ステートメントに対する処理を取り消します。	サポートされる	ISO 92
SQLEndTran (3.0)	接続に関連付けられたすべてのステートメントに対するトランザクションのコミットまたはロールバックを要求します。また、SQLEndTran は、環境に関連付けられたすべての接続に関して、コミット操作またはロールバック操作を実行するよう要求できます。	サポートされる	ISO 92
SQLTransact (1.0)	N/A	推奨されない (SQLEndTran で置き換え)	N/A
接続の終了			

表 170. solidDB がサポートする ODBC 関数 (続き)

関数名/導入されたバージョン [1]	目的	ODBC の使用時の可用性	規格適合 [2]
SQLDisconnect (1.0)	特定の接続ハンドルに関連付けられた接続を閉じます。	サポートされる	ISO 92
SQLFreeConnect (1.0)	N/A	推奨されない (SQLFreeHandle で置き換え)	N/A
SQLFreeEnv (1.0)	N/A	推奨されない (SQLFreeHandle で置き換え)	N/A
SQLFreeHandle (3.0)	特定の環境、接続、ステートメント、または記述子のハンドルに関連付けられたリソースを解放します。	サポートされる	ISO 92

[1] 導入されたバージョンとは、関数が最初に ODBC API に追加された時点のバージョンです。

[2] 規格適合レベルは、以下のいずれかにすることができます。

- ISO 92 (X/Open は ISO 92 の完全なスーパーセットであるため、X/Open バージョン 1 にもあります)
- X/Open (ODBC 3.x は X/Open バージョン 1 の完全なスーパーセットであるため、ODBC 3.x にもあります)
- ODBC (ISO 92 または X/Open のどちらにもありません)
- N/A (ODBC 3.x では推奨されません)

付録 B. solidDB ODBC ドライバー 3.5.1 属性のサポート

このトピックでは、solidDB ODBC ドライバー 3.5.1 の属性について説明します。

属性は、以下のカテゴリーにグループ化されています。

- 環境レベルの属性
- 接続レベルの属性
- ステートメント・レベルの属性
- 列レベルの属性

表 171. 001 環境レベル

属性	値 (オプション)	ドライバー・マネージャー	ドライバー単独	コメント
SQL_ATTR_CONNECTION_POOLING	SQL_CP_OFF SQL_CP_ONE_PER_DRIVER SQL_CP_ONE_PER_HENV	すべての値がサポートされる	すべての値がドライバーに適用されない	すべての値がODBC ドライバーに適用されず、ドライバー・マネージャーによって処理されます。したがって、この属性は、アプリケーションが ODBC DM にリンクされ、ドライバー自体ではシミュレートできない場合にサポートされます。
SQL_ATTR_CP_MATCH	SQL_CP_STRICT_MATCH SQL_CP_RELAXED_MATCH	すべての値がサポートされる	すべての値がドライバーに適用されない	すべての値がODBC ドライバーに適用されず、ドライバー・マネージャーによって処理されます。したがって、この属性は、アプリケーションが ODBC DM にリンクされ、ドライバー自体ではシミュレートできない場合にサポートされます。

表 171. 001 環境レベル (続き)

属性	値 (オプション)	ドライバー・マネージャー	ドライバー単独	コメント
SQL_ATTR_ ODBC_VERSION	SQL_OV_ODBC3 SQL_OV_ODBC2	サポートされる サポートされない	サポートされる サポートされない	ユーザーはバージョンを 2 に設定して取得できますが、動作は 3.0 以上に従って行われます。
SQL_ATTR_ OUTPUT_NTS	SQL_TRUE SQL_FALSE	サポートされる サポートされない	サポートされる サポートされない	

表 172. 002 接続レベル

属性	値 (オプション)	ドライバー・マネージャー	ドライバー単独	コメント
SQL_ATTR_ODBC_CURSORS	SQL_CUR_IF_NEEDED SQL_FETCH_PRIOR SQL_CUR_USE_ODBC SQL_CUR_ USE_DRIVER	すべての値がサポートされない	すべての値がサポートされない	
SQL_ATTR_ACCESS_MODE	SQL_MODE_ READ_ONLY SQL_MODE_ READ_WRITE	すべての値がサポートされない	すべての値がサポートされない	
SQL_ATTR_ASYNC_ENABLE	SQL_ASYNC_ ENABLE_OFF SQL_ASYNC_ ENABLE_ON	すべての値がサポートされない	すべての値がサポートされない	
SQL_ATTR_AUTO_IPD	SQL_TRUE SQL_FALSE	すべての値がサポートされない	すべての値がサポートされない	
SQL_ATTR_AUTOCOMMIT	SQL_ATTR_ AUTOCOMMIT_OFF SQL_ATTR_ AUTOCOMMIT_ON	すべての値がサポートされる	すべての値がサポートされる	

表 172. 002 接続レベル (続き)

属性	値 (オプション)	ドライバー・マネージャー	ドライバー単独	コメント
SQL_ATTR_ CONNECTION_ TIMEOUT	秒単位のタイムアウト値	サポートされる	サポートされる	
SQL_ATTR_ CURRENT_CATALOG	カタログ名	サポートされる	サポートされる	
SQL_ATTR_ LOGIN_TIMEOUT	秒単位のタイムアウト値	サポートされる	サポートされる	
SQL_ATTR_ METADATA_ID	SQL_TRUE SQL_FALSE	すべての値がサポートされない	すべての値がサポートされない	
SQL_ATTR_PACKET_SIZE	バイト単位のパケット・サイズ	要求サイズ	サポートされない	
SQL_ATTR_QUIET_MODE	NULL 設定	設定と取得が可能	サポートされない	
SQL_ATTR_TRACE	SQL_TRACE_OFF SQL_TRACE_ON	すべての値がサポートされる	すべての値がサポートされない	すべての値が、ドライバーではなく、DM によって処理されます。
SQL_ATTR_TRACEFILE	トレース・ファイル名を指すポインター	サポートされる	サポートされない	ドライバーではなく、DM によって処理されます。
SQL_ATTR_TRNSLATE_LIB	lib のポインター名	サポートされる	サポートされない	ドライバーではなく、DM によって処理されます。
SQL_ATTR_TXN_ISOLATION	SQL_TXN_ SERIALIZABLE SQL_TXN_READ_ UNCOMMITTED SQL_TXN_READ_ COMMITTED SQL_TXN_ REPEATABLE_ READ	以下を除くすべての値がサポートされる SQL_TXN_ READ_ UNCOMMITTED	以下を除くすべての値がサポートされる SQL_TXN_ READ_ UNCOMMITTED	solidDB サーバーは、 READ_ UNCOMMITTED 機能をサポートしません。

表 173. 03 ステートメント・レベル

属性	値 (オプション)	ドライバー・マネージャー	ドライバー単独	コメント
SQL_ATTR_ CONCURRENCY	SQL_CONCUR_ READ_ONLY SQL_CONCUR_LOCK SQL_CONCUR_ROWVER SQL_CONCUR_VALUES	すべての値がサポートされる	すべての値がサポートされる	SQL_CONCUR_READ_ONLY については、設定と取得がサポートされます。その他のすべての値については、設定はサポートされますが、取得では READ_ONLY が返されます。
SQL_ATTR_ CURSOR_TYPE	SQL_CURSOR_ FORWARD_ONLY SQL_CURSOR_ KEYSET_DRIVEN SQL_CURSOR_DYNAMIC SQL_CURSOR_STATIC	サポートされる 強制的に動的となる 強制的に動的となる 強制的に動的となる	サポートされる 強制的に動的となる 強制的に動的となる 強制的に動的となる	
SQL_ATTR_MAX_LENGTH	バイト単位の長さ	サポートされない	サポートされない	長さに関係なく、デフォルト (0) にのみ設定されます。
SQL_ATTR_MAX_ROWS	行の最大数	サポートされない	サポートされない	長さに関係なく、デフォルト (0) にのみ設定されます。
SQL_ATTR_ RETRIEVE_DATA	SQL_RD_OFF SQL_RD_ON	サポートされない サポートされる	サポートされない サポートされる	SQL_RD_ON にのみ設定されます。
SQL_ATTR_ USE_BOOKMARKS	SQL_UB_OFF SQL_UB_ON	すべての値がサポートされない	すべての値がサポートされない	
SQL_ATTR_ ROW_ARRAY_SIZE	SQLFetch または SQLFetchScroll の呼び出し後、行状況値の入った SQLUSMALLINT 値の配列を指す SQLUSMALLINT* 値	サポートされる	サポートされる	この配列のエレメントの数は、行セット内にある行の数と同じです。

表 173. 03 ステートメント・レベル (続き)

属性	値 (オプション)	ドライバー・マネージャー	ドライバー単独	コメント
SQL_ATTR_ROWS_FETCHED_PTR	SQLFetch または SQLFetchScroll の呼び出し後、フェッチした行の数を返すバッファーを指す SQLUINTEGER* 値	サポートされる	サポートされる	
SQL_ATTR_ROW_STATUS_PTR	SQLFetch または SQLFetchScroll の各呼び出しで返される行数を指定する SQLUINTEGER 値	サポートされる	サポートされる	
SQL_ROWSET_SIZE	返される行数	サポートされる	サポートされる	ODBC アプリケーションは、1 より大きい値を設定できます。
SQL_ASYNC_ENABLE	SQL_ASYNC_ENABLE_ON SQL_ASYNC_ENABLE_OFF	すべての値がサポートされない	すべての値がサポートされない	
SQL_BIND_TYPE	SQL_BIND_BY_COLUMN	サポートされない	サポートされない	
SQL_ATTR_KEYSET_SIZE	サイズ	サポートされない	サポートされない	サイズに関係なく、デフォルト (0) にのみ設定されます。
SQL_ATTR_NOSCAN	SQL_NOSCAN_OFF SQL_NOSCAN_ON	サポートされない サポートされない	サポートされない サポートされない	SQL_NOSCAN_OFF にのみ設定されます。
SQL_ATTR_SIMULATE_CURSOR	SQL_SC_NON_UNIQUE SQL_SC_TRY_UNIQUE SQL_SC_UNIQUE	すべての値がサポートされない	すべての値がサポートされない	すべての値が、solidDB ドライバーに関連しません。
SQL_ATTR_APP_PARAM_DESC	SQL_NULL_HDESC	サポートされない	サポートされない	
SQL_ATTR_APP_ROW_DESC	SQL_NULL_HDESC	サポートされない	サポートされない	

表 173. 03 ステートメント・レベル (続き)

属性	値 (オプション)	ドライバー・マネージャー	ドライバー単独	コメント
SQL_ATTR_ CURSOR_SCROLLABLE	SQL_SCROLLABLE SQL_NONSCROLLABLE	サポートされない サポートされない	サポートされない サポートされない	SQL_NONSCROLLABLE にのみ設定されます。
SQL_ATTR_ CURSOR_SENSITIVITY	SQL_UNSPECIFIED SQL_INSENSITIVE SQL_SENSITIVE	サポートされない サポートされない サポートされない	サポートされない サポートされない サポートされない	SQL_UNSPECIFIED にのみ設定されます。
SQL_ATTR_ROW_NUMBER	現在行数	サポートされる	サポートされる	ユーザーは行数を取得できます。読み取り専用プロパティーなので、設定はできません。
SQL_ATTR_ ENABLE_AUTO_IPD	SQL_TRUE SQL_FALSE	両方の値がサポートされない	両方の値がサポートされない	
SQL_ATTR_METADATA_ID	SQL_TRUE SQL_FALSE	両方の値がサポートされない	両方の値がサポートされない	
SQL_ATTR_PARAM_ BIND_OFFSET_PTR	SQL_DESC_DATA_PTR SQL_DESC_ INDICATOR_PTR SQL_DESC_ OCTET_LENGTH_PTR SQL_DESC_ BIND_OFFSET_PTR	すべての値がサポートされる	すべての値がサポートされる	
SQL_ATTR_PARAM_ OPERATION_PTR	無視されるパラメーターのリストを含む配列を指すポインター	サポートされる	サポートされる	

表 173. 03 ステートメント・レベル (続き)

属性	値 (オプション)	ドライバー・マネージャー	ドライバー単独	コメント
SQL_ATTR_PARAMS_PROCESSED_PTR	SQLExecute または SQLExecDirect を通して実行される SQL ステートメントによって処理されたパラメーターのセット数を返す符号なし整数ポインター	サポートされる	サポートされる	

表 174. 04 列属性

属性	値 (オプション)	ドライバー・マネージャー	ドライバー単独	コメント
SQL_DESC_BASE_COLUMN_NAME		サポートされる	サポートされる	
SQL_DESC_BASE_TABLE_NAME		サポートされる	サポートされる	
SQL_DESC_DISPLAY_SIZE		サポートされる	サポートされる	
SQL_DESC_NAME				
SQL_DESC_NULLABLE		サポートされる	サポートされる	
SQL_DESC_OCTET_LENGTH		サポートされる	サポートされる	
SQL_DESC_PRECISION		サポートされる	サポートされる	
SQL_DESC_SCALE		サポートされる	サポートされる	
SQL_DESC_UPDATABLE		サポートされる	サポートされる	
SQL_DESC_FIXED_PREC_SCALE		サポートされる	サポートされる	
SQL_DESC_TABLE_NAME		サポートされる	サポートされる	
SQL_DESC_TYPE		サポートされる	サポートされる	
SQL_DESC_UNNAMED		サポートされる	サポートされる	
SQL_DESC_SCHEMA_NAME		サポートされる	サポートされる	
SQL_DESC_LOCAL_TYPE_NAME		サポートされる	サポートされる	
SQL_DESC_LABEL		サポートされる	サポートされる	
SQL_DESC_TYPE_NAME		サポートされる	サポートされる	

表 174. 04 列属性 (続き)

属性	値 (オプション)	ドライバー・マネージャー	ドライバー単独	コメント
SQL_DESC_AUTO_UNIQUE_VALUE		サポートされる	サポートされる	
SQL_DESC_CONCISE_TYPE		サポートされる	サポートされる	
SQL_DESC_LITERAL_PREFIX		サポートされる	サポートされる	
SQL_DESC_UNSIGNED		サポートされる	サポートされる	
SQL_DESC_LITERAL_PREFIX		サポートされる	サポートされる	
SQL_DESC_UNSIGNED		サポートされる	サポートされる	
SQL_DESC_LITERAL_SUFFIX		サポートされる	サポートされる	
SQL_DESC_CATALOG_NAME		サポートされる	サポートされる	
SQL_DESC_COUNT		サポートされる	サポートされる	
SQL_DESC_SEARCHABLE		サポートされる	サポートされる	
SQL_DESC_LENGTH		サポートされる	サポートされる	
SQL_DESC_CASE_SENSITIVE		サポートされる	サポートされる	
SQL_DESC_NUM_PREX_RADIX		サポートされる	サポートされる	

付録 C. エラー・コード

このトピックでは、SQLGetDiagRec 関数用にドライバーから返される可能性のある SQLSTATE 値を記載したエラー・コード表を示します。

注: SQLGetDiagRec および SQLGetDiagField は、X/Open (現 Open Group) の「*Data Management: Structured Query Language (SQL), Version 2 (3/95)*」に準拠する SQLSTATE 値を返します。

エラー・コード表の規約

表 175. エラー・コード・クラス値

クラス値	意味
01	警告を表し、SQL_SUCCESS_WITH_INFO の戻りコードを含みます。
01、 07、 08、 21、 22、 25、 28、 34、 3C、 3D、 3F、 40、 42、 44、 HY	SQL_ERROR の戻り値を含むエラーを表します。
IM	ODBC から派生した警告およびエラーを表します。

注: 一般に、関数は、実行が成功すると SQL_SUCCESS 値を返します。しかし、場合によっては、関数は SQLSTATE 00000 を返すこともあり、同様に実行が成功したことを表します。

SQLSTATE コード

表 176. SQLSTATE コード

SQLSTATE	エラー	該当エラーを返す可能性のある関数
01000	通常の警告	以下を除くすべての ODBC 関数 SQLGetDiagField SQLGetDiagRec
01001	カーソル操作の競合	SQLExecDirect SQLExecute SQLParamData SQLSetPos
01002	切断エラー	SQLDisconnect
01003	set 関数内で NULL 値を除去	SQLExecDirect SQLExecute SQLParamData

表 176. SQLSTATE コード (続き)

SQLSTATE	エラー	該当エラーを返す可能性のある関数
01004	ストリング・データ、右側の切り捨て	• SQLColAttribute • SQLDataSources • SQLDescribeCol • SQLDriverConnect • SQLDrivers • SQLExecDirect • SQLExecute • SQLExtendedFetch • SQLFetch • SQLFetchScroll • SQLGetConnectAttr • SQLGetCursorName • SQLGetData • SQLGetDescField • SQLGetDescRec • SQLGetEnvAttr • SQLGetInfo • SQLGetStmtAttr • SQLParamData • SQLPutData • SQLSetCursorName
01006	特権が取り消されない	SQLExecDirect SQLExecute SQLParamData
01007	特権が付与されない	SQLExecDirect SQLExecute SQLParamData
01S00	接続ストリング属性が無効	SQLDriverConnect SQLSetPos
01S01	行内のエラー	SQLExtendedFetch
01S02	オプション値の変更	• SQLConnect • SQLDriverConnect • SQLExecDirect • SQLExecute • SQLParamData • SQLPrepare • SQLSetConnectAttr • SQLSetDescField • SQLSetEnvAttr • SQLSetStmtAttr
01S06	結果セットが最初の行セットを返す前にフェッチを試行	SQLExtendedFetch SQLFetchScroll

表 176. SQLSTATE コード (続き)

SQLSTATE	エラー	該当エラーを返す可能性のある関数
01S07	小数部の切り捨て	SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLFetchScroll SQLGetData SQLParamData SQLSetPos
01S08	ファイル DSN の保存エラー	SQCriverConnect
01S09	無効なキーワード	SQLDriverConnect
07001	パラメーター数の誤り	SQLExecDirect SQLExecute
07002	COUNT フィールドが不正	SQLExecDirect SQLExecute SQLParamData
07005	準備済みステートメントがカーソル指定ではない	SQLColAttribute SQLDescribeCol
07006	制限付きデータ型属性違反	SQLBindCol SQLBindParameter SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLFetchScroll SQLGetData SQLParamData SQLPutData

表 176. SQLSTATE コード (続き)

SQLSTATE	エラー	該当エラーを返す可能性のある関数
07009	無効な記述子索引	SQLBindCol SQLBindParameter SQLColAttribute SQLDescribeCol SQLDesribeParam SQLFetch SQLFetchScroll SQLGetData SQLGetDescField SQLParamData SQLSetDescField SQLSetDescRec SQLSetPos
07S01	デフォルト・パラメーターの無効な使用	SQLExecDirect SQLExecute SQLParamData SQLPutData
08001	クライアントが接続を確立できない	SQLConnect SQLDriverConnect
08002	接続名は使用中	SQLConnect SQLDriverConnect SQLSetConnectAttr
08003	接続が存在しない	SQLAllocHandle SQLDisconnect SQLEndTran SQLGetConnectAttr SQLGetInfo SQLSetConnectAttr
08004	サーバーが接続を拒否	SQLConnect SQLDriverConnect
08007	トランザクション中の接続障害	SQLEndTran

表 176. SQLSTATE コード (続き)

SQLSTATE	エラー	該当エラーを返す可能性のある関数
08S01	通信リンク障害	• SQLColumnPrivileges • SQLColumns • SQLConnect • SQLConnect • SQLCopyDesc • SQLDescribeCol • SQLDescribeParam • SQLDriverConnect • SQLExecDirect • SQLExecute • SQLExtendedFetch • SQLFetch • SQLFetchScroll • SQLForeignKeys • SQLGetConnectAttr • SQLGetData • SQLGetDescField • SQLGetDescRec • SQLGetFunctions • SQLGetInfo • SQLGetTypeInfo • SQLMoreResults • SQLNumParams • SQLNumResultCols • SQLParamData • SQLPrepare • SQLPrimaryKeys • SQLProcedureColumns • SQLProcedures • SQLPutData • SQLSetConnectAttr • SQLSetDescField • SQLSetDescRec • SQLSetEnvAttr • SQLSetStmtAttr • SQLSpecialColumns • SQLStatistics • SQLTablePrivileges • SQLTables
21S01	挿入値リストと列リストが一致しない	SQLExecDirect SQLPrepare
21S02	派生表の次数と列リストが一致しない	SQLExecDirect SQLExecute SQLParamData SQLPrepare SQLSetPos

表 176. SQLSTATE コード (続き)

SQLSTATE	エラー	該当エラーを返す可能性のある関数
22001	ストリング・データ、右側の切り捨て	SQLExecDirect SQLExecute SQLFetch SQLFetchScroll SQLParamData SQLPutData SQLSetDescField SQLSetPos
22002	標識変数が必要であるが、提供されていない	SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLFetchScroll SQLGetData SQLParamData
22003	範囲外の数値	SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLFetchScroll SQLGetData SQLGetInfo SQLParamData SQLPutData SQLSetPos
22007	無効な日時フォーマット	SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLFetchScroll SQLGetData SQLParamData SQLPutData SQLSetPos

表 176. SQLSTATE コード (続き)

SQLSTATE	エラー	該当エラーを返す可能性のある関数
22008	日時フィールドのオーバーフロー	SQLExecDirect SQLExecute SQLParamData SQLPutData
22012	ゼロ除算	SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLFetchScroll SQLGetData SQLParamData SQLPutData
22015	インターバル・フィールドのオーバーフロー	SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLFetchScroll SQLGetData SQLParamData SQLPutData SQLSetPos
22018	キャスト指定に関する無効文字値	SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLFetchScroll SQLGetData SQLParamData SQLPutData SQLSetPos
22019	無効なエスケープ文字	SQLExecDirect SQLExecute SQLPrepare

表 176. SQLSTATE コード (続き)

SQLSTATE	エラー	該当エラーを返す可能性のある関数
22025	無効なエスケープ・シーケンス	SQLExecDirect SQLExecute SQLPrepare
22026	ストリング・データ、長さの不一致	SQLParamData
23000	保全性制約違反	SQLExecDirect SQLExecute SQLParamData SQLSetPos
24000	無効なカーソル状態	SQLCloseCursor SQLColumnPrivileges SQLColumns SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLFetchScroll SQLForeignKeys SQLGetData SQLGetStmtAttr SQLGetTypeInfo SQLPrepare SQLPrimaryKeys SQLProcedureColumns SQLProcedures SQLConnectAttr SQLSetCursorName SQLSetPos SQLSpecialColumns SQLStatistics SQLTablePrivileges SQLTables
25000	無効なトランザクション状態	SQLDisconnect
25S01	トランザクション状態	SQLEndTran
25S02	トランザクションが引き続きアクティブ	SQLEndTran

表 176. SQLSTATE コード (続き)

SQLSTATE	エラー	該当エラーを返す可能性のある関数
25S03	トランザクションがロールバックされている	SQLEndTran
28000	無効な許可指定	SQLConnect SQLDriverConnect
34000	無効なカーソル名	SQLExecDirect SQLPrepare SQLSetCursorName
3C000	重複カーソル名	SQLSetCursorName
3D000	無効なカタログ名	SQLExecDirect SQLPrepare SQLSetConnectAttr
3F000	無効なスキーマ名	SQLExecDirect SQLPrepare
40001	逐次化障害	SQLColumnPrivileges SQLColumns SQLEndTran SQLExecDirect SQLExecute SQLFetch SQLFetchScroll SQLForeignKeys SQLGetTypeInfo SQLMoreResults SQLParamData SQLPrimaryKeys SQLProcedureColumns SQLProcedures SQLSetPos SQLSpecialColumns SQLStatistics SQLTablePrivileges SQLTables
40002	保全性制約違反	SQLEndTran

表 176. SQLSTATE コード (続き)

SQLSTATE	エラー	該当エラーを返す可能性のある関数
40003	ステートメントの完了が不明	SQLColumnPrivileges SQLColumns SQLExecDirect SQLExecute SQLFetch SQLFetchScroll SQLGetTypeInfo SQLForeignKeys SQLMoreResults SQLPrimaryKeys SQLProcedureColumns SQLProcedures SQLParamData SQLSetPos SQLSpecialColumns SQLStatistics SQLTables
42000	構文エラーまたはアクセス違反	SQLExecDirect SQLExecute SQLParamData SQLPrepare SQLSetPos
42S01	基本表またはビューが既に存在する	SQLExecDirect SQLPrepare
42S02	基本表またはビューが検出されない	SQLExecDirect SQLPrepare
42S11	索引が既に存在する	SQLExecDirect SQLPrepare
42S12	索引が検出されない	SQLExecDirect SQLPrepare
42S21	列が既に存在する	SQLExecDirect SQLPrepare
42S22	列が検出されない	SQLExecDirect SQLPrepare

表 176. SQLSTATE コード (続き)

SQLSTATE	エラー	該当エラーを返す可能性のある関数
44000	WITH CHECK OPTION 違反	SQLExecDirect SQLExecute SQLParamData
HY000	一般エラー	以下を除くすべての ODBC 関数 SQLGetDiagField SQLGetDiagRec
HY001	メモリー割り振りエラー	以下を除くすべての ODBC 関数 SQLGetDiagField SQLGetDiagRec
HY003	無効なアプリケーション・バッファー型	SQLBindCol SQLBindParameter SQLGetData
HY004	無効な SQL データ型	SQLBindParameter SQLGetTypeInfo
HY007	関連付けられたステートメントが準備されていない	SQLCopyDesc SQLGetDescField SQLGetDescRec

表 176. SQLSTATE コード (続き)

SQLSTATE	エラー	該当エラーを返す可能性のある関数
HY008	操作の取り消し	非同期に処理可能なすべての ODBC 関数 SQLColAttribute SQLColumnPrivileges SQLColumns SQLDescribeCol SQLDescribeParam SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLFetchScroll SQLForeignKeys SQLGetData SQLGetTypeInfo SQLMoreResults SQLNumParams SQLNumResultCols SQLParamData SQLPrepare SQLPrimaryKeys SQLProcedureColumns SQLProcedures SQLPutData SQLSetPos SQLSpecialColumns SQLStatistics SQLTablePrivileges SQLTables

表 176. SQLSTATE コード (続き)

SQLSTATE	エラー	該当エラーを返す可能性のある関数
HY009	NULL ポインターの無効な使用	SQLAllocHandle SQLBindParameter SQLColumnPrivileges SQLColumns SQLExecDirect SQLForeignKeys SQLGetCursorName SQLGetData SQLGetFunctions SQLPrepare SQLPrimaryKeys SQLProcedureColumns SQLProcedures SQLPutData SQLSetConnectAttr SQLSetCursorName SQLSetEnvAttr SQLSetStmtAttr SQLSpecialColumns SQLStatistics SQLTablePrivileges SQLTables

表 176. SQLSTATE コード (続き)

SQLSTATE	エラー	該当エラーを返す可能性のある関数
HY010	関数のシーケンス・エラー	• SQLAllocHandle • SQLBindCol • SQLBindParameter • SQLCloseCursor • SQLColAttribute • SQLColumnPrivileges • SQLColumns • SQLCopyDesc • SQLDescribeCol • SQLDescribeParam • SQLDisconnect • SQLEndTran • SQLExecDirect • SQLExecute • SQLExtendedFetch • SQLFetch • SQLFetchScroll • SQLForeignKeys • SQLFreeHandle • SQLFreeStmt • SQLGetConnectAttr • SQLGetCursorName • SQLGetData • SQLGetDescField • SQLGetDescRec • SQLGetFunctions • SQLGetStmtAttr • SQLGetTypeInfo • SQLMoreResults • SQLNumParams • SQLNumResultCols • SQLParamData • SQLPrepare • SQLPrimaryKeys • SQLProcedureColumns • SQLProcedures • SQLPutData • SQLRowCount • SQLSetConnectAttr • SQLSetCursorName • SQLSetDescField • SQLSetEnvAttr • SQLSetDescRec • SQLSetPos • SQLSetStmtAttr • SQLSpecialColumns • SQLStatistics • SQLTablePrivileges • SQLTables

表 176. SQLSTATE コード (続き)

SQLSTATE	エラー	該当エラーを返す可能性のある関数
HY011	現在、属性を設定できない	SQLParamData SQLSetConnectAttr SQLSetPos SQLSetStmtAttr
HY012	無効なトランザクション命令コード	SQLEndTran
HY013	メモリー管理エラー	以下を除くすべての ODBC 関数 SQLGetDiagField SQLGetDiagRec
HY014	ハンドル数の制限を超過	SQLAllocHandle
HY015	使用可能なカーソル名が存在しない	SQLGetCursorName
HY016	インプリメンテーション行記述子を変更できない	SQLCopyDesc SQLSetDescField SQLSetDescRec
HY017	自動的に割り振られた記述子ハンドルの無効な使用	SQLFreeHandle SQLSetStmtAttr
HY018	サーバーが取り消し要求を拒否	SQLCancel
HY019	文字以外およびバイナリー以外のデータが断片で送信された	SQLPutData
HY020	NULL 値の連結を試行した	SQLPutData
HY021	不整合な記述子情報	SQLBindParameter SQLCopyDesc SQLGetDescField SQLSetDescField SQLSetDescRec
HY024	無効な属性値	SQLSetConnectAttr SQLSetEnvAttr SQLSetStmtAttr

表 176. SQLSTATE コード (続き)

SQLSTATE	エラー	該当エラーを返す可能性のある関数
HY090	ストリングまたはバッファーの長さが無効	• SQLBindCol • SQLBindParameter • SQLBrowseConnect • SQLColAttribute • SQLColumnPrivileges • SQLColumns • SQLConnect • SQLDataSources • SQLDescribeCol • SQLDriverConnect • SQLDrivers • SQLExecDirect • SQLExecute • SQLFetch • SQLFetchScroll • SQLForeignKeys • SQLGetConnectAttr • SQLGetCursorName • SQLGetData • SQLGetDescField • SQLGetInfo • SQLGetStmtAttr • SQLParamData • SQLPrepare • SQLPrimaryKeys • SQLProcedureColumns • SQLProcedures • SQLPutData • SQLSetConnectAttr • SQLSetCursorName • SQLSetDescField • SQLSetDescRec • SQLSetEnvAttr • SQLSetStmtAttr • SQLSetPos • SQLSpecialColumns • SQLTablePrivileges • SQLStatistics • SQLTables
HY091	無効な記述子フィールド ID	SQLColAttribute SQLGetDescField SQLSetDescField

表 176. SQLSTATE コード (続き)

SQLSTATE	エラー	該当エラーを返す可能性のある関数
HY092	無効な属性/オプション ID	SQLAllocHandle SQLCopyDesc SQLDriverConnect SQLEndTran SQLFreeStmt SQLGetConnectAttr SQLGetEnvAttr SQLGetStmtAttr SQLParamData SQLSetConnectAttr SQLSetDescField SQLSetEnvAttr SQLSetPos SQLSetStmtAttr
HY095	関数型が範囲外	SQLGetFunctions
HY096	無効な情報タイプ	SQLGetInfo
HY097	列型が範囲外	SQLSpecialColumns
HY098	有効範囲タイプが範囲外	SQLSpecialColumns
HY099	NULL 可能タイプが範囲外	SQLSpecialColumns
HY100	ユニーク性オプション・タイプが範囲外	SQLStatistics
HY101	確度オプション・タイプが範囲外	SQLStatistics
HY103	無効なリトリーブ・コード	SQLDataSources SQLDrivers
HY104	無効な精度または位取りの値	SQLBindParameter
HY105	無効なパラメーター型	SQLBindParameter SQLExecDirect SQLExecute SQLParamData SQLSetDescField
HY106	フェッチ・タイプが範囲外	SQLExtendedFetch SQLFetchScroll

表 176. SQLSTATE コード (続き)

SQLSTATE	エラー	該当エラーを返す可能性のある関数
HY107	行値が範囲外	SQLExtendedFetch SQLFetch SQLFetchScroll SQLSetPos
HY109	無効なカーソル位置	SQLExecDirect SQLExecute SQLGetData SQLGetStmtAttr SQLParamData SQLSetPos
HY110	無効なドライバー完了	SQLDriverConnect
HY111	無効なブックマーク値	SQLExtendedFetch SQLFetchScroll

表 176. SQLSTATE コード (続き)

SQLSTATE	エラー	該当エラーを返す可能性のある関数
HYC00	オプション機能がインプリメントされない	• SQLBindCol • SQLBindParameter • SQLColAttribute • SQLColumnPrivileges • SQLColumns • SQLDriverConnect • SQLEndTran • SQLConnect • SQLExecDirect • SQLExecute • SQLExtendedFetch • SQLFetch • SQLFetchScroll • SQLForeignKeys • SQLGetConnectAttr • SQLGetData • SQLGetEnvAttr • SQLSetPos • SQLGetInfo • SQLGetStmtAttr • SQLGetTypeInfo • SQLParamData • SQLPrepare • SQLPrimaryKeys • SQLProcedureColumns • SQLProcedures • SQLSetConnectAttr • SQLSetEnvAttr • SQLSetStmtAttr • SQLSpecialColumns • SQLStatistics • SQLTablePrivileges • SQLTables

表 176. SQLSTATE コード (続き)

SQLSTATE	エラー	該当エラーを返す可能性のある関数
HYT00	タイムアウトの期限切れ	SQLBrowseConnect SQLColumnPrivileges SQLColumns SQLConnect SQLDriverConnect SQLExecDirect SQLExecute SQLExtendedFetch SQLForeignKeys SQLGetTypeInfo SQLParamData SQLPrepare SQLPrimaryKeys SQLProcedureColumns SQLProcedures SQLSetPos SQLSpecialColumns SQLStatistics SQLTablePrivileges SQLTables
HYT01	接続タイムアウトの期限切れ	以下を除くすべての ODBC 関数 SQLDrivers SQLDataSources SQLGetEnvAttr SQLSetEnvAttr
IM001	ドライバーはこの関数をサポートしない	以下を除くすべての ODBC 関数 SQLAllocHandle SQLDataSources SQLDrivers SQLFreeHandle SQLGetFunctions
IM002	データ・ソース名が検出されず、デフォルト・ドライバーが指定されていない	SQLConnect SQLDriverConnect
IM003	指定されたドライバーをロードできなかった	SQLConnect

表 176. SQLSTATE コード (続き)

SQLSTATE	エラー	該当エラーを返す可能性のある関数
IM004	SQL_HANDLE_ENV に対するドライバーの SQLAllocHandle が失敗	SQLDriverConnec SQLConnect SQLDriverConnect
IM005	SQL_HANDLE_DBC に対するドライバーの SQLAllocHandle が失敗	SQLConnect SQLDriverConnect
IM006	ドライバーの SQLSetConnectAttr が失敗	SQLConnect SQLDriverConnect
IM007	データ・ソースまたはドライバーが指定されていない。ダイアログが禁止される	SQLDriverConnect
IM008	ダイアログの失敗	SQLDriverConnect
IM009	変換 DLL をロードできない	SQLConnect SQLDriverConnect SQLSetConnectAttr
IM010	データ・ソース名が長すぎる	SQLConnect SQLDriverConnect
IM011	ドライバー名が長すぎる	SQLDriverConnect
IM012	DRIVER キーワードの構文エラー	SQLDriverConnect
IM013	トレース・ファイル・エラー	すべての ODBC 関数
IM014	無効なファイル DSN 名	SQLDriverConnect
IM015	ファイル・データ・ソースが破損	SQLDriverConnect

付録 D. ODBC に関する SQL 文法の最小要件

このセクションでは、ODBC ドライバーがサポートしなければならない SQL-92 エントリー・レベル構文の最小サブセットについて説明します。この構文を使用するアプリケーションは、任意の ODBC 準拠ドライバーでサポートされます。

アプリケーションから SQL_SQL_CONFORMANCE を指定して SQLGetInfo を呼び出すことにより、このセクションに記載されていない SQL-92 の追加機能がサポートされているかどうか判断できます。

注: ドライバーが読み取り専用データ・ソースのみをサポートする場合、データの変更に適用される SQL 構文は、このドライバーには適用されません。データ・ソースが読み取り専用かどうか判断するには、アプリケーションで SQL_DATA_SOURCE_READ_ONLY 情報タイプを指定して SQLGetInfo を呼び出す必要があります。

SQL ステートメント

このセクションでは、SQL ステートメントとエレメントのサブセットについて説明します。

```
create-table-statement ::=
        CREATE TABLE base_table_name
        (column_identifier data_type [, column_identifier data_type]...)
```

重要: アプリケーションは、*create_table_statement* 内の *data_type* として、SQLGetTypeInfo から返される結果セットの TYPE_NAME 列のデータ型を要求します。

```
delete_statement_searched ::=
        DELETE FROM table_name [WHERE search_condition]
drop_table_statement ::=
        DROP TABLE base_table_name
select_statement ::=
        SELECT [ALL | DISTINCT] select_list
        FROM table_reference_list
        [WHERE search_condition]
        [order_by_clause]
statement ::= create_table_statement |
        delete_statement_searched |
        drop_table_statement |
        insert_stetement |
        select_statement |
        update_statement_searched
Update_statement_searched ::=
   UPDATE table_name
   SET column_identifier = {expression |
         NULL}
[, column_identifier = {expression |
         NULL}]...
[WHERE search_condition]
```

SQL ステートメントのエレメント

```
base_table_identifier ::= user_defined_name
base_table_name ::= base_table_identifier
boolean_factor ::= [NOT] boolean_primary
boolean_primary ::= predicate | ( search_condition )
boolean_term ::= boolean_factor [AND boolean_term]
character_string_literal :: = "{character}..."
(character は、ドライバー/データ・ソースの
文字セット内の任意の文字です。
character_string_literal 内に単一リテラル引用文字 (') を含めるには、
リテラル引用文字を 2 つ使用します [""]。)
column_identifier ::= user_defined_name
column_name ::= [table_name.]column_identifier
comparison_operator ::= < | > | <= | >= | = | <>
comparison_predicate ::= expression comparison_operator expression
data_type ::= character_string_type
(character_string_type は、SQLGetTypeInfo から返される結果セット内の
「DATA_TYPE」列が SQL_CHAR または SQL_VARCHAR のどちらかである、
任意のデータ型です。)
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
dynamic_parameter ::= ?
expression ::= term | expression {+|-} term
factor ::= [+|-]primary
insert_value ::= dynamic_parameter | literal | NULL | USER
letter ::= lower_case_letter | upper_case_letter
literal ::= character_string_literal
lower_case_letter ::= a | b | c | d | e | f | g |
  h | i | j | k | l | m | n | o | p | q | r | s |
  t | u | v | w | x | y | z
order_by_clause ::= ORDER BY sort_specification [, sort_specification]...
primary ::= column_name | dynamic_parameter | literal | ( expression )
search_condition ::= boolean_term [OR search_condition]
select_list ::= * | select_sublist [, select_sublist]...
(select_list は、パラメーターを含むことはできません。)
select_sublist ::= expression
sort_specification ::= {unsigned_integer | column_name } [ASC | DESC]
table_identifier ::= user_defined_name
table_name ::= table_identifier
table_reference ::= table_name
table_reference ::= table_name [,table_reference]...
term ::= factor | term {*|/} factor
unsigned_integer ::= {digit}
upper_case_letter ::=  A | B | C | D | E | F | G |
   H | I | J | K | L | M | N | O | P | Q | R | S |
   T | U | V | W | X | Y | Z
user_defined_name ::= letter[ digit | letter| _ ]...
```

制御ステートメント (論理条件)

このトピックでは、solidDB データベース・プロシージャーで使用可能な制御ステートメントのサマリーを記載します。

これらの制御ステートメントについて詳しくは、「*solidDB SQL ガイド*」でストアード・プロシージャーの説明を参照してください。

表 177. 制御ステートメント

制御ステートメント	説明
`set variable = expression`	変数に値を割り当てます。この値には、リテラル値 (例えば、10 や 'text') または別の変数のいずれかを使用できます。パラメーターは、通常の変数と解釈されます。
`variable := expression`	変数に値を割り当てるための代替構文。
`boolean_expr`	「true」または「false」として評価されるブール式。この式は、=、>、< などの比較演算子、および AND、OR、NOT などの論理演算子を含むことができます。
`statement_list`	ブール式の結果に従って実行される有効なプロシージャー・ステートメント。

`while`
`  boolean_expr`
`loop`
`  statement_list`
`end loop`
	式が true の間ループします。WHILE ループ内の有効な括弧の使用例については、「*solidDB SQL ガイド*」のストアード・プロシージャーの説明を参照してください。

`leave`	最も内側の while ループを抜け、キーワード end loop の後の次のステートメントからプロシージャーの実行を継続します。

`if`
`  boolean_expr`
`then`
`  statement_list1`
`else`
`  statement_list2`
`end if`
	boolean_expr が true の場合、*statement_list1* を実行します。その他の場合には、*statement_list2* を実行します。IF ステートメント内の有効な括弧の使用例については、「*solidDB SQL ガイド*」のストアード・プロシージャーの説明を参照してください。

`if`
`  boolean_expr1`
`then`
`  statement_list1`
`elseif`
`  boolean_expr2`
`then`
`  statement_list2`
`end if`
	boolean_expr1 が true の場合、*statement_list1* を実行します。*boolean_expr2* が true の場合、*statement_list2* を実行します。このステートメントは、オプションで複数の elseif ステートメントを含むことができ、また else ステートメントを 1 つ含むことができます。IF ステートメント内の有効な括弧の使用例については、「*solidDB SQL ガイド*」のストアード・プロシージャーの説明を参照してください。

`return`	出力パラメーターの現行値を返し、プロシージャーを終了します。プロシージャーに *return row* ステートメントが含まれる場合、*return* は *return norow* のように動作します。

表 177. 制御ステートメント (続き)

制御ステートメント	説明
`return sqlerror of cursor_name`	カーソルに関連付けられた sqlerror を返し、プロシージャーを終了します。
`return row`	出力パラメーターの現行値を返し、プロシージャーの実行を継続します。return row はプロシージャーを終了せず、呼び出し元に制御を返しません。
`return norow`	セットの終了を返し、プロシージャーを終了します。

データ型のサポート

ODBC ドライバーは、少なくとも SQL_CHAR または SQL_VARCHAR のどちらかをサポートする必要があります。

その他のデータ型のサポートは、ドライバーまたはデータ・ソースの SQL-92 適合レベルによって決まります。ドライバーまたはデータ・ソースの SQL-92 適合レベルを判断するには、アプリケーションから SQLGetTypeInfo を呼び出す必要があります。

パラメーター・データ型

このトピックでは、パラメーターのデータ型を判断する方法、およびパラメーター・マーカーのサポートについて説明します。

SQLBindParameter で指定する各パラメーターは SQL データ型を使用して定義されるにもかかわらず、SQL ステートメント内のパラメーターは組み込みのデータ型を持ちません。したがって、ステートメント内の別のオペランドからパラメーター・マーカーのデータ型を推論できる場合のみ、SQL ステートメントにパラメーター・マーカーを組み込むことができます。例えば、? + COLUMN1 のような算術式では、パラメーターのデータ型は、COLUMN1 で表される名前付き列のデータ型から推論できます。アプリケーションは、データ型を判断できない場合、パラメーター・マーカーを使用できません。

以下の表では、いくつかのタイプのパラメーターについて、SQL-92 標準に従ってデータ型を判断する方法を説明します。パラメーター型を推論するための総合的な情報については、SQL-92 仕様を参照してください。

表 178. いくつかのタイプのパラメーターについてデータ型を判断する方法

パラメーターの位置	想定データ型
2 進算術計算演算子または比較演算子の一方のオペランド	他方のオペランドと同じ

表 178. いくつかのタイプのパラメーターについてデータ型を判断する方法 (続き)

パラメーターの位置	想定データ型
BETWEEN 節の第 1 オペランド	第 2 オペランドと同じ
BETWEEN 節の第 2 または第 3 オペランド	第 1 オペランドと同じ
IN で使用される式	最初の値または副照会の結果列と同じ
IN で使用される値	式と同じ、または式にパラメーター・マーカーが存在する場合には最初の値と同じ
LIKE で使用されるパターン値	VARCHAR
UPDATE で使用される更新値	更新列と同じ

パラメーター・マーカー

SQL-92 仕様に従うと、アプリケーションは、以下の位置にパラメーター・マーカーを配置することはできません。

- SELECT リスト内
- *comparison_predicate* の両方の *expression* として
- 2 項演算子の両方のオペランドとして
- BETWEEN 演算の第 1 および第 2 オペランドの両方として
- BETWEEN 演算の第 1 および第 3 オペランドの両方として
- IN 演算の式および最初の値の両方として
- 単項の + または - 演算のオペランドとして
- *set-function-reference* の引数として

総合的なリストと詳細については、SQL-92 仕様を参照してください。

ODBC のリテラル

このセクションでは、ドライバー作成者が、文字ストリング型から数値型またはインターバル型への変換を行う際、または数値型またはインターバル型から文字ストリング型への変換を行う際に役立つ情報を記載します。

インターバル・リテラル構文

以下の構文は、ODBC のインターバル・リテラルで使用されます。

```
interval_literal ::= INTERVAL [+|_] interval_string interval_qualifier
interval_string ::= quote { year_month_literal
     | day_time_literal } quote
year_month_literal ::= years_value | [years_value] months_value
day_time_literal ::= day_time_interval | time_interval
day_time_interval ::= days_value [hours_value
     [:minutes_value[:seconds_value]]]
time_interval ::= hours_value [:minutes_value [:seconds_value ] ]
     | minutes_value [:seconds_value ]
     | seconds_value
```

```
years_value ::= datetime_value
months_value ::= datetime_value
days_value ::= datetime_value
hours_value ::= datetime_value
minutes_value ::= datetime_value
seconds_value ::= seconds_integer_value [.[seconds_fraction] ]
seconds_integer_value ::= unsigned_integer
seconds_fraction ::= unsigned_integer
datetime_value ::= unsigned_integer
interval_qualifier ::= start_field TO end_field
     | single_datetime_field
start_field ::= non_second_datetime_field
     [(interval_leading_field_precision )]
end_field ::= non_second_datetime_field
     | SECOND[(interval_fractional_seconds_precision)]
single_datetime_field ::= non_second_datetime_field
     [(interval_leading_field_precision)]
     | SECOND[(interval_leading_field_precision
     [, (interval_fractional_seconds_precision)]
datetime_field ::= non_second_datetime_field | SECOND
non_second_datetime_field ::= YEAR | MONTH | DAY | HOUR | MINUTE
interval_fractional_seconds_precision ::= unsigned_integer
interval_leading_field_precision ::= unsigned_integer
quote ::= '
unsigned_integer ::= digit...
```

数値リテラル構文

以下の構文は、ODBC の数値リテラルで使用されます。

```
numeric_literal ::= signed_numeric_literal | unsigned_numeric_literal
signed_numeric_literal ::= [sign] unsigned_numeric_literal
unsigned_numeric_literal ::= exact_numeric_literal
     | approximate_numeric_literal
exact_numeric_literal ::= unsigned_integer [period[unsigned_integer]]
     | period unsigned_integer
sign ::= plus_sign | minus_sign
approximate_numeric_literal ::= mantissa E exponent
mantissa ::= exact_numeric_literal
exponent ::= signed_integer
signed_integer ::= [sign] unsigned_integer
unsigned_integer ::= digit...
plus_sign ::= +
minus_sign ::= -
digit ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
period ::= .
```

予約キーワードのリスト

コア SQL 文法をサポートするドライバーとの互換性を確保するため、アプリケーションで使用を回避すべきキーワードが存在します。

これらの単語は、最小 SQL 文法を制約しません。#define 値 SQL_ODBC_KEYWORDS には、これらのキーワードのリストがコンマ区切りで含まれています。

複数の SQL 標準および solidDB ODBC API の予約キーワードの完全なリストについては、「*IBM solidDB SQL ガイド*」の『予約語』を参照してください。

表 179. 予約キーワードのリスト

キーワード	キーワード	キーワード	キーワード
ABSOLUTE	ACTION	ADA	ADD
ALL	ALLOCATE	ALTER	AND
ANY	ARE	AS	ASC
ASSERTION	AT	AUTHORIZATION	AVG
BEGIN	BETWEEN	BIT	BIT_LENGTH
BOTH	BY	CASCADE	CASCADED
CASE	CAST	CATALOG	CHAR
CHAR_LENGTH	CHARACTER	CHARACTER_LENGTH	CHECK
CLOSE	COALESCE	COLLATE	COLLATION
COLUMN	COMMIT	CONNECT	CONNECTION
CONSTRAINT	CONSTRAINTS	CONTINUE	CONVERT
CORRESPONDING	COUNT	CREATE	CROSS
CURRENT	CURRENT_DATE	CURRENT_TIME	CURRENT_TIMESTAMP
CURRENT_USER	CURSOR	DATE	DAY
DEALLOCATE	DEC	DECIMAL	DECLARE
DEFAULT	DEFERRABLE	DEFERRED	DELETE
DESC	DESCRIBE	DESCRIPTOR	DIAGNOSTICS
DISCONNECT	DISTINCT	DOMAIN	DOUBLE
DROP	ELSE	END	END-EXEC
ESCAPE	EXCEPT	EXCEPTION	EXEC
EXECUTE	EXISTS	EXTERNAL	EXTRACT
FALSE	FETCH	FIRST	FLOAT
FOR	FOREIGN	FORTRAN	FOUND
FROM	FULL	GET	GLOBAL
GO	GOTO	GRANT	GROUP

表 179. 予約キーワードのリスト (続き)

キーワード	キーワード	キーワード	キーワード
HAVING	HOUR	IDENTITY	IMMEDIATE
IN	INCLUDE	INDEX	INDICATOR
INITIALLY	INNER	INPUT	INSENSITIVE
INSERT	INT	INTEGER	INTERSECT
INTERVAL	INTO	IS	ISOLATION
JOIN	KEY	LANGUAGE	LAST
LEADING	LEFT	LEVEL	LIKE
LOCAl	LOWER	MATCH	MAX
MIN	MINUTE	MODULE	MONTH
NAMES	NATIONAL	NATURAL	NCHAR
NEXT	NO	NONE	NOT
NULL	NULLIF	NUMERIC	OCTET_LENGTH
OF	ON	ONLY	OPEN
OPTION	OR	ORDER	OUTER
OUTPUT	OVERLAPS	PASCAL	POSITION
PRECISION	PREPARE	PRESERVE	PRIMARY
PRIOR	PRIVILEGES	PROCEDURE	PUBLIC
READ	REAL	REFERENCES	RELATIVE
RESTRICT	REVOKE	RIGHT	ROLLBACK
ROWS	SCHEMA	SCROLL	SECOND
SECOND	SECTION	SELECT	SESSION
SESSION_USER	SET	SIZE	SMALLINT
SOME	SPACE	SQL	SQLCA
SQLCODE	SQLERROR	SQLSTATE	SQLWARNING
SUBSTRING	SUM	SYSTEM_USER	TABLE

表 179. 予約キーワードのリスト (続き)

キーワード	キーワード	キーワード	キーワード
TEMPORARY	THEN	TIME	TIMESTAMP
TIMEZONE_HOUR	TIMEZONE_MINUTE	TO	TRAILING
TRANSACTION	TRANSLATE	TRANSLATION	TRIM
TRUE	UNION	UNIQUE	UNKNOWN
UPDATE	UPPER	USAGE	USER
USING	VALUE	VALUES	VARCHAR
VARYING	VIEW	WHEN	WHENEVER
WHERE	WITH	WORK	WRITE
YEAR	ZONE		

付録 E. データ型

このセクションでは、ODBC データ型について説明します。

ODBC では、以下の一連のデータ型が定義されています。

- SQL データ型。データ・ソース (例えば、solidDB サーバー) に格納されるデータのデータ型を示します。
- C データ型。アプリケーション・バッファーに格納されるデータのデータ型を示します。

各 SQL データ型は、ODBC C データ型に対応しています。データ・ソースからデータを返す前に、ドライバーは、指定された C データ型にそのデータを変換します。データ・ソースにデータを送信する前に、ドライバーは、指定された C データ型からそのデータを変換します。

ドライバー固有の SQL データ型については、ドライバーの資料を参照してください。

SQL データ型

各 DBMS は、SQL-92 標準に従って、独自の SQL データ型セットを定義しています。SQL-92 標準の各 SQL データ型について、型 ID と呼ばれる #define 値が ODBC 関数の引数として渡され、また結果セットのメタデータ内で返されます。

ドライバーは、データ・ソース固有の SQL データ型を ODBC SQL データ型 ID およびドライバー固有の SQL データ型 ID にマップします。SQL データ型は、インプリメンテーション記述子の SQL_DESC_CONCISE_TYPE フィールドに格納されます。

solidDB の ODBC ドライバーは、以下の SQL_92 データ型をサポートしません。

- BIT
- BIT_VARYING
- TIME_WITH_TIMEZONE
- TIMESTAMP_WITH_TIMEZONE
- NATIONAL_CHARACTER

C データ型

ODBC は、C データ型と、それに対応する ODBC 型 ID を定義します。

アプリケーションは、以下の関数の 1 つを呼び出します。

- SQLBindCol または SQLGetData を呼び出して、適用可能な C 型 ID を TargetType 引数で受け渡します。このようにして、アプリケーションは、結果セット・データを受け取るバッファーの C データ型を指定します。

- SQLBindParameter を呼び出して、該当する C 型 ID を ValueType 引数で受け渡します。このようにして、アプリケーションは、ステートメント・パラメーターを含むバッファーの C データ型を指定します。

C データ型は、アプリケーション記述子の SQL_DESC_CONCISE_TYPE フィールドに格納されます。

注: ドライバー固有の C データ型は存在しません。

データ型 ID

データ型 ID は、記述子の SQL_DESC_CONCISE_TYPE フィールドに格納されます。アプリケーション内のデータ型 ID は、ドライバーに対して自身のバッファーを記述します。

また、ドライバーから結果セットに関するメタデータをリトリーブし、したがってアプリケーションは、データ・ストレージに使用する C バッファーの型を認識できます。アプリケーションは、以下の関数を呼び出すことにより、データ型 ID を使用してこれらのタスクを実行します。

- アプリケーション・バッファーの C データ型を記述するために、アプリケーションは、SQLBindParameter、SQLBindCol、および SQLGetData を呼び出します。
- 動的パラメーターの SQL データ型を記述するために、アプリケーションは、SQLBindParameter を呼び出します。
- 結果セット列の SQL データ型をリトリーブするために、アプリケーションは、SQLColAttribute および SQLDescribeCol を呼び出します。
- パラメーターの SQL データ型をリトリーブするために、アプリケーションは、SQLDescribeParameter を呼び出します。
- 各種スキーマ情報の SQL データ型をリトリーブするために、アプリケーションは、SQLColumns、SQLProcedureColumns、および SQLSpecialColumns を呼び出します。
- サポートされているデータ型のリストをリトリーブするために、アプリケーションは、SQLGetTypeInfo を呼び出します。

さらに、上記のタスクの実行に、SQLSetDescField および SQLSetDescRec の記述子関数も使用されます。詳しくは、SQLSetDescField 関数と SQLSetDescRec 関数を参照してください。

SQL データ型

指定したドライバーおよびデータ・ソースが、ODBC 文法で定義されているすべての SQL データ型をサポートしているとは限りません。また、ドライバー固有の追加の SQL データ型をサポートしている場合もあります。

ドライバーのサポートは、SQL-92 準拠のレベルにより決まります。ドライバーがどのデータ型をサポートするかを判別するには、アプリケーションで SQLGetTypeInfo を呼び出します。 281 ページの『SQLGetTypeInfo 結果セットの例』を参照してください。ドライバー固有の SQL データ型については、ドライバーの資料を参照してください。

ドライバーは、以下の関数を使用して列およびパラメーターのデータ型を記述する際にも SQL データ型を返します。

- SQLColAttribute
- SQLColumns
- SQLDescribeCol
- SQLDescribeParam
- SQLProcedureColumns
- SQLSpecialColumns

注:

SQL データ型の値および特性を格納しているフィールドについて詳しくは、291 ページの『データ型 ID および記述子』を参照してください。

以下の表は、SQL データ型の総合的なリストではありませんが、よく使用される名前、範囲、および制限が示されています。データ・ソースが表にリストされたデータ型の一部のみサポートする場合や、ご使用のドライバーによっては、データ型の特性が表の説明と異なる場合があります。この表では、適宜 SQL-92 の関連データ型も説明しています。

表 180. 一般的な SQL データ型の名前、範囲、および制限

SQL 型 ID [1]	**一般的な SQL データ型 [2]**	**一般的な型の説明**
SQL_CHAR	CHAR(n)	固定ストリング長が *n* の文字ストリング。
SQL_VARCHAR	VARCHAR(n)	最大ストリング長が *n* の可変長文字ストリング。
SQL_LONGVARCHAR	LONG VARCHAR	可変長文字データ。最大長は、データ・ソースに依存します。[3]
SQL_WCHAR	WCHAR(n)	固定ストリング長が *n* の Unicode 文字ストリング。
SQL_WVARCHAR	VARWCHAR(n)	最大ストリング長が *n* の Unicode 可変長文字ストリング。
SQL_WLONGVARCHAR	LONGWVARCHAR	Unicode 可変長文字データ。最大長は、データ・ソースに依存します。
SQL_DECIMAL	DECIMAL(p, s)	精度が *p* で位取りが *s* の符号付きの厳密な数値。(最大精度は、ドライバーにより定義されます。) (1 <= *p* <= 16、*s* <= *p*) [4]
SQL_NUMERIC	NUMERIC(p,s)	精度が *p* で位取りが *s* の符号付きの厳密な数値。 (1 <= *p* <= 16、*s* <= *p*) [4]

表 180. 一般的な SQL データ型の名前、範囲、および制限 (続き)

SQL 型 ID [1]	一般的な SQL データ型 [2]	一般的な型の説明
SQL_SMALLINT	SMALLINT	精度が 5 で位取りが 0 の厳密な数値。 (符号付き: -32,768 <= n <= 32,767、符号なし: 0 <= n <= 65,535) solidDB では、SMALLINT は符号付きのもののみをサポートし、符号なしのものをサポートしません。[5]
SQL_INTEGER	INTEGER	精度が 10 で位取りが 0 の厳密な数値。(符号付き: -2^{31} <= n <= 2^{31} -1、符号なし: 0 <= n <= 2^{32} -1) solidDB では、INTEGER は符号付きのもののみをサポートし、符号なしのものをサポートしません。[5]
SQL_REAL	REAL	バイナリー精度が 24 の符号付き近似数値 (ゼロまたは 10^{-38} から 10^{38} の絶対値)。
SQL_FLOAT	FLOAT(p)	バイナリー精度が *p* 以上の符号付き近似数値。(最大精度は、ドライバーにより定義されます。) [6]
SQL_DOUBLE	DOUBLE PRECISION	バイナリー精度が 53 の符号付き近似数値 (ゼロまたは 10^{-308} から 10^{308} の絶対値)。
SQL_BIT	BIT	単一ビットのバイナリー・データ。注: solidDB は、BIT/SQL_BIT をサポートしていません。 [7]
SQL_TINYINT	TINYINT	精度が 3 で位取りが 0 の厳密な数値。 (符号付き :-128 <= *n* <= 127、符号なし: 0 <= *n* <= 255) solidDB では、TINYINT は符号付きのもののみをサポートし、符号なしのものをサポートしません。[5]
SQL_BIGINT	BIGINT	精度が 19 (符号付きの場合) または 20 (符号なしの場合) で位取りが 0 の厳密な数値。(符号付き: -2^{63} <= *n* <= 2^{63} - 1、符号なし: 0 <= *n* <= 2^{64} - 1) solidDB では、BIGINT は符号付きのもののみをサポートし、符号なしのものをサポートしません。[3]、[5]
SQL_BINARY	BINARY(n)	固定長が *n* のバイナリー・データ。[3]
SQL_VARBINARY	VARBINARY(n)	最大長が *n* の可変長バイナリー・データ。最大長は、ユーザーが設定します。[3]
SQL_LONGVARBINARY	LONG VARBINARY	可変長バイナリー・データ。最大長は、データ・ソースに依存します。[3]
SQL_TYPE_DATE [8]	DATE	グレゴリオ暦のルールによる年、月、および日のフィールド。(296 ページの『グレゴリオ暦の制約』を参照してください。)

表 180. 一般的な SQL データ型の名前、範囲、および制限 (続き)

SQL 型 ID [1]	一般的な SQL データ型 [2]	一般的な型の説明
SQL_TYPE_TIME [8]	TIME(p)	時、分、および秒のフィールド。時の有効値は、00 から 23 です。分の有効値は、00 から 59 です。秒の有効値は、00 から 61 (60 および 61 は、「うるう秒」処理用) です (http://tycho.usno.navy.mil/leapsec.html を参照してください)。精度 *p* は、秒フィールドの精度を示します。
SQL_TYPE_TIMESTAMP [8]	TIMESTAMP(p)	日付および時刻のデータ型に定義されている有効値を持つ年、月、日、時、分、および秒のフィールド。

注:

[1] これは、SQLGetTypeInfo の呼び出しにより DATA_TYPE 列に返される値です。

[2] これは、SQLGetTypeInfo の呼び出しにより NAME 列および CREATE PARAMS 列に返される値です。 NAME 列には、例えば CHAR のような指定が返され、CREATE PARAMS 列には、精度、位取り、および長さのような作成パラメーターのコンマ区切りリストが返されます。

[3] このデータ型に対応するものは、SQL-92 にはありません。

[4] SQL_DECIMAL データ型と SQL_NUMERIC データ型との違いは、精度のみです。DECIMAL(p,s) の精度が、インプリメンテーションで定義された p 以上の 10 進数精度であるのに対し、NUMERIC(p,s) の精度は p に完全に等しくなります。

[5] アプリケーションは SQLGetTypeInfo または SQLColAttribute を使用して、結果セット内の特定のデータ型や列が符号なしであるかどうかを判別します。

[6] インプリメンテーションによって、SQL_FLOAT の精度は 24 または 53 のいずれかになります。24 である場合、SQL_FLOAT データ型は SQL_REAL と同じになり、53 である場合、SQL_FLOAT データ型は SQL_DOUBLE と同じになります。

[7] SQL_BIT データ型の特性は、SQL-92 の BIT 型と異なります。

[8] このデータ型に対応するものは、SQL-92 にはありません。

SQLGetTypeInfo 結果セットの例

アプリケーションは、SQLGetTypeInfo の結果セットを呼び出し、指定データ・ソースに対してサポートされたデータ型およびその特性のリストを調べます。

あるデータ・ソースに対し SQLGetTypeInfo により返されるデータ型の例を以下に示します。このデータ・ソースでは、「DATA_TYPE」に示されたすべてのデータ型がサポートされています。

この例は、ページ幅に収まるよう 3 つのセクションに分割されています。実際には、すべてが 1 つの例です。

表 181. SQLGetTypeInfo が返すデータ型 (1)

TYPE_NAME	DATA_TYPE	COLUMN_ SIZE	LITERAL_PREFIX	LITERAL_SUFFIX	CREATE_PARAMS	NULLABLE
″char″	SQL_CHAR	255	″″	″″	″length″	SQL_TRUE
″text″	SQL_LONG VARCHAR	2147483647	″″	″″	<NULL>	SQL_TRUE

表 181. SQLGetTypeInfo が返すデータ型 (1) (続き)

TYPE_ NAME	DATA_TYPE	COLUMN_ SIZE	LITERAL_ PREFIX	LITERAL_ SUFFIX	CREATE_ PARAMS	NULLABLE
″decimal″	SQL_DECIMAL	18 [a]	<NULL>	<NULL>	″precision, scale″	SQL_TRUE
″real″	SQL_REAL	7	<NULL>	<NULL>	<NULL>	SQL_TRUE
″datetime″	SQL_TYPE_ TIMESTAMP	29 [b]	″″	″″	<NULL>	SQL_TRUE

表 182. SQLGetTypeInfo が返すデータ型 (2)

(続き)	CASE_ SENSITIVE	SEARCHABLE	UNSIGNED_ ATTRIBUTE	FIXED_ PREC_ SCALE	AUTO_ UNIQUE_ VALUE	LOCAL_ TYPE_ NAME
SQL_CHAR	SQL_FALSE	SQL_ SEARCHABLE	<NULL>	SQL_FALSE	<NULL>	″char″
SQL_LONG VARCHAR	SQL_FALSE	SQL_PRED_ CHAR	<NULL>	SQL_FALSE	<NULL>	″text″
SQL_DECIMAL	SQL_FALSE	SQL_PRED_ BASIC	SQL_FALSE	SQL_FALSE	SQL_FALSE	″decimal″
SQL_REAL	SQL_FALSE	SQL_PRED_ BASIC	SQL_FALSE	SQL_FALSE	SQL_FALSE	″real″
SQL_TYPE_ TIMESTAMP	SQL_FALSE	SQL_ SEARCHABLE	<NULL>	SQL_FALSE	<NULL>	″datetime″

表 183. SQLGetTypeInfo が返すデータ型 (3)

(続き)	MINIMUM_ SCALE	MAXIMUM_ SCALE	SQL_DATA_ TYPE	SQL_ DATETIME_ SUB	NUM_ PREC_ RADIX	INTERVAL_ PRECISION
SQL_CHAR	<NULL>	<NULL>	SQL_CHAR	<NULL>	<NULL>	<NULL>
SQL_LONG VARCHAR	<NULL>	<NULL>	SQL_LONG VARCHAR	<NULL>	<NULL>	<NULL>
SQL_DECIMAL	0	16	SQL_ DECIMAL	<NULL>	10	<NULL>
SQL_REAL	<NULL>	<NULL>	SQL_REAL	<NULL>	10	<NULL>
SQL_TYPE_ TIMESTAMP	3	3	SQL_ DATETIME	SQL_CODE_ TIMESTAMP	<NULL>	12

上記の表の脚注番号に関する説明

[a] 16 桁、小数点は 1 つ、負の数値に対しオプションの符号文字

[b] yyyy-mm-dd hh:MM:ss.nnnnnnnnn の 29 文字表示

C データ型

ODBC ドライバーは、すべての C 型と SQL 文字型との変換の必要性に対応するため、すべての C データ型をサポートしています。

C データ型は以下の関数で指定されます。

- TargetType 引数を指定した SQLBindCol および SQLGetData 関数
- ValueType 引数を指定した SQLBindParameter
- ARD1 または APD2 の SQL_DESC_CONCISE_TYPE フィールドを設定する SQLSetDescField
- Type 引数、SubType 引数 (必要な場合)、および ARD [1] または APD [2] のハンドルに設定された DescriptorHandle 引数を指定した SQLSetDescRec。

以下の表には次の 3 つの列があります。

1. 最初の列には、C 型 ID が含まれています。 これら C 型 ID は SQLBindCol などの関数に渡され、列にバインドされる変数の型を示します。以下の例では、SQL_C_DECIMAL が C 型 ID です。

   ```
   // MySharedVariable を結果セットの列 1 にバインドします。
   // (列 1 は DECIMAL 列です。) C 型 ID である SQL_C_DECIMAL は、
   // 変数 MySharedVariable が DECIMAL と同等の型であることを示します。
   SQLBindCol(..., 1, SQL_C_DECIMAL, &MySharedVariable, ...);
   ```

2. 2 番目の列は、C 型 ID それぞれに関連付けられている ODBC C データ型を示します。この ODBC C データ型は、ODBC プログラムで変数を定義する際に使用する「typedef」です。これによって、プラットフォーム固有の要件からプログラムを分離することができます。例えば、SQL FLOAT 型の列があり、その列に変数をバインドすると仮定します。以下のように、変数を SQLFLOAT 型と宣言することができます。

   ```
   SQLFLOAT MySharedVariable; // SQL FLOAT 型の列にバインドできます。
   ```

3. 3 番目の列には、ODBC C データ型「typdef」に対応する C 型定義の例が含まれています。この列の例は、32 ビット・プラットフォームで最も頻繁に使用される定義を示すものです。この列で指定されているデータ型は一例で、プラットフォームに依存します。

   ```
   // SQL FLOAT 型の列にバインドする変数の移植可能な宣言の方法です。
   SQLFLOAT MySharedSQLFLOATVariable = 0.0;
   // SQL INTEGER 型の列にバインドする変数の移植不可能な宣言の方法です。
   // この宣言はほとんどの 32 ビット・プラットフォームでは正しく機能しますが、
   // 64 ビット・プラットフォームでは失敗することがあります。
   long int MySharedSQLINTEGERVariable = 0;
   ```

---

1. ARD: アプリケーション行記述子

   これらの記述子には、SQL ステートメントによって返される列にバインドされるアプリケーション変数に関する情報が含まれています。その情報には、バインドされた変数のアドレス、長さ、および C データ型が含まれます。

2. APD: アプリケーション・パラメーター記述子

   これらの記述子には、例えば SQL ステートメントで使用されるパラメーター・マーカー ("?") にバインドされるアプリケーション変数に関する情報が含まれています。

   ```
   SELECT * FROM table1 WHERE id = ?
   ```

   記述子の情報には、バインドされた変数のアドレス、長さ、および C データ型が含まれます。

```
// MySharedSQLFLOATVariable を結果セットの列 1 にバインドします。
SQLBindCol(..., 1, SQL_C_DOUBLE, &MySharedSQLFLOATVariable, ...);
// MySharedSQLINTEGERVariable を結果セットの列 2 にバインドします。
SQLBindCol(..., 2, SQL_C_SLONG, &MySharedSQLINTEGERVariable, ...);
```

C 型 ID と ODBC C 型は、必ずしも類似した名前を持つとは限りません。C 型 ID の名前は C 言語データ型 (「float」など) を基にしており、ODBC C typedef の名前は、SQL データ型を基にしています。C 言語の「float」は SQL の「REAL」に対応しているため、表では「SQL_C_FLOAT」を ODBC C typedef「SQLREAL」に対応する C 型 ID としてリストしています。

表 184. C と ODBC の名前の対応

C 型 ID	ODBC C typedef	C 型
SQL_C_CHAR	SQLCHAR	unsigned char
SQL_C_STINYINT	SCHAR	char
SQL_C_UTINYINT [i]	UCHAR	unsigned char
SQL_C_SSHORT [h]	SQLSMALLINT	short int
SQL_C_USHORT [h] [i]	SQLUSMALLINT	unsigned short int
SQL_C_SLONG [h]	SQLINTEGER	long int
SQL_C_ULONG [h] [i]	SQLUINTEGER	unsigned long int
SQL_C_SBIGINT	SQLBIGINT	_int64 [g]
SQL_C_UBIGINT [i]	SQLUBIGINT	unsigned _int64 [g] solidDB はこのような符号なしデータ型をサポートしていません。
SQL_C_FLOAT	SQLREAL	float
SQL_C_DOUBLE	SQLDOUBLE SQLFLOAT	double
SQL_C_NUMERIC	SQLNUMERIC	unsigned char [f]
SQL_C_DECIMAL	SQLDECIMAL	unsigned char [f]
SQL_C_BINARY	SQLCHAR *	unsigned char *
SQL_C_TYPE_DATE [c]	SQL_DATE_STRUCT	struct tagDATE_STRUCT{ SQLSMALLINT year; SQLUSMALLINT month; SQLUSMALLINT day; } DATE_STRUCT; [a]

表 184. C と ODBC の名前の対応 (続き)

C 型 ID	ODBC C typedef	C 型

SQL_C_TYPE_TIME [c]	SQL_TIME_STRUCT	struct tagTIME_STRUCT {
		SQLUSMALLINT hour;
		SQLUSMALLINT minute;[d]
		SQLUSMALLINT second;[e]
		}

SQL_C_TYPE_TIMESTAMP [c]	SQL_TIMESTAMP_STRUCT	struct tagTIMESTAMP_STRUCT {
		SQLSMALLINT year; [a]
		SQLUSMALLINT month; [b]
		SQLUSMALLINT day; [c]
		SQLUSMALLINT hour;
		SQLUSMALLINT minute; [d]
		SQLUSMALLINT second;[e]
		SQLUINTEGER fraction;
		}

表 184. C と ODBC の名前の対応 (続き)

C 型 ID	ODBC C typedef	C 型

注:

[a] C の日時データ型の year、month、day、hour、minute、および second のフィールドの値は、グレゴリオ暦の制約に準拠している必要があります。 (296 ページの『グレゴリオ暦の制約』を参照してください。)

[b] fraction フィールドの値は、0 から 999,999,999 (10 億から 1 を引いた値) の範囲のナノ秒 (10 億分の 1 秒) の数です。例えば、fraction フィールドの値は、2 分の 1 秒の場合は 500,000,000、1,000 分の 1 秒 (1 ミリ秒) の場合は 1,000,000、100 万分の 1 秒 (1 マイクロ秒) の場合は 1,000、10 億分の 1 秒 (1 ナノ秒) の場合は 1 です。

[c] ODBC 2.x では、C の日付、時刻、およびタイム・スタンプのデータ型は SQL_C_DATE、SQL_C_TIME、および SQL_C_TIMESTAMP です。

[d] SQL_NUMERIC_STRUCT 構造体の val フィールドには、位取り整数としてリトル・エンディアン・モード (左端のバイトが最下位バイト) で数値が格納されています。例えば、数値 10.001 基数 10 に 4 桁の位取りを指定すると、100010 という整数に位取りされます。これは 16 進形式では 186AA なので、SQL_NUMERIC_STRUCT の値は「AA 86 01 00 00 ... 00」となり、SQL_MAX_NUMERIC_LEN #define で定義されたバイト数になります。

[e] SQL_C_NUMERIC データ型の precision および scale のフィールドは、ドライバーからアプリケーションへの出力専用で、アプリケーションからの入力に使用されることはありません。ドライバーが数値を SQL_NUMERIC_STRUCT に書き込む際、ドライバー固有のデフォルトを precision フィールドの値として使用し、scale フィールドには、アプリケーション記述子の SQL_DESC_SCALE フィールドの値 (デフォルトは 0) を使用します。アプリケーションは、アプリケーション記述子の SQL_DESC_PRECISION および SQL_DESC_SCALE のフィールドを設定して、精度および位取りに独自の値を提供することができます。

[f] DECIMAL および NUMERIC のデータ型には、1 バイト/1 文字より多くが必要です。データ型は実際には、列に必要な精度に基づき、配列として宣言されます。例えば、SQL DECIMAL(10,4) 型の列は、10 桁、符号文字、小数点文字、およびストリング終止符を考慮して、SQL_DECIMAL[13] として宣言されます。

[g] コンパイラーによっては、_int64 が提供されない場合があります。

[h] _SQL_C_SHORT、SQL_C_LONG、および SQL_C_TINYINT は ODBC では符号付きおよび符号なしの型 SQL_C_SSHORT と SQL_C_USHORT、SQL_C_SLONG と SQL_C_ULONG、および SQL_C_STINYINT と SQL_C_UTINYINT に置き換えられます。ODBC 2.x アプリケーションと連携する ODBC 3.x ドライバーは、SQL_C_SHORT、SQL_C_LONG、および SQL_C_TINYINT をサポートしていなければなりません。これらを呼び出すと、ドライバー・マネージャーがドライバーに渡すからです。

[i] solidDB は符号なし SQL データ型をサポートしていません。符号なし C データ型を符号付き SQL 列にバインドできますが、SQL 列と C 変数に格納された値が、両方のデータ型の有効範囲にない場合はバインドしないでください。例えば、符号付き TINYINT 列には -128 から +127 の値が含まれ、符号なし SQL_C_UTINYINT 変数に 0 から 255 の値が含まれているため、値を正しく解釈させるには、列とバインドされた変数には 0 から +127 までの値しか格納できません。

64 ビットの整数構造体

Microsoft C コンパイラーでは、C データ型 ID の SQL_C_SBIGINT と SQL_C_UBIGINT は _int64 として定義されます。 Microsoft 以外の C コンパイラーを使用する場合は、C 型は異なることがあります。使用しているコンパイラーが 64 ビット整数をネイティブにサポートしている場合、ドライバーまたはアプリケーションの ODBCINT64 をネイティブな 64 ビット整数型として定義します。使用しているコンパイラーが 64 ビット整数をネイティブにサポートしていない場合は、以下の構造体を定義して、これらの C 型に確実にアクセスできるようにします。

```
typedef struct{
SQLUINTEGER dwLowWord;
SQLUINTEGER dwHighWord;
} SQLUBIGINT

typedef struct {
SQLUINTEGER dwLowWord;
SQLINTEGER sdwHighWord;
} SQLBIGINT
```

64 ビット整数は 8 バイト境界に位置合わせされるので、これらの構造体を必ず 8 バイト境界に位置合わせしておいてください。

注:

solidDB は符号付き BIGINT はサポートしていますが、符号なし BIGINT はサポートしていません。

デフォルト C データ型

SQLBindCol、SQLGetData、または SQLBindParameter で SQL_C_DEFAULT を指定するアプリケーションでは、ドライバーは出力バッファーまたは入力バッファーの C データ型が、バッファーのバインド先である列やパラメーターの SQL データ型に対応していると仮定します。

重要: さまざまなプラットフォームを使用する際の互換性の問題を回避するために、SQL_C_DEFAULT の使用を避けることを強く推奨します。代わりに使用中のバッファーの C 型を指定してください。

ドライバーは以下の理由で正しいデフォルト C 型を判断できない場合があります。

- DBMS によって、列またはパラメーターの SQL データ型がプロモートされていることがあります。この場合、ドライバーは元の SQL データ型を判別できず、そのため、対応するデフォルト C データ型を判別できません。
- DBMS によって、列またはパラメーターのデータ型が符号付きか、符号なしかが判別されています。この場合、ドライバーは特定の SQL データ型についてこの判別を行えず、そのため、対応するデフォルト C データ型についてこの判別を行えません。

297 ページの『SQL から C データ型へのデータ変換』を参照してください。

SQL_C_TCHAR

SQL_C_TCHAR 型 ID は Unicode に使用します。この ID は、ASCII と Unicode の文字セットの両方を使用するようにコンパイルされる、文字データの転送用アプ

リケーションで使用します。SQL_C_TCHAR は従来の意味の型 ID ではなく、Unicode 変換用にヘッダー・ファイルに含まれているマクロであることに注意してください。SQL_C_CHAR または SQL_C_WCHAR は、UNICODE #define の設定に応じて SQL_C_TCHAR の代わりとなります。

数値リテラル

文字ストリングに数値データ値を格納するには、数値リテラルを使用します。

数値リテラルの構文は、以下の変換中にターゲットに何を格納するかを指定します。

- SQL データから SQL_C_CHAR ストリングへの変換
- C データから SQL_CHAR または SQL_VARCHAR のストリングへの変換

構文は、以下の変換中にソースに何が格納されているかの検証もします。

- SQL_C_CHAR ストリングとして格納されている数値から数値 SQL データへの変換
- SQL_CHAR ストリングとして格納されている数値から数値 C データへの変換

詳しくは、 272 ページの『数値リテラル構文』を参照してください。

変換ルール

以下のルールは数値リテラルを伴う変換に適用されます。このトピックでは以下の用語を使用しています。

表 185. 数値リテラルを伴う変換

用語	意味
ストア割り当て	SQLExecute および SQLExecDirect の呼び出しの際に、データベース内の表の列にデータを送信することを指します。ストア割り当て中は、「ターゲット」はデータベース列を、「ソース」はアプリケーション・バッファーのデータを指します。
リトリーブ割り当て	SQLFetch、 SQLGetData、および SQLFetchScroll の呼び出しの際に、データベースからアプリケーション・バッファーにデータをリトリーブすることを指します。リトリーブ割り当て中は、「ターゲット」はアプリケーション・バッファーを、「ソース」はデータベース列を指します。
CS	文字ソース内の値。
NT	数値ターゲット内の値。
NS	数値ソース内の値。
CT	文字ターゲット内の値。
厳密な数値リテラルの精度	リテラルに含まれている桁数。

表 185. 数値リテラルを伴う変換 (続き)

用語	意味
厳密な数値リテラルの位取り	明示的または暗黙の小数点以下の桁数。
近似数値リテラルの精度	リテラルの小数部の精度。

文字ソースから数値ターゲットへの変換のルール

以下は、文字ソース (CS) から数値ターゲット (NT) への変換時のルールです。

1.

   CS 内の先行スペースまたは末尾スペースを除去して得た値に CS を置き換えます。CS が有効な数値リテラルでない場合は、SQLSTATE 22018 (キャスト指定に関する無効な文字値) が返されます。

2.

   小数点より前の先行ゼロ、小数点の後の後続ゼロ、あるいはその両方を除去して得た値に CS を置き換えます。

3.

   CS を NT に変換します。変換によって有効数字が失われる場合は、SQLSTATE 22003 (範囲外の数値) が返されます。変換によって無効数字が失われる場合は、SQLSTATE 01S07 (小数部の切り捨て) が返されます。

以下は、数値ソース (NS) から文字ターゲット (CT) への変換時のルールです。

1.

   LT が CT の文字数の長さであるとします。

   リトリーブ割り当てでは、LT は文字数で表したバッファーの長さから、この文字セットの NULL 終了文字のバイト数を差し引いたものと等しくなります。

2.

   NS の型に応じて、以下のいずれかのアクションを実行します。

   •

     NS が厳密な数値型の場合は、YP が厳密な数値リテラルの定義に準拠する最短の文字ストリングであるとして、YP の位取りが NS の位取りと同じであり、YP の解釈値が NS の絶対値になるようにします。

   •

     NS が近似数値型の場合は、YP が以下のような文字ストリングであるとします。

   ケース:

   a. NS が 0 に等しい場合、YP はストリング「0」です。

b. YSN が厳密な数値リテラルの定義に準拠する最短の文字ストリングであるとして、その解釈値が NS の絶対値になるようにします。YSN の長さが NS のデータ型の (精度 + 1) よりも短い場合は、YP が YSN と等しくなるものとします。

c. それ以外の場合は、YP は近似数値リテラルの定義に準拠する最短文字ストリングで、その解釈値が NS の絶対値であり、その小数部が「0」以外の 1 桁の数字とそれに続くピリオドおよび符号なし整数からなります。

3. NS が 0 未満の場合は、Y が以下の結果であるとします。

   ```
   '-' || YP
   ```

   ここで「||」は、ストリング連結演算子です。

   それ以外の場合は、Y が YP と等しくなるものとします。

4.

   LY が Y の文字数の長さであるとします。

5. LY の値に応じて、以下のいずれかのアクションを実行します。

   - LY が LT と等しい場合は、CT は Y に設定されます。
   - LY が LT より小さい場合は、CT は適切なスペースの数だけ右側に拡張された Y に設定されます。
   - それ以外の場合 (LY > LT) は、Y の最初の LT 文字を CT にコピーします。

     ケース:

     – これがストア割り当ての場合は、エラー SQLSTATE 22001 (ストリング・データ、右側切り捨て) を返します。

     – これがリトリーブ割り当ての場合は、警告 SQLSTATE 01004 (ストリング・データ、右側切り捨て) を返します。コピーによって (後続ゼロ以外の) 小数桁を失うことになる場合は、ドライバー定義によって、以下のいずれかのアクションが発生します。

     a. ドライバーは Y のストリングを適切な位取りに合わせて切り捨て (ゼロの場合もある)、結果を CT に書き込みます。

     b. ドライバーは Y のストリングを適切な位取りに合わせて丸め (ゼロの場合もある)、結果を CT に書き込みます。

     c. ドライバーは切り捨てや丸めは行わず、Y の最初の LT 文字を CT にコピーするだけです。

数値データ型の精度と位取りのオーバーライド用デフォルト

以下の表には、数値データ型の精度と位取りのオーバーライド用デフォルト値を示します。

表 186. 数値データ型の精度と位取りのオーバーライド用デフォルト値

関数呼び出し	設定	オーバーライド
SQLBindCol または SQLSetDescField	ARD の SQL_DESC_TYPE フィールドは SQL_C_NUMERIC に設定されます。	ARD の SQL_DESC_SCALE フィールドは 0 に設定され、SQL_DESC_PRECISION フィールドはドライバーが定義したデフォルト精度に設定されます。[a]
SQLBindParameter または SQLSetDescField	APD の SQL_DESC_SCALE フィールドは SQL_C_NUMERIC に設定されます。	ARD の SQL_DESC_SCALE フィールドは 0 に設定され、SQL_DESC_PRECISION フィールドはドライバーが定義したデフォルト精度に設定されます。これは入力、入出力、または出力のパラメーターに適用されます。[a]
SQLGetData	データは SQL_C_NUMERIC 構造体に返されます。	デフォルトの SQL_DESC_SCALE および SQL_DESC_PRECISION のフィールドが使用されます。[b]

上記の表の脚注番号に関する説明

[a] デフォルトがアプリケーションに受け入れられない場合、アプリケーションは SQLSetDescField または SQLSetDescRec を呼び出して、SQL_DESC_SCALE または SQL_DESC_PRECISION のフィールドを設定します。

[b] デフォルトが受け入れられない場合は、アプリケーションは SQLSetDescRec または SQLSetDescField を呼び出して、フィールドを設定し、次に SQL_ARD_TYPE の TargetType を指定して SQLGetData を呼び出して、記述子フィールドの値を使用します。

データ型 ID および記述子

各 ID が単一のデータ型を参照する「簡易」SQL データ型および C データ型とは異なり、記述子は、すべての場合に単一値を使用してデータ型を識別するとは限りません。場合によっては、記述子は、詳細データ型および型のサブコードを使用します。大部分のデータ型に関して、詳細データ型 ID は、簡易型 ID と一致します。

ただし、日時データ型とインターバル・データ型は例外です。これらのデータ型では、以下のようになります。

- SQL_DESC_TYPE は詳細型 (SQL_DATETIME) を格納します。
- SQL_DESC_CONCISE_TYPE は簡易型を格納します。

フィールドの設定および他のフィールドに対する設定の効果について詳しくは、Microsoft ODBC の Web サイトで SQLSetDescField 関数の説明を参照してください。

いくつかのデータ型用に SQL_DESC_TYPE フィールドまたは SQL_DESC_CONCISE_TYPE フィールドを設定すると、以下のフィールドは、そのデータ型に対して適切なデフォルト値に設定されます。

- SQL_DESC_DATETIME_INTERVAL_PRECISION
- SQL_DESC_LENGTH
- SQL_DESC_PRECISION
- SQL_DESC_SCALE

詳しくは、Microsoft ODBC の Web サイトで SQLSetDescField 関数の SQL_DESC_TYPE フィールドを参照してください。

注: 設定されたデフォルト値が適切ではない場合、アプリケーションで SQLSetDescField を呼び出すことにより、明示的に記述子フィールドを設定できます。

以下の表に、各 SQL 型 ID および C 型 ID に関して、日時ごとに簡易型 ID、詳細型 ID、および型のサブコードをリストします。

日時データ型に関して、SQL_DESC_TYPE は、SQL データ型 (インプリメンテーション記述子内) と C データ型 (アプリケーション記述子内) の両方に対して同じマニフェスト定数を持ちます。

表 187. 日時ごとの簡易型 ID、詳細型 ID、および型のサブコード

簡易 SQL 型	簡易 C 型	詳細型	DATETIME_INTERVAL_CODE (「型のサブコード」とも呼ばれる)
SQL_TYPE_DATE	SQL_C_TYPE_DATE	SQL_DATETIME	SQL_CODE_DATE
SQL_TYPE_TIME	SQL_C_TYPE_TIME	SQL_DATETIME	SQL_CODE_TIME
SQL_TYPE_TIMESTAMP	SQL_C_TYPE_TIMESTAMP	SQL_DATETIME	SQL_CODE_TIME STAMP

疑似型 ID

ODBC は疑似型 ID をいくつか定義しており、状態に応じて、既存のデータ型に解決されます。これらの ID は、実際のデータ型には対応していませんが、アプリケーション・プログラミングの便宜のために提供されています。

小数桁数

小数桁数は 10 進数と数値のデータ型に適用されます。これは、小数点以下の最大桁数、すなわちデータの位取りを指しています。

小数点以下の桁数は固定されていないため、近似浮動小数点数の列やパラメーターに対する位取りは定義されていません。日時データに秒コンポーネントが含まれている場合は、小数桁数はデータの秒コンポーネントの小数点以下の桁数です。

通常、最大位取りは SQL_DECIMAL および SQL_NUMERIC のデータ型の最大精度と一致します。ただし、一部のデータ・ソースには独自の最大位取り制限があり

ます。アプリケーションは SQLGetTypeInfo を呼び出して、データ型に認められている最小および最大の位取りを判別できます。

以下の ODBC 関数は、SQL ステートメント・データ型のパラメーター 10 進数属性を返すか、またはデータ・ソースの 10 進数属性を返します。

表 188. ODBC 関数の戻りパラメーター

ODBC 関数	**戻り値**
SQLDescribeCol	記述する列の小数桁数
SQLDescribeParam	記述するパラメーターの小数桁数
SQLProcedureColumns	プロシージャーの列の小数桁数
SQLColumns	指定された表 (基本表、ビュー、またはシステム表など) の小数桁数
SQLColAttribute	データ・ソースの列の小数桁数
SQLGetTypeInfo	データ・ソース上の SQL データ型の最小および最大の小数桁数

注: SQLBindParameter は SQL ステートメントのパラメーターの小数桁数を設定します。

ODBC 関数が小数桁数について返す値は、ODBC 2.x で定義されている「位取り」に対応しています。

記述子フィールドは、結果セットの特性を記述します。ステートメントの実行前は有効なデータ値は含まれていません。ただし、SQLColumns、SQLProcedureColumns、および SQLGetTypeInfo によって返される小数桁数の値は、データ・ソースのカタログからの表の列やデータ型など、データベース・オブジェクトの特性を示すものです。

簡易 SQL データ型ごとに、以下の表に記載の小数桁数定義があります。

表 189. SQL データ型小数桁数

SQL 型 ID	**小数桁数**
すべての文字型とバイナリー型 [a]	N/A

表 189. SQL データ型小数桁数 (続き)

SQL 型 ID	小数桁数
SQL_DECIMAL SQL_NUMERIC	定義された小数点以下の桁数。例えば、NUMERIC(10,3) と定義されている列の位取りは 3 です。(インプリメンテーションによっては負の数値を使用して、指数表記を使用せずに非常に大きな数値の格納をサポートしている場合もあります。例えば、「12000」は -3 の位取りで「12」として格納することができます。ただし、solidDB は負の位取りはサポートしていません。)
SQL_DECIMAL および SQL_NUMERIC 以外のすべての厳密な数値型 [a]	0
すべての近似データ型 [a]	N/A
注: [a] このデータ型では、SQLBindParameter の DecimalDigits 引数は無視されます。	

小数桁数については、返された値は、いずれか 1 つの記述子フィールドの値に対応しているわけではありません。小数桁数について返される値 (例えば SQLColAttribute での値) は、以下の表に示すように、データ型によって、SQL_DESC_SCALE フィールドまたは SQL_DESC_PRECISION フィールドの値のいずれかになります。

表 190. 小数桁数に対応する記述子フィールド

SQL 型 ID	小数桁数に対応する記述子フィールド
すべての文字型とバイナリー型	N/A
すべての厳密な数値型	SCALE
すべての近似数値型	N/A
すべての日時型	PRECISION

転送オクテット長

データをデフォルト C データ型に転送すると、アプリケーションは最大バイト数を受信します。この最大値は列の転送オクテット長と呼ばれています。

文字データの場合は、NULL 終了文字のスペースは転送オクテット長に含まれません。バイト数で表した転送オクテット長は、データ・ソースにデータを格納するために必要なバイト数とは異なることがある点に注意してください。

以下の ODBC 関数は、SQL ステートメント・データ型のパラメーター 10 進数属性を返すか、またはデータ・ソースの 10 進数属性を返します。

表 191. ODBC 関数の戻りパラメーター 10 進数属性

ODBC 関数	戻り値
SQLColumns	指定された表 (基本表、ビュー、またはシステム表など) の列の転送オクテット長
SQLColAttribute	データ・ソースの列の転送オクテット長
SQLProcedureColumns	プロシージャーの列の転送オクテット長

ODBC 関数が転送オクテット長について返す値は、SQL_DESC_LENGTH で返される値には対応していないことがあります。すべての文字型およびバイナリー型については、値は記述子フィールドの SQL_DESC_OCTET_LENGTH の値です。その他のデータ型については、この情報を格納する記述子フィールドはありません。

記述子フィールドは、結果セットの特性を記述します。ステートメントの実行前は有効なデータ値は含まれていません。結果セット内では、SQLColAttribute はデータ・ソースの列の転送オクテット長を返します。これらの値は、SQL_DESC_OCTET_LENGTH 記述子フィールドの値と一致しないことがあります。記述子フィールドについて詳しくは、Microsoft ODBC の Web サイトで SQLSetDescField 関数に関する説明を参照してください。

簡易 SQL データ型ごとに、以下の表に記載の転送オクテット長定義があります。

表 192. 転送オクテット長

SQL 型 ID	転送オクテット長
すべての文字型とバイナリー型 [a]	バイト数で表した、列の定義済みの長さまたは最大の長さ (可変型の場合)。この値は、SQL_DESC_OCTET_LENGTH 記述子フィールドの値と一致します。
SQL_DECIMAL SQL_NUMERIC	文字セットが ASCII の場合は、このデータの文字表現を格納するために必要なバイト数。文字セットが UNICODE の場合は、その倍のバイト数になります。文字表現は、最大桁数に 2 を足したものです。データは文字ストリングとして返されますが、ここで桁、符号、小数点の分の文字が必要です。例えば、NUMERIC(10,3) として定義された列の転送長は 12 です。これは、桁に 10 バイト、符号に 1 バイト、小数点に 1 バイトがあるからです。
SQL_TINYINT	1
SQL_SMALLINT	2
SQL_INTEGER	4

表 192. 転送オクテット長 (続き)

SQL 型 ID	転送オクテット長
SQL_BIGINT	文字セットが ASCII の場合は、このデータの文字表現を格納するために必要なバイト数。文字セットが UNICODE の場合は、その倍のバイト数になります。デフォルトでは、このデータ型は文字ストリングとして返されます。文字表現は 19 桁と符号 (符号付きの場合)、または 20 桁 (符号なしの場合) に対する 20 文字から構成されます。長さは 20 です。 solidDB では、BIGINT は符号付きのもののみをサポートし、符号なしのものをサポートしません。
SQL_REAL	4
SQL_FLOAT	8
SQL_DOUBLE	8
すべてのバイナリー型 [a]	定義済みの数 (固定型の場合) または最大の数 (可変型の場合) の文字を格納するために必要なバイト数。
SQL_TYPE_DATE SQL_TYPE_TIME	6 (構造体 SQL_DATE_STRUCT または SQL_TIME_STRUCT のサイズ)
SQL_TYPE_TIMESTAMP	16 (構造体 SQL_TIMESTAMP_STRUCT のサイズ)

上記の表の脚注番号に関する説明

[a] ドライバーが可変型の列またはパラメーターの長さを判別できない場合は、SQL_NO_TOTAL が返されます。

グレゴリオ暦の制約

日付および日時のデータ型に関するグレゴリオ暦の制約を以下の表に示します。

表 193. グレゴリオ暦の制約

値	要件
月フィールド	1 から 12 までの値である必要があります。
日フィールド	範囲は 1 からその月の日数までである必要があります。月の日数は年および月のフィールドの値で決まり、28、29、30、または 31 のいずれかになります。うるう年も、月の日数に影響します。
時フィールド	0 から 23 までの値である必要があります。
分フィールド	0 から 59 までの値である必要があります。

表 193. グレゴリオ暦の制約 (続き)

値	要件
末尾秒フィールド	0 から 61.9(n) までの値である必要があります。n は「9」の位置の桁数を指定するもので、n の値は小数秒精度です。秒の範囲には、恒星時間との同期を維持するため、最大 2 秒のうるう秒が許可されています。

SQL から C データ型へのデータ変換

このセクションでは、SQL から C データ型へのデータ変換に関する情報を提供します。

アプリケーションが SQLFetch、SQLFetchScroll、または SQLGetData を呼び出すと、ドライバーはデータ・ソースからデータをリトリーブします。ドライバーは、必要であれば、ドライバーがリトリーブしたときのデータ型から、SQLBindCol または SQLGetData の TargetType 引数で指定されたデータ型にデータを変換します。最後に、SQLBindCol または SQLGetData (および ARD の SQL_DESC_DATA_PTR フィールド) の TargetValuePtr 引数が指す場所にデータを格納します。

以下の表には、ODBC SQL データ型から ODBC C データ型へのサポートされている変換を示します。アスタリスクは SQL データ型のデフォルト変換を示します (TargetType の値が SQL_C_DEFAULT の場合に、データの変換先になる C データ型です)。白丸はサポートされている変換を示します。

ODBC 2.x ドライバーと連携している ODBC 3.x アプリケーションでは、ドライバー固有のデータ型からの変換がサポートされない場合があります。

変換されたデータのフォーマットは、Microsoft Windows の国別設定には影響されません。

solidDB では、整数データ型 (SQL_TINYINT、SQL_SMALLINT、SQL_INTEGER、および SQL_BIGINT) は符号付きのもののみをサポートし、符号なしのものをサポートしません。符号なし C 変数を符号付き SQL 列にバインドできますが、格納する値が両方のデータ型にサポートされている範囲内にあることを確認する必要があります。

solidDB は SQL 列の BIT/SQL_BIT データ型をサポートしていません。ただし、C アプリケーションでは、数値 SQL 列を BIT データ型にバインドすることができます。例えば、データベースで TINYINT 列を使用して、その列を SQL_C_BIT 型の C 変数にバインドすることができます。solidDB ODBC ドライバーはデータベース内の数値型を、C 変数用に BIT データ型に変換することを試みます。数値データ値は、1、0、または NULL でなければなりません。他の値の場合、データ変換エラーが発生します。以下の表では、BIT/SQL_BIT データ型については説明していません。

注意:
この表には、符号なしデータ型を伴う変換を含む広範な ODBC 変換が示されていますが、solidDB は符号付き整数データ型 (例えば TINYINT、 SMALLINT、INTEGER、および BIGINT など) のみをサポートしています。

表 194. C データ型: データ型が以下の SQL_C_datatype

SQL データ型	CHAR	WCHAR	STINYINT	UTINYINT	TINYINT	SSHORT	USHORT	SHORT	SLONG	ULONG	LONG	SBIGINT	UBIGINT	FLOAT	DOUBLE	NUMERIC	BINARY	DATE	TIME	TIMESTAMP
SQL_CHAR	*	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o
SQL_VARCHAR	*	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o
SQL_LONGVARCHAR	*	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o
SQL_WCHAR	o	*	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o
SQL_WVARCHAR	o	*	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o
SQL_WLONGVARCHAR	o	*	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o
SQL_TINYINT (符号付き)	o	o	*	o	o	o	o	o	o	o	o	o	o	o	o	o	o			
SQL_TINYINT (符号なし)	o	o	o	*	o	o	o	o	o	o	o	o	o	o	o	o	o			
SQL_SMALLINT (符号付き)	o	o	o	o	o	*	o	o	o	o	o	o	o	o	o	o	o			
SQL_SMALLINT (符号なし)	o	o	o	o	o	o	*	o	o	o	o	o	o	o	o	o	o			
SQL_INTEGER (符号付き)	o	o	o	o	o	o	o	o	*	o	o	o	o	o	o	o	o			
SQL_INTEGER (符号なし)	o	o	o	o	o	o	o	o	o	*	o	o	o	o	o	o	o			
SQL_BIGINT (符号付き)	o	o	o	o	o	o	o	o	o	o	o	*	o	o	o	o				
SQL_BIGINT (符号なし)	o	o	o	o	o	o	o	o	o	o	o	o	*	o	o	o	o			
SQL_REAL	o	o	o	o	o	o	o	o	o	o	o	o	o	*	o	o	o			
SQL_FLOAT	o	o	o	o	o	o	o	o	o	o	o	o	o	o	*	o	o			
SQL_DOUBLE	o	o	o	o	o	o	o	o	o	o	o	o	o	o	*	o	o			
SQL_DECIMAL	*	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o			
SQL_NUMERIC	*	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o			

表 194. C データ型: データ型が以下の SQL_C_datatype (続き)

SQL データ型	CHAR	WCHAR	STINYINT	UTINYINT	TINYINT	SSHORT	USHORT	SHORT	SLONG	ULONG	LONG	SBIGINT	UBIGINT	FLOAT	DOUBLE	NUMERIC	BINARY	DATE	TIME	TIMESTAMP
SQL_BINARY	o	o															*			
SQL_VARBINARY	o	o															*			
SQL_LONGVARBINARY	o	o															*			
SQL_TYPE_DATE	o	o															o	*		o
SQL_TYPE_TIME	o	o															o		*	o
SQL_TYPE_TIMESTAMP	o	o															o	o	o	*

* これらのデータ型は、データ型の名前に「TYPE」の語を含んでいます。例えば、SQL_C_TYPE_DATE、SQL_C_TYPE_TIME、および SQL_C_TYPE_TIMESTAMP のようになります。

凡例:

* デフォルトの変換

o サポートされている変換

SQL から C へのデータ変換表

以下のセクションの表では、ドライバーまたはデータ・ソースがデータ・ソースからリトリーブしたデータを変換する方法を示します。ドライバーは、サポートしている ODBC SQL データ型から、すべての ODBC C データ型への変換をサポートする必要があります。

変換表の説明 (SQL から C への変換)

表には以下の列が含まれています。

- 特定の ODBC SQL データ型では、表の最初の列には、SQLBindCol および SQLGetData の TargetType 引数の有効な入力値がリストされています。
- 2 番目の列にはテストの結果がリストされています。このテストでは、SQLBindCol または SQLGetData で指定された BufferLength 引数が使用されることが多く、ドライバーは、データを変換できるかどうかを判断するためにこのテストを行います。
- 結果ごとに、3 番目と 4 番目の列には、ドライバーがデータの変換を試みた後に、バッファーに入れられた値がリストされています。これらのバッファーは、

SQLBindCol または SQLGetData で指定された TargetValuePtr および StrLen_or_IndPtr の引数によって指定されます。(StrLen_or_IndPtr 引数は、ARD の SQL_DESC_OCTET_LENGTH_PTR フィールドに対応しています。)

- 最後の列には、SQLFetch、SQLFetchScroll、または SQLGetData により結果ごとに返される SQLSTATE がリストされます。

SQLBindCol または SQLGetData の TargetType 引数に、特定の ODBC SQL データ型について表には記載されていない ODBC C データ型の値が含まれている場合、SQLFetch、SQLFetchScroll、または SQLGetData は SQLSTATE 07006 (制限付きデータ型属性違反) を返します。TargetType 引数にドライバー固有の SQL データ型から ODBC C データ型への変換を指定する値が含まれており、ドライバーがこの変換をサポートしていない場合は、SQLFetch、SQLFetchScroll、または SQLGetData は SQLSTATE HYC00 (オプション機能がインプリメントされない) を返します。

表には記載されていませんが、ドライバーは SQL データ値が NULL の場合は、StrLen_or_IndPtr 引数によって指定されたバッファーに、SQL_NULL_DATA を返します。StrLen_or_IndPtr で指定される長さには、NULL 終了バイトは含まれません。TargetValuePtr が NULL ポインターの場合は、SQLGetData は、SQLSTATE HY009 (NULL ポインターの無効な使用) を返します。SQLBindCol では、これによって列がアンバインドされます。

これらの表では以下の用語と規則を使用します。

- データがアプリケーションに返される前に切り捨てられるかどうかに関係なく、データのバイト長は *TargetValuePtr で返すことのできる C データのバイト数です。ストリング・データの場合、これには NULL 終了文字のスペースは含まれません。
- 文字バイト長は、データを文字フォーマットで表示するために必要な合計バイト数です。
- イタリックで記載した語は、SQL 文法の関数引数または要素を示します。文法要素の構文については、 267 ページの『付録 D. ODBC に関する SQL 文法の最小要件』を参照してください。

SQL から C への変換: 文字

文字 ODBC SQL データ型は、以下のとおりです。

```
SQL_CHAR
SQL_VARCHAR
SQL_LONGVARCHAR
SQL_WCHAR
SQL_WVARCHAR
SQL_WLONGVARCHAR
```

以下の表には、文字 SQL データから変換することができる ODBC C データ型を記載します。表中の列および用語の説明は、 299 ページの『変換表の説明 (SQL から C への変換)』を参照してください。

表 195. 文字 SQL データから ODBC C データ型への変換

C 型 ID	テスト	*TargetValuePtr	*StrLen_or_IndPtr	SQLSTATE
SQL_C_CHAR	データのバイト長 < BufferLength データのバイト長 >= BufferLength	データ 切り捨てデータ	データの長さ (バイト数) データの長さ (バイト数)	N/A 01004
SQL_C_WCHAR	データの文字長 < BufferLength (データの文字長) >= BufferLength	データ 切り捨てデータ	データの長さ (文字数) データの長さ (文字数)	N/A 01004
EXACT NUMERIC TYPES [h] SQL_C_STINYINT SQL_C_UTINYINT SQL_C_TINYINT SQL_C_SSHORT SQL_C_USHORT SQL_C_SHORT SQL_C_SLONG SQL_C_ULONG SQL_C_LONG SQL_C_SBIGINT SQL_C_UBIGINT SQL_C_NUMERIC	切り捨てなしで変換したデータ [b] 小数桁を切り捨てて変換したデータ [a] データ変換の結果、(小数桁ではなく) 整数桁が失われる [b] データは数値リテラルではない [b]	データ 切り捨てデータ 未定義 未定義	C データ型のバイト数 C データ型のバイト数 未定義 未定義	N/A 01S07 22003 22018
APPROXIMATE NUMERIC TYPES [h] SQL_C_FLOAT SQL_C_DOUBLE	データは数値が変換されるデータ型の範囲内 [a] データは数値が変換されるデータ型の範囲外 [a] データは数値リテラルではない [b]	データ 未定義 未定義	C データ型のサイズ 未定義 未定義	N/A 2003 22018
SQL_C_BINARY	データのバイト長 <= BufferLength データのバイト長 > BufferLength	データ 切り捨てデータ	データの長さ データの長さ	N/A 01004

表 195. 文字 SQL データから ODBC C データ型への変換 (続き)

C 型 ID	テスト	*TargetValuePtr	*StrLen_or_IndPtr	SQLSTATE
SQL_C_TYPE_DATE	データ値は有効な日付値 [a]	データ	6 [b]	N/A
	データ値は有効なタイム・スタンプ値で、時刻部分はゼロ [a]	データ	6 [b]	N/A
		切り捨てデータ	6 [b]	01S07
	データ値は有効なタイム・スタンプ値で、時刻部分はゼロ以外 [a]、[c]	未定義	未定義	22018
	データ値は有効な日付値またはタイム・スタンプ値ではない [a]			
SQL_C_TYPE_TIME	データ値は有効な時刻値で、小数秒値は 0 [a]	データ	6 [b]	N/A
		データ	6 [b]	N/A
	データ値は有効なタイム・スタンプ値または有効な時刻値で、小数秒部分はゼロ [a]、[d]	切り捨てデータ	6 [b]	01S07
		未定義	未定義	22018
	データ値は有効なタイム・スタンプ値で、小数秒部分はゼロ以外 [a]、[d]、[e]			
	データ値は有効なタイム・スタンプ値または時刻値ではない [a]			
SQL_C_TYPE_TIMESTAMP	データ値は有効なタイム・スタンプ値または有効な時刻値で、小数秒部分は切り捨てられない [a]、[d]	データ	16 [b]	N/A
		切り捨てデータ	16 [b]	01S07
	データ値は有効なタイム・スタンプ値または有効な時刻値で、小数秒部分は切り捨てられる [a]	データ [f]	16 [b]	N/A
		データ [g]	16 [b]	N/A
		未定義	未定義	22018
	データ値は有効な日付値 [a]			
	データ値は有効な時刻値 [a]			
	データ値は有効な日付値、時刻値、またはタイム・スタンプ値ではない [a]			

表 195. 文字 SQL データから ODBC C データ型への変換 (続き)

C 型 ID	テスト	*TargetValuePtr	*StrLen_or_IndPtr	SQLSTATE
注: [a] BufferLength の値はこの変換では無視されます。ドライバーは、*TargetValuePtr のサイズが C データ型のサイズであると想定します。 [b] これは対応する C データ型のサイズです。 [c] タイム・スタンプ値の時刻部分は切り捨てられます。 [d] タイム・スタンプ値の日付部分は無視されます。 [e] タイム・スタンプの小数秒部分は切り捨てられます。 [f] タイム・スタンプ構造体の時刻フィールドはゼロに設定されます。 [g] タイム・スタンプ構造体の日付フィールドは現在日付に設定されます。 [h] 厳密な数値型には、NUMERIC/DECIMAL と整数が含まれます。これらのデータ型は、データ型の精度範囲内であれば、指定した値をそのまま格納します。近似データ型には FLOAT/REAL が含まれており、指定した値の近似値のみを格納します (場合によっては、最下位桁が、実際に指定したものとは若干異なることがあります)。				

文字 SQL データを数値、日付、時刻、またはタイム・スタンプの C データに変換すると、前後のスペースが無視されます。

SQL から C への変換: 数値

```
SQL_DECIMAL     SQL_BIGINT
SQL_NUMERIC     SQL_REAL
SQL_TINYINT     SQL_FLOAT
SQL_SMALLINT    SQL_DOUBLE
SQL_INTEGER
```

以下の表には、数値 SQL データから変換することができる ODBC C データ型を記載します。表中の列および用語の説明は、 299 ページの『変換表の説明 (SQL から C への変換)』を参照してください。

表 196. 数値 SQL データから ODBC C データ型への変換

C 型 ID	テスト	*TargetValuePtr	*StrLen_or_IndPtr	SQLSTATE
SQL_C_CHAR	文字のバイト長 < BufferLength (小数桁ではなく) 整数桁の数 < BufferLength (小数桁ではなく) 整数桁の数 ≥ BufferLength	データ 切り捨てデータ 未定義	データの長さ (バイト数) データの長さ (バイト数) 未定義	N/A 01004 22003

表 196. 数値 SQL データから ODBC C データ型への変換 (続き)

C 型 ID	テスト	*TargetValuePtr	*StrLen_or_IndPtr	SQLSTATE
SQL_C_WCHAR	文字の長さ < BufferLength (小数桁ではなく) 整数桁の数 < BufferLength (小数桁ではなく) 整数桁の数 ≥ BufferLength	データ 切り捨てデータ 未定義	データの長さ (バイト数) データの長さ (バイト数) 未定義	N/A 01004 22003
EXACT NUMERIC TYPES [c] SQL_C_STINYINT SQL_C_UTINYINT SQL_C_TINYINT SQL_C_SBIGINT SQL_C_UBIGINT SQL_C_SSHORT SQL_C_USHORT SQL_C_SHORT a SQL_C_SLONG SQL_C_ULONG SQL_C_LONG SQL_C_NUMERIC	切り捨てなしで変換したデータ [a] 小数桁を切り捨てて変換したデータ [a] データ変換の結果、(小数桁ではなく) 整数桁が失われる [a]	データ 切り捨てデータ 未定義	C データ型のサイズ C データ型のサイズ 未定義	N/A 01S07 22003
APPROXIMATE NUMERIC TYPES [c] SQL_C_FLOAT SQL_C_DOUBLE	データは数値が変換されるデータ型の範囲内 [a] データは数値が変換されるデータ型の範囲外 [a]	データ 未定義	C データ型のサイズ 未定義	N/A 22003
SQL_C_BINARY	データの長さ ≤ BufferLength データの長さ > BufferLength	データ 未定義	データの長さ 未定義	N/A 22003

表 196. 数値 SQL データから ODBC C データ型への変換 (続き)

C 型 ID	テスト	*TargetValuePtr	*StrLen_or_IndPtr	SQLSTATE
注: [a] BufferLength の値はこの変換では無視されます。ドライバーは、*TargetValuePtr のサイズが C データ型のサイズであると想定します。 [b] これは対応する C データ型のサイズです。 [c] 厳密な数値型には、NUMERIC/DECIMAL と整数が含まれます。これらのデータ型は、データ型の精度範囲内であれば、指定した値をそのまま格納します。近似データ型には FLOAT/REAL が含まれており、指定した値の近似値のみを格納します (場合によっては、最下位桁が、実際に指定したものとは若干異なることがあります)。				

SQL から C への変換: バイナリー

バイナリー ODBC SQL データ型は、以下のとおりです。

```
SQL_BINARY
SQL_VARBINARY
SQL_LONGVARBINARY
```

以下の表には、バイナリー SQL データから変換することができる ODBC C データ型を記載します。表中の列および用語の説明は、 299 ページの『変換表の説明 (SQL から C への変換)』を参照してください。

表 197. バイナリー SQL データから ODBC C データ型への変換

C 型 ID	テスト	*TargetValuePtr	*StrLen_or_IndPtr	SQLSTATE
SQL_C_CHAR	(データのバイト長) * 2 < BufferLength (データのバイト長) * 2 >= BufferLength	データ 切り捨てデータ	データの長さ (バイト数) データの長さ (バイト数)	N/A 01004
SQL_C_WCHAR	(データの文字長) * 2 < BufferLength (データの文字長) * 2 >= BufferLength	データ 切り捨てデータ	データの長さ (バイト数) データの長さ (バイト数)	N/A 01004
SQL_C_BINARY	データのバイト長 <= BufferLength データのバイト長 > BufferLength	データ 切り捨てデータ	データの長さ (バイト数) データの長さ (バイト数)	N/A 01004

バイナリー SQL データを文字の C データに変換すると、ソース・データの各バイト (8 ビット) は 2 つの ASCII 文字として表現されます。これらの文字は、16 進形式での数値の ASCII 文字表現です。例えば、バイナリー 00000001 は「01」に、バイナリー 11111111 は「FF」に変換されます。

ドライバーは常に個々のバイトを 16 進数字のペアに変換し、NULL バイトで文字ストリングを終了します。このため、BufferLength が偶数で、変換後のデータの長さよりも短い場合は、*TargetValuePtr バッファーの最後のバイトは使用されません。(変換済みのデータでは、偶数バイトが必要で、最後から 2 番目のバイトが NULL バイトであり、最後のバイトが使用できません。)

アプリケーション開発者には、文字 C データ型へのバイナリー SQL データのバインドを避けることを推奨します。この変換は普通、非効率で低速です。

SQL から C への変換: 日付

日付 ODBC SQL データ型は、以下のとおりです。

```
SQL_DATE
```

以下の表には、日付 SQL データから変換することができる ODBC C データ型を記載します。表中の列および用語の説明は、 299 ページの『変換表の説明 (SQL から C への変換)』を参照してください。

表 198. 日付 SQL データから ODBC C データ型への変換

C 型 ID	テスト	*TargetValuePtr	*StrLen_or_IndPtr	SQLSTATE
SQL_C_CHAR	BufferLength > 文字のバイト長 11 <= BufferLength <= 文字のバイト長 BufferLength < 11	データ 切り捨てデータ 未定義	10 データの長さ (バイト数) 未定義	N/A 01004 22003
SQL_C_WCHAR	BufferLength > 文字の長さ 11 <= BufferLength <= 文字の長さ BufferLength < 11	データ 切り捨てデータ 未定義	10 データの長さ (バイト数) 未定義	N/A 01004 22003
SQL_C_BINARY	データのバイト長 <= BufferLength データのバイト長 > BufferLength	データ 未定義	データの長さ (バイト数) 未定義	N/A 22003
SQL_C_DATE	なし [a]	データ	6 [c]	N/A
SQL_C_TIMESTAMP	なし [a]	データ [b]	16 [c]	N/A

注:

[a] BufferLength の値はこの変換では無視されます。ドライバーは、*TargetValuePtr のサイズが C データ型のサイズであると想定します。

[b] タイム・スタンプ構造体の時刻フィールドはゼロに設定されます。

[c] これは対応する C データ型のサイズです。

日付 SQL データを文字 C データに変換すると、結果ストリングは「yyyy-mm-dd」フォーマットになります。このフォーマットは Microsoft Windows の国別設定には影響されません。

SQL から C への変換: 時刻

時刻 ODBC SQL データ型は、以下のとおりです。

```
SQL_TIME
```

以下の表には、時刻 SQL データから変換することができる ODBC C データ型を記載します。表中の列および用語の説明は、 299 ページの『変換表の説明 (SQL から C への変換)』を参照してください。

表 199. 時刻 SQL データから ODBC C データ型への変換

C 型 ID	テスト	*TargetValuePtr	*StrLen_or_IndPtr	SQLSTATE
SQL_C_CHAR	BufferLength > 文字のバイト長 9 <= BufferLength <= 文字のバイト長 BufferLength < 9	データ 切り捨てデータ [a] 未定義	データの長さ (バイト数) データの長さ (バイト数) 未定義	N/A 01004 22003
SQL_C_WCHAR	BufferLength > 文字のバイト長 9 <= BufferLength <= 文字のバイト長 BufferLength < 9	データ 切り捨てデータ [a] 未定義	データの長さ (文字数) データの長さ (文字数) 未定義	N/A 01004 22003
SQL_C_BINARY	データのバイト長 <= BufferLength データのバイト長 > BufferLength	データ 未定義	データの長さ (バイト数) 未定義	N/A 22003
SQL_C_DATE	なし [a]	データ	6 [c]	N/A
SQL_C_TIMESTAMP	なし [a]	データ [b]	16 [c]	N/A

注:

[a]: 時刻の小数秒は切り捨てられます。

[b]: BufferLength の値はこの変換では無視されます。ドライバーは、*TargetValuePtr のサイズが C データ型のサイズであると想定します。

[c]: タイム・スタンプ構造体の日付フィールドは、現在日付に設定され、タイム・スタンプ構造体の小数秒フィールドはゼロに設定されます。

[d]: これは対応する C データ型のサイズです。

時刻 SQL データを文字の C データに変換すると、結果ストリングは「hh:mm:ss」フォーマットになります。

SQL から C への変換: タイム・スタンプ

タイム・スタンプ ODBC SQL データ型は、以下のとおりです。

```
SQL_TIMESTAMP
```

以下の表には、タイム・スタンプ SQL データから変換することができる ODBC C データ型を記載します。表中の列および用語の説明は、 299 ページの『変換表の説明 (SQL から C への変換)』を参照してください。

表 200. タイム・スタンプ SQL データから ODBC C データ型への変換

C 型 ID	テスト	*TargetValuePtr	*StrLen_or_IndPtr	SQLSTATE
SQL_C_CHAR	BufferLength > 文字のバイト長 20 <= BufferLength <= 文字のバイト長 BufferLength < 20	データ 切り捨てデータ [b] 未定義	データの長さ (バイト数) データの長さ (バイト数) 未定義	N/A 01004 22003
SQL_C_WCHAR	BufferLength > 文字のバイト長 20 <= BufferLength <= 文字のバイト長 BufferLength < 20	データ 切り捨てデータ [b] 未定義	データの長さ (文字数) データの長さ (文字数) 未定義	N/A 01004 22003
SQL_C_BINARY	データのバイト長 <= BufferLength データのバイト長 > BufferLength	データ 未定義	データの長さ (バイト数) 未定義	N/A 22003
SQL_C_TYPE_DATE	タイム・スタンプの時刻部分はゼロ [a] タイム・スタンプの時刻部分はゼロ以外 [a]	データ 切り捨てデータ [c]	6 [f] 6 [f]	N/A 01S07
SQL_C_TYPE_TIME	タイム・スタンプの小数秒部分はゼロ [a] タイム・スタンプの小数秒部分はゼロ以外 [a]	データ [d] 切り捨てデータ [d]、[e]	6 [f] 6 [f]	N/A 01S07
SQL_C_TYPE_TIMESTAMP	タイム・スタンプの小数秒部分は切り捨てられない [a] タイム・スタンプの小数秒部分は切り捨てられる [a]	データ [e] 切り捨てデータ [e]	6 [f] 6 [f]	N/A 01S07

表 200. タイム・スタンプ SQL データから ODBC C データ型への変換 (続き)

C 型 ID	テスト	*TargetValuePtr	*StrLen_or_IndPtr	SQLSTATE
注:				
[a] BufferLength の値はこの変換では無視されます。ドライバーは、*TargetValuePtr のサイズが C データ型のサイズであると想定します。				
[b] タイム・スタンプの小数秒は切り捨てられます。				
[c] タイム・スタンプの時刻部分は切り捨てられます。				
[d] タイム・スタンプの日付部分は無視されます。				
[e] タイム・スタンプの小数秒部分は切り捨てられます。				
[f] これは対応する C データ型のサイズです。				

タイム・スタンプ SQL データを文字 C データに変換すると、結果ストリングは「yyyy-mm-dd hh:mm:ss [.f ...]」フォーマットになります。ここで、小数秒には最大 9 桁を使用できます。このフォーマットは Microsoft Windows の国別設定には影響されません。(小数点と小数秒を除いて、タイム・スタンプ SQL データ型の精度に関係なく、フォーマット全体を使用する必要があります。)

SQL から C へのデータ変換例

以下の例では、ドライバーによる SQL データから C データへの変換方法を示します。

表 201. SQL から C へのデータ変換例

SQL 型 ID	SQL データ値	C 型 ID	バッファー長	*TargetValuePtr	SQLSTATE
SQL_CHAR	abcdef	SQL_C_CHAR	7	abcdef¥0 [a]	N/A
SQL_CHAR	abcdef	SQL_C_CHAR	6	abcde¥0 [a]	01004
SQL_DECIMAL	1234.56	SQL_C_CHAR	8	1234.56¥0 [a]	N/A
SQL_DECIMAL	1234.56	SQL_C_CHAR	5	1234¥0 [a]	01004
SQL_DECIMAL	1234.56	SQL_C_CHAR	4	----	22003
SQL_DECIMAL	1234.56	SQL_C_FLOAT	無視	1234.56	N/A
SQL_DECIMAL	1234.56	SQL_C_SSHORT	無視	1234	01S07
SQL_DECIMAL	1234.56	SQL_C_STINYINT	無視	----	22003
SQL_DOUBLE	1.2345678	SQL_C_DOUBLE	無視	1.2345678	N/A
SQL_DOUBLE	1.2345678	SQL_C_FLOAT	無視	1.234567	N/A

表 201. SQL から C へのデータ変換例 (続き)

SQL 型 ID	SQL データ値	C 型 ID	バッファー長	*TargetValuePtr	SQLSTATE
SQL_DOUBLE	1.2345678	SQL_C_STINYINT	無視	1	N/A
SQL_TYPE_DATE	1992-12-31	SQL_C_CHAR	11	1992-12-31¥0 [a]	N/A
SQL_TYPE_DATE	1992-12-31	SQL_C_CHAR	10	-----	22003
SQL_TYPE_DATE	1992-12-31	SQL_C_ TIMESTAMP	無視	1992,12,31, 0,0,0,0 [b]	N/A
SQL_TYPE_TIMESTAMP	1992-12-31 23:45:55.12	SQL_C_CHAR	23	1992-12-31 23:45:55.12¥0 [a]	N/A
SQL_TYPE_TIMESTAMP	1992-12-31 23:45:55.12	SQL_C_CHAR	22	1992-12-31 23:45:55.1¥0 [a]	01004
SQL_TYPE_TIMESTAMP	1992-12-31 23:45:55.12	SQL_C_CHAR	18	----	22003

[a] 「¥0」は NULL 終了バイトを表します。ドライバーは常に SQL_C_CHAR データを NULL で終了します。

[b] このリストの数値は、TIMESTAMP_STRUCT 構造体のフィールドに格納される数値です。

C から SQL データ型へのデータ変換

このセクションでは、C から SQL データ型へのデータ変換に関する情報を提供します。

アプリケーションが SQLExecute または SQLExecDirect を呼び出すと、ドライバーはアプリケーションのストレージ・ロケーションから SQLBindParameter にバインドされたパラメーターのデータをリトリーブします。実行時データのパラメーターについては、アプリケーションは SQLPutData によってパラメーター・データを送信します。必要であれば、ドライバーは SQLBindParameter の ValueType 引数で指定されたデータ型から、SQLBindParameter の ParameterType 引数で指定されたデータ型にデータを変換します。最後に、ドライバーはデータ・ソースにデータを送信します。

以下の表には、ODBC C データ型から ODBC SQL データ型へのサポートされている変換を示します。アスタリスクは SQL データ型のデフォルト変換を示します (ValueType または SQL_DESC_CONCISE_TYPE 記述子フィールドの値が SQL_C_DEFAULT の場合に、データの変換元になる C データ型です)。白丸はサポートされている変換を示します。

変換されたデータのフォーマットは、Microsoft Windows の国別設定には影響されません。

solidDB では、整数データ型 (SQL_TINYINT、SQL_SMALLINT、SQL_INTEGER、および SQL_BIGINT) は符号付きのもののみをサポートし、符号なしのものをサポートしません。符号なし C 変数を符号付き SQL 列にバインドできますが、格納する値が両方のデータ型にサポートされている範囲内にあることを確認する必要があります。

solidDB は SQL 列の BIT/SQL_BIT データ型をサポートしていません。ただし、C アプリケーションでは、数値 SQL 列を BIT データ型にバインドすることができます。例えば、データベースで TINYINT 列を使用して、その列を SQL_C_BIT 型の C 変数にバインドすることができます。solidDB ODBC ドライバーはデータベース内の数値型を、C 変数用に BIT データ型に変換することを試みます。数値データ値は、1、0、または NULL でなければなりません。他の値の場合、データ変換エラーが発生します。以下の表では、BIT/SQL_BIT データ型については説明していません。

注意:
以下の表には、符号なしデータ型を伴う変換を含む広範な ODBC 変換が示されていますが、solidDB は符号付き整数データ型 (例えば TINYINT、 SMALLINT、INT、および BIGINT など) のみをサポートしています。

表 202. SQL データ型: データ型が以下の SQL_datatype

C データ型	CHAR	VARCHAR	LONGVARCHAR	WCHAR	WVARCHAR	WLONGVARCHAR	DECIMAL	NUMERIC	TINYINT (符号付き)	TINYINT (符号なし)	SMALLINT (符号付き)	SMALLINT (符号なし)	INTEGER (符号付き)	INTEGER (符号なし)	BIGINT (符号付き)	BIGINT (符号なし)	REAL	FLOAT	DOUBLE	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
SQL_C_CHAR	*	*	*	o	o	o	*	*	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o
SQL_WCHAR	o	o	o	*	*	*	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o
SQL_C_NUMERIC	*	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o						
SQL_C_STINYINT	o	o	o	o	o	o	o	o	*	o	o	o	o	o	o	o	o	o	o						
SQL_C_UTINYINT	o	o	o	o	o	o	o	o	o	*	o	o	o	o	o	o	o	o	o						
SQL_C_TINYINT	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o						
SQL_C_SSHORT	o	o	o	o	o	o	o	o	o	o	*	o	o	o	o	o	o	o	o						
SQL_C_USHORT	o	o	o	o	o	o	o	o	o	o	o	*	o	o	o	o	o	o	o						
SQL_C_SHORT	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o						
SQL_C_SLONG	o	o	o	o	o	o	o	o	o	o	o	o	*	o	o	o	o	o	o						
SQL_C_ULONG	o	o	o	o	o	o	o	o	o	o	o	o	o	*	o	o	o	o	o						
SQL_C_LONG	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o						

表 202. SQL データ型: データ型が以下の SQL_datatype (続き)

C データ型	CHAR	VARCHAR	LONGVARCHAR	WCHAR	WVARCHAR	WLONGVARCHAR	DECIMAL	NUMERIC	TINYINT (符号付き)	TINYINT (符号なし)	SMALLINT (符号付き)	SMALLINT (符号なし)	INTEGER (符号付き)	INTEGER (符号なし)	BIGINT (符号付き)	BIGINT (符号なし)	REAL	FLOAT	DOUBLE	BINARY	VARBINARY	LONGVARBINARY	DATE	TIME	TIMESTAMP
SQL_C_SBIGINT	o	o	o	o	o	o	o	o	o	o	o	o	o	o	*	*	o	o	o						
SQL_C_UBIGINT	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	*	o	o	o						
SQL_C_FLOAT	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	*	o	o						
SQL_C_DOUBLE	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	*	*						
SQL_C_BINARY	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	o	*	*	*	o	o	o
SQL_C_DATE	o	o	o	o	o	o																	*		o
SQL_C_TIME	o	o	o	o	o	o																		*	o
SQL_C_TIMESTAMP	o	o	o	o	o	o																	o	o	*

凡例:

* デフォルトの変換

o サポートされている変換

C から SQL へのデータ変換表

以下のセクションの表では、ドライバーまたはデータ・ソースがデータ・ソースに送信されたデータを変換する方法を示します。ドライバーは、すべての ODBC C データ型から、サポートしている ODBC SQL データ型への変換をサポートする必要があります。

変換表の説明 (C から SQL への変換)

表には以下の列が含まれています。

- 特定の ODBC C データ型では、表の最初の列には、SQLBindParameter の ParameterType 引数の有効な入力値がリストされます。
- 2 番目の列には、ドライバーがデータを変換できるかどうかを判断するために行うテストの結果がリストされています。
- 3 番目の列には、SQLExecDirect、SQLExecute、または SQLPutData により結果ごとに返される SQLSTATE がリストされます。SQL_SUCCESS が返される場合にのみ、データがデータ・ソースに送信されます。

SQLBindParameter 内の ParameterType 引数に特定の C データ型について表には記載されていない ODBC SQL データ型の値が含まれている場合は、SQLBindParameter は SQLSTATE 07006 (制限付きデータ型属性違反) を返します。ParameterType 引数にドライバー固有の値が含まれており、ドライバーが特定の ODBC C データ型からそのドライバー固有の SQL データ型への変換をサポートしていない場合は、SQLBindParameter は SQLSTATE HYC00 (オプション機能がインプリメントされない) を返します。

SQLBindParameter で指定された ParameterValuePtr および StrLen_or_IndPtr の引数が両方とも NULL ポインターである場合、その関数は SQLSTATE HY009 (NULL ポインターの無効な使用) を返します。表には示されていませんが、アプリケーションは、SQLBindParameter の StrLen_or_IndPtr 引数が指す値、または StrLen_or_IndPtr 引数の値を SQL_NULL_DATA に設定して、NULL SQL データ値を指定します。(StrLen_or_IndPtr 引数は、APD の SQL_DESC_OCTET_LENGTH_PTR フィールドに対応しています。) アプリケーションはこれらの値を SQL_NTS に設定して、SQLBindParameter の *ParameterValuePtr、または (APD の SQL_DESC_DATA_PTR フィールドが指している) SQLPutData の *DataPtr の値を NULL 終了ストリングとして指定します。

表では以下の用語が使用されています。

- *データのバイト長* は、データ・ソースへの送信前にデータを切り捨てるかどうかに関係なく、データ・ソースに送信することができる SQL データのバイト数です。ストリング・データの場合は、これには NULL 終了文字は含まれません。
- *列のバイト長* は、データ・ソースでデータを格納するために必要なバイト数です。
- *文字のバイト長* は、文字形式でデータを表示するために必要な最大バイト数です。
- *桁数* は、負符号 (-)、小数点、指数 (必要な場合) など、数値を表現するために使用される文字数です。
- イタリックの語は、ODBC SQL 文法の要素を表します。文法要素の構文については、267 ページの『付録 D. ODBC に関する SQL 文法の最小要件』を参照してください。

C から SQL への変換: 文字

文字 ODBC C データ型は、以下のとおりです。

SQL_C_CHAR SQL_C_WCHAR

以下の表には、C 文字データから変換することができる ODBC SQL データ型を記載します。表中の列および用語の説明は、312 ページの『変換表の説明 (C から SQL への変換)』を参照してください。

注: 文字 C データを Unicode SQL データに変換する場合は、Unicode データ型の長さは偶数でなければなりません。

表 203. C 文字データから ODBC SQL データ型への変換

SQL 型 ID	テスト	SQLSTATE
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR	データのバイト長 <= 列の長さ データのバイト長 > 列の長さ	N/A 22001
SQL_WCHAR SQL_WVARCHAR SQL_WLONGVARCHAR	データの文字長 <= 列の長さ データの文字長 > 列の長さ	N/A 22001
SQL_DECIMAL SQL_NUMERIC SQL_TINYINT SQL_SMALLINT SQL_INTEGER SQL_BIGINT	切り捨てなしで変換したデータ 小数桁を切り捨てて変換したデータ [e] データ変換の結果、(小数桁ではなく) 整数桁が失われる [e] データ値は*数値リテラル* ではない	N/A 22001 22001 22018
SQL_REAL SQL_FLOAT SQL_DOUBLE	データは数値が変換されるデータ型の範囲内 データは数値が変換されるデータ型の範囲外 データ値は*数値リテラル* ではない	N/A 22003 22005
SQL_BIT	データは 0 または 1 データは 0 より大きく、2 未満で、1 とは等しくない データは 0 未満、または 2 以上 データは*数値リテラル* ではない 注: solidDB は SQL_BIT をサポートしていません。	N/A 22001 22003 22018
SQL_BINARY SQL_VARBINARY SQL_LONG-VARBINARY	(データのバイト長)/2 <= 列のバイト長 (データのバイト長)/2 > 列のバイト長 データ値は 16 進値ではない	N/A 22001 22018

表 203. C 文字データから ODBC SQL データ型への変換 (続き)

SQL 型 ID	テスト	SQLSTATE
SQL_TYPE_DATE	データ値は有効な *ODBC_date_literal*	N/A
	データ値は有効な *ODBC_timestamp_literal*。時刻部分はゼロ	N/A
	データ値は有効な *ODBC_timestamp_literal*。時刻部分はゼロ以外 [a]	22008
	データ値は有効な *ODBC_date_literal* または *ODBC_timestamp_literal* ではない	22018
SQL_TYPE_TIME	データ値は有効な *ODBC_time_literal*	N/A
	データ値は有効な *ODBC_timestamp_literal*。小数秒部分はゼロ [b]	N/A
	データ値は有効な *ODBC_timestamp_literal*。小数秒部分はゼロ以外 [b]	22008
	データ値は有効な *ODBC_time_literal* または *ODBC_timestamp_literal* ではない	22018
SQL_TYPE_TIMESTAMP	データ値は有効な *ODBC_timestamp_literal*。小数秒部分は切り捨てられない	N/A
	データ値は有効な *ODBC-timestamp-literal*。小数秒部分は切り捨てられる	22008
	データ値は有効な *ODBC-date-literal* [c]	N/A
	データ値は有効な *ODBC-time-literal* [d]	N/A
	データ値は有効な *ODBC-date-literal*、*ODBC-time-literal*、または *ODBC-timestamp-literal* ではない	22018

表 203. C 文字データから ODBC SQL データ型への変換 (続き)

SQL 型 ID	テスト	SQLSTATE
注: [a] タイム・スタンプの時刻部分は切り捨てられます。 [b] タイム・スタンプの日付部分は無視されます。 [c] タイム・スタンプの時刻部分はゼロに設定されます。 [d] タイム・スタンプの日付部分は現在日付に設定されます。 [e] ドライバー/データ・ソースは、変換実行を試みる前に、(SQLPutData を呼び出して、文字データを分割送信する場合でも) ストリング全体を受信するまで実際上は待機しています。		

文字 C データを数値、日付、時刻、またはタイム・スタンプの SQL データに変換すると、前後のブランクは無視されます。

文字 C データをバイナリー SQL データに変換すると、文字データの各 2 バイトはバイナリー・データの 1 バイト (8 ビット) に変換されます。文字データの各 2 バイトは、16 進形式の数値を表します。例えば、「01」はバイナリー 00000001 に、「FF」はバイナリー 11111111 に変換されます。

ドライバーは常に 16 進数字のペアを個々のバイトに変換して、NULL 終了バイトを無視します。このため、文字ストリングの長さが奇数の場合は、ストリングの最後のバイト (NULL 終了バイトが存在する場合はそのバイトを除く) は変換されません。

注: 文字 C データをバイナリー SQL データ型にバインドすることは非効率で低速なので、このようなバインドは避けてください。

C から SQL への変換: 数値

数値 ODBC C データ型は、以下のとおりです。

- SQL_C_STINYINT
- SQL_C_SLONG
- SQL_C_UTINYINT
- SQL_C_ULONG
- SQL_C_TINYINT
- SQL_C_LONG
- SQL_C_SSHORT
- SQL_C_FLOAT
- SQL_C_USHORT
- SQL_C_DOUBLE
- SQL_C_SHORT
- SQL_C_NUMERIC
- SQL_C_SBIGINT

- SQL_C_UBIGINT

SQL_C_TINYINT、SQL_C_SHORT、および SQL_C_LONG のデータ型について詳しくは、 283 ページの『C データ型』を参照してください。以下の表には、数値 C データから変換することができる ODBC SQL データ型を記載します。表中の列および用語の説明は、 312 ページの『変換表の説明 (C から SQL への変換)』を参照してください。

表 204. 数値 C データから ODBC SQL データ型への変換

SQL 型 ID	**テスト**	**SQLSTATE**
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR	桁数 <= 列のバイト長 桁数 > 列のバイト長	N/A 22001
SQL_WCHAR SQL_WVARCHAR SQL_WLONGVARCHAR	文字数 <= 列の文字長 文字数 > 列の文字長	N/A 22001
SQL_DECIMAL [a] SQL_NUMERIC [a] SQL_TINYINT [a] SQL_SMALLINT [a] SQL_INTEGER [a] SQL_BIGINT [a]	切り捨てなしで変換したデータ、または小数桁を切り捨てて変換したデータ 整数桁を切り捨てて変換したデータ	N/A 22003
SQL_REAL SQL_FLOAT SQL_DOUBLE	データは数値が変換されるデータ型の範囲内 データは数値が変換されるデータ型の範囲外	N/A 22003
注: [a] 「N/A」の場合は、ドライバーは小数部分の切り捨てがある場合に、オプションで SQL_SUCCESS_WITH_INFO および 01S07 を返すことがあります。		

数値 C データ型からデータを変換する際にドライバーは長さまたは標識値を無視し、データ・バッファーのサイズが数値 C データ型のサイズであると想定します。長さまたは標識値は、SQLPutData の StrLen_or_IndPtr 引数、および SQLBindParameter の StrLen_or_IndPtr 引数で指定されたバッファーに渡されます。データ・バッファーは、SQLPutData の DataPtr 引数、および SQLBindParameter の ParameterValuePtr 引数で指定されます。

C から SQL への変換: ビット

ビット ODBC C データ型は、以下のとおりです。

SQL_C_BIT

以下の表には、ビット C データから変換することができる ODBC SQL データ型を記載します。表の列と用語についての説明は、312 ページの『変換表の説明 (C から SQL への変換)』を参照してください。

表 205. ビット C データから ODBC SQL データ型への変換

SQL 型 ID	テスト	SQLSTATE
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR SQL_WCHAR SQL_WVARCHAR SQL_WLONGVARCHAR	なし	N/A
SQL_DECIMAL SQL_NUMERIC SQL_TINYINT SQL_SMALLINT SQL_INTEGER SQL_BIGINT SQL_REAL SQL_FLOAT SQL_DOUBLE	なし	N/A

ビット C データ型からデータを変換する際に、ドライバーは長さまたは標識値を無視し、データ・バッファーのサイズがビット C データ型のサイズであると想定します。長さまたは標識値は、SQLPutData の StrLen_or_Ind 引数、および SQLBindParameter の StrLen_or_IndPtr 引数で指定されたバッファーに渡されます。データ・バッファーは、SQLPutData の DataPtr 引数、および SQLBindParameter の ParameterValuePtr 引数で指定されます。

C から SQL への変換: バイナリー

バイナリー ODBC C データ型は、以下のとおりです。

SQL_C_BINARY

以下の表には、バイナリー C データから変換することができる ODBC SQL データ型を記載します。表中の列および用語の説明は、 312 ページの『変換表の説明 (C から SQL への変換)』を参照してください。

表 206. バイナリー C データから ODBC SQL データ型への変換

SQL 型 ID	テスト	SQLSTATE
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR	データのバイト長 <= 列のバイト長 データのバイト長 > 列の長さ	N/A 22001
SQL_WCHAR SQL_WVARCHAR SQL_WLONGVARCHAR	データの文字長 <= 列の文字長 データの文字長 > 列の文字長	N/A 22001
SQL_DECIMAL SQL_NUMERIC SQL_TINYINT SQL_SMALLINT SQL_INTEGER SQL_BIGINT SQL_REAL SQL_FLOAT SQL_DOUBLE SQL_TYPE_DATE SQL_TYPE_TIME SQL_TYPE_TIMESTAMP	データのバイト長 = SQL データの長さ データの長さ <> SQL データの長さ	N/A 22003
SQL_BINARY SQL_VARBINARY SQL_LONGVARBINARY	データの長さ <= 列の長さ データの長さ > 列の長さ	N/A 22001

C から SQL への変換: 日付

日付 ODBC C データ型は、以下のとおりです。

```
SQL_C_DATE
```

以下の表には、日付 C データから変換することができる ODBC SQL データ型を記載します。表中の列および用語の説明は、 312 ページの『変換表の説明 (C から SQL への変換)』を参照してください。

表 207. 日付 C データから ODBC SQL データ型への変換

SQL 型 ID	テスト	SQLSTATE
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR	列のバイト長 >= 10 列のバイト長 < 10 データ値は有効な日付ではない	N/A 22001 22008
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR	列の文字長 >= 10 列の文字長 < 10 データ値は有効な日付ではない	N/A 22001 22008
SQL_TYPE_DATE	データ値は有効な日付 データ値は有効な日付ではない	N/A 22007
SQL_TYPE_TIMESTAMP	データ値は有効な日付 [a] データ値は有効な日付ではない	N/A 22007
注: [a] タイム・スタンプの時刻部分はゼロに設定されます。		

SQL_C_TYPE_DATE 構造体で有効な値については、 283 ページの『C データ型』を参照してください。

日付 C データを文字 SQL データに変換すると、結果の文字データは「yyyy-mm-dd」フォーマットになります。

日付 C データ型からデータを変換する際に、ドライバーは長さまたは標識値を無視し、データ・バッファーのサイズが日付 C データ型のサイズであると想定します。長さまたは標識値は、SQLPutData の StrLen_or_Ind 引数、および SQLBindParameter の StrLen_or_IndPtr 引数で指定されたバッファーに渡されます。データ・バッファーは、SQLPutData の DataPtr 引数、および SQLBindParameter の ParameterValuePtr 引数で指定されます。

C から SQL への変換: 時刻

時刻 ODBC C データ型は、以下のとおりです。

SQL_C_TIME

以下の表には、時刻 C データから変換することができる ODBC SQL データ型を記載します。表中の列および用語の説明は、 312 ページの『変換表の説明 (C から

SQL への変換)』を参照してください。

表 208. 時刻 C データから ODBC SQL データ型への変換

SQL 型 ID	テスト	SQLSTATE
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR	列のバイト長 >= 8	N/A
	列のバイト長 < 8	22001
	データ値は有効な時刻ではない	22008
SQL_WCHAR SQL_WVARCHAR SQL_WLONGVARCHAR	列の文字長 >= 8	N/A
	列の文字長 < 8	22001
	データ値は有効な時刻ではない	22008
SQL_TYPE_TIME	データ値は有効な時刻	N/A
	データ値は有効な時刻ではない	22007
SQL_TYPE_TIMESTAMP	データ値は有効な時刻 [a]	N/A
	データ値は有効な時刻ではない	22007
注: [a] タイム・スタンプの日付部分は現在日付に設定され、タイム・スタンプの小数秒部分はゼロに設定されます。		

SQL_C_TYPE_TIME 構造体で有効な値については、 283 ページの『C データ型』を参照してください。

時刻 C データを文字 SQL データに変換すると、結果の文字データは「hh:mm:ss」フォーマットになります。

時刻 C データ型からデータを変換する際に、ドライバーは長さまたは標識値を無視し、データ・バッファーのサイズが時刻 C データ型のサイズであると想定します。長さまたは標識値は、SQLPutData の StrLen_or_Ind 引数、および SQLBindParameter の StrLen_or_IndPtr 引数で指定されたバッファーに渡されます。データ・バッファーは、SQLPutData の DataPtr 引数、および SQLBindParameter の ParameterValuePtr 引数で指定されます。

C から SQL への変換: タイム・スタンプ

タイム・スタンプ ODBC C データ型は、以下のとおりです。

SQL_C_TIMESTAMP

以下の表には、タイム・スタンプ C データから変換することができる ODBC SQL データ型を記載します。表中の列および用語の説明は、 312 ページの『変換表の説明 (C から SQL への変換)』を参照してください。

表 209. タイム・スタンプ C データから ODBC SQL データ型への変換

SQL 型 ID	テスト	SQLSTATE
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR	列のバイト長 >= 文字のバイト長 19 <= 列のバイト長 < 文字のバイト長 列のバイト長 < 19 データ値は有効な日付ではない	N/A 22001 22001 22008
SQL_WCHAR SQL_WVARCHAR SQL_WLONGVARCHAR	列の文字長 >= データの文字長 19 <= 列の文字長 < データの文字長 列の文字長 < 19 データ値は有効なタイム・スタンプではない	N/A 22001 22001 22008
SQL_TYPE_DATE	時刻フィールドがゼロ 時刻フィールドがゼロ以外 データ値には有効な日付が含まれていない	N/A 22008 22007
SQL_TYPE_TIME	小数秒フィールドはゼロ [a] 小数秒フィールドはゼロ以外 [a] データ値に有効な時刻が含まれていない	N/A 22008 22007
SQL_TYPE_TIMESTAMP	小数秒フィールドは切り捨てられない 小数秒フィールドは切り捨てられる データ値は有効なタイム・スタンプではない	N/A 22008 22007
注: [a] タイム・スタンプ構造体の日付フィールドは無視されます。		

SQL_C_TIMESTAMP 構造体で有効な値については、283 ページの『C データ型』を参照してください。

タイム・スタンプ C データを文字 SQL データに変換すると、結果の文字データは「yyyy-mm-dd hh:mm:ss [.f. ..]」フォーマットになります。

タイム・スタンプ C データ型からデータを変換する際に、ドライバーは長さまたは標識値を無視し、データ・バッファーのサイズがタイム・スタンプ C データ型のサイズであると想定します。長さまたは標識値は、SQLPutData の StrLen_or_Ind 引数、および SQLBindParameter の StrLen_or_IndPtr 引数で指定されたバッファーに渡されます。データ・バッファーは、SQLPutData の DataPtr 引数、および SQLBindParameter の ParameterValuePtr 引数で指定されます。

C から SQL へのデータ変換例

以下の例では、ドライバーによる C データから SQL データへの変換方法を示します。

表 210. C データから SQL データへの変換

C データ型	C データ値	SQL データ型	列の長さ	SQL データ値	SQLSTATE
SQL_C_CHAR	abcdef¥0 [a]	SQL_CHAR	6	abcdef	N/A
SQL_C_CHAR	abcdef¥0 [a]	SQL_CHAR	5	abcde	22001
SQL_C_CHAR	1234.56¥0 [a]	SQL_DECIMAL	8 [b]	1234.56	N/A
SQL_C_CHAR	1234.56¥0 [a]	SQL_DECIMAL	7 [b]	1234.5	22001
SQL_C_CHAR	1234.56¥0 [a]	SQL_DECIMAL	4	----	22003
SQL_C_FLOAT	1234.56	SQL_FLOAT	適用外	1234.56	N/A
SQL_C_FLOAT	1234.56	SQL_INTEGER	適用外	1234	22001
SQL_C_FLOAT	1234.56	SQL_TINYINT	適用外	----	22003
SQL_C_TYPE_DATE	1992,12,31 [c]	SQL_CHAR	10	1992-12-31	N/A
SQL_C_TYPE_DATE	1992,12,31 [c]	SQL_CHAR	9	----	22003
SQL_C_TYPE_DATE	1992,12,31 [c]	SQL_TIMESTAMP	適用外	1992-12-31 00:00:00.0	N/A
SQL_C_TYPE TIMESTAMP	1992,12,31, 23,45,55, 120000000 [d]	SQL_CHAR	22	1992-12-31 23:45:55.12	N/A
SQL_C_TYPE TIMESTAMP	1992,12,31, 23,45,55, 120000000 [d]	SQL_CHAR	21	1992-12-31 23:45:55.1	22001
SQL_C_TYPE TIMESTAMP	1992,12,31, 23,45,55, 120000000 [d]	SQL_CHAR	18	----	22003

表 210. C データから SQL データへの変換 (続き)

C データ型	C データ値	SQL データ型	列の長さ	SQL データ値	SQLSTATE
注: [a] 「¥0」は NULL 終了バイトを表します。NULL 終了バイトは、データの長さが SQL_NTS の場合に限り必要です。 [b] 数値のバイトに加えて、符号と小数点にそれぞれ 1 バイトずつ必要です。 [c] このリストの数値は、SQL_DATE_STRUCT 構造体のフィールドに格納される数値です。 [d] このリストの数値は、SQL_TIMESTAMP_STRUCT 構造体のフィールドに格納される数値です。					

付録 F. スカラー関数

このセクションでは、ODBC スカラー関数の追加情報を示します。

ODBC は、5 つのタイプのスカラー関数を指定します。

- ストリング関数
- 数字関数
- 日時関数
- システム関数
- データ型変換関数

スカラー関数は、照会の各行に対する値を 1 つ返す関数です。SQRT() や ABS() のような関数は、スカラー関数です。SUM() や AVG() のような関数は、複数の行を処理する場合であっても単一の値を返すので、スカラー関数ではありません。

このセクションに、各スカラー関数カテゴリーの表を記載します。各表内では、SQL-92 に合わせて ODBC 3.0 に関数が追加されています。また、各表には関数が導入された時点のバージョン番号が記載されています。

ODBC および SQL-92 のスカラー関数

このトピックでは、ODBC および SQL-92 のスカラー関数についての情報を提供します。

関数はデータ・ソース固有の場合が多いので、ODBC はスカラー関数からの戻り値のデータ型を必要としません。データ型変換を強制するには、アプリケーションで CONVERT スカラー関数を使用する必要があります。

注:

ODBC および SQL-92 は関数を異なる方法で分類します。ODBC はスカラー関数を引数の型で分類しますが、SQL-92 は戻り値で分類します。例えば、ODBC では EXTRACT 関数は日時関数として分類されます。これは、extract-field 引数が日時キーワードで、extract_source 引数が日時またはインターバルの表現だからです。ところが SQL-92 では、戻り値が数値なので、EXTRACT 関数は数値スカラー関数として分類されます。

アプリケーションは SQLGetInfo を呼び出して、ドライバーがサポートしているスカラー関数を判別する必要があります。スカラー関数の分類用に、ODBC および SQL-92 の情報タイプが使用可能です。ODBC および SQL-92 は異なる分類を使用しているので、同じ関数の情報タイプが ODBC と SQL-92 で異なっていることがあります。例えば、EXTRACT 関数のサポートを確認するには、ODBC では SQL_TIMEDATE_FUNCTIONS、SQL-92 では SQL_SQL92_NUMERIC_VALUE_FUNCTIONS の情報タイプが必要です。

ストリング関数

このトピックではストリング処理関数をリストします。

アプリケーションは、SQL_STRING_FUNCTIONS 情報タイプを指定して SQLGetInfo を呼び出し、ドライバーがサポートしているストリング関数を判別することができます。

ストリング関数引数

表 211. ストリング関数引数

表示引数	定義
string_exp	これらの引数は、列の名前、ストリング・リテラル、または別のスカラー関数の結果のいずれかであり、基礎となるデータ型は SQL_CHAR、SQL_VARCHAR、または SQL_LONGVARCHAR で表すことができます。
start、*length*、または *count*	これらの引数は、数値リテラル、または別のスカラー関数の結果であり、基礎となるデータ型は SQL_TINYINT、SQL_SMALLINT、または SQL_INTEGER で表すことができます。
character_exp	これらの引数は可変長の文字ストリングです。

以下のストリング関数は 1 を基数としたもので、ストリングの先頭文字は文字 1 で、文字 0 ではありません。

注: ODBC 3.0 では、SQL-92 との整合性のために、BIT_LENGTH、CHAR_LENGTH、 CHARACTER_LENGTH、 OCTET_LENGTH、および POSITION のストリング・スカラー関数が追加されました。

ストリング関数のリスト

表 212. ストリング関数のリスト

関数	説明
ASCII(string_exp) (ODBC 1.0)	*string_exp* の左端の文字の ASCII コード値を整数として返します。
BIT_LENGTH(string_exp) (ODBC 3.0)	ストリング式の長さをビット数で返します。
CHAR(code) (ODBC 1.0)	code によって ASCII コード値が指定されている文字を返します。*code* の値は 0 と 255 の間でなければなりません。それ以外の場合は戻り値はデータ・ソースに依存します。

表 212. ストリング関数のリスト (続き)

関数	説明
CHAR_LENGTH(string_exp) (ODBC 3.0)	ストリング式が文字データ型の場合は、ストリング式の長さを文字数で返します。それ以外の場合は、ストリング式の長さをバイト数で返します (ビット数を 8 で除算した数値以上の最小の整数)。(この関数は CHARACTER_LENGTH 関数と同じです。)
CHARACTER_LENGTH(string_exp) (ODBC 3.0)	ストリング式が文字データ型の場合は、ストリング式の長さを文字数で返します。それ以外の場合は、ストリング式の長さをバイト数で返します (ビット数を 8 で除算した数値以上の最小の整数)。(この関数は CHAR_LENGTH 関数と同じです。)
CONCAT(string_exp1, string_exp2) (ODBC 1.0)	*string_exp2* を *string_exp1* に連結した結果の文字ストリングを返します。結果ストリングは DBMS に依存します。
DIFFERENCE(string_exp1, string_exp2) (ODBC 2.0)	この関数は、2 つの文字式の soundex (以下の soundex 関数を参照) の値の違いを整数として返します。返される整数は、soundex の値内で同じ文字の数です。戻り値は 0 から 4 の間で、0 は類似性がほとんどまたはまったくないことを、4 は類似性が高いか、値が同一であることを示します。
INSERT(string_exp1, start, length, string_exp2) (ODBC 1.0)	*start* から開始して *string_exp1* から *length* 文字が削除され、 *start* から開始して *string_exp* に *string_exp2* が挿入された文字ストリングを返します。
LCASE(string_exp) (ODBC 1.0)	*string_exp* と同等のストリングを、すべての大文字を小文字に変換して返します。
LEFT(string_exp, count) (ODBC 1.0)	*string_exp* の文字の左端の *count* を返します。
LENGTH(string_exp) (ODBC 1.0)	末尾ブランクを除外して、*string_exp* の文字数を返します。

表 212. ストリング関数のリスト (続き)

関数	説明
LOCATE(string_exp1, string_exp2[, start])	*string_exp2* 内で *string_exp1* の最初のオカレンスの開始位置を返します。オプションの引数である *start* が指定されていない限り、*string_exp1* の最初のオカレンスの検索は、*string_exp2* 内の先頭文字位置から開始します。*start* を指定すると、検索は *start* の値で示される文字位置から開始します。 *string_exp2* の先頭文字位置は値 1 で示されます。 *string_exp1* が *string_exp2* で見つからないと、値 0 が返されます。 アプリケーションが *string_exp1*、*string_exp2*、および *start* の引数を指定して LOCATE スカラー関数を呼び出すことができる場合は、SQL_STRING_FUNCTIONS オプションを指定して SQLGetInfo が呼び出されると、ドライバーは SQL_FN_STR_LOCATE を返します。アプリケーションが、*string_exp1* および *string_exp2* の引数でしか LOCATE スカラー関数を呼び出すことができない場合は、SQL_STRING_FUNCTIONS オプションを指定して SQLGetInfo が呼び出されると、ドライバーは SQL_FN_STR_LOCATE_2 を返します。これら 2 つまたは 3 つの引数を指定した LOCATE 関数の呼び出しをサポートするドライバーは、SQL_FN_STR_LOCATE および SQL_FN_STR_LOCATE_2 の両方を返します。
LTRIM(string_exp) (ODBC 1.0)	先行ブランクを除外して *string_exp* の文字を返します。
OCTET_LENGTH(string_exp) (ODBC 3.0)	ストリング式の長さをバイト数で返します。結果は、ビット数を 8 で除算した数値以上の最小の整数です。
POSITION(character_exp IN character_exp) (ODBC 3.0)	2 番目の文字式での、先頭文字式の位置を返します。結果は、インプリメンテーションで定義された精度および 0 の位取りを持つ厳密な数値です。
REPEAT(string_exp, count) (ODBC 1.0)	*count* 回繰り返された *string_exp* から構成される文字ストリングを返します。
REPLACE(string_exp1, string_exp2, string_exp3) (ODBC 1.0)	*string_exp1* 内で *string_exp2* のオカレンスを検索し、*string_exp3* に置き換えます。

表 212. ストリング関数のリスト (続き)

関数	説明
RIGHT(string_exp, count) (ODBC 1.0)	*string_exp* の文字の右端の *count* を返します。
RTRIM(string_exp) (ODBC 1.0)	末尾ブランクを除外して *string_exp* の文字を返します。
SOUNDEX(string_exp1) (ODBC 2.0)	引数の音声表現を含む文字ストリングを返します。この関数によって、スペルは違うけれども、発音が似ている英語の単語を比較することができます。Soundex に単語を提供すると、米国国勢調査局が 1930 年代以降使用している 4 文字の音声コードが返されます。
SPACE(count) (ODBC 2.0)	*count* 個のスペースから成る文字ストリングを返します。
SUBSTRING(string_exp, start, length) (ODBC 1.0)	*string_exp* から派生した、*start* で指定された文字位置から *length* 文字分の文字ストリングを返します。
TRIM(string_exp)	先行ブランクおよび末尾ブランクを除外して *string_exp* の文字を返します。
UCASE(string_exp) (ODBC 1.0)	*string_exp* と同等のストリングを、すべての小文字を大文字に変換して返します。

数字関数

このトピックでは、ODBC スカラー関数セットに含まれている数字関数について説明します。

アプリケーションは、SQL_NUMERIC_FUNCTIONS 情報タイプを指定して SQLGetInfo を呼び出し、ドライバーがサポートしている数字関数を判別することができます。

ABS、ROUND、TRUNCATE、SIGN、FLOOR、および CEILING (入力パラメーターと同じデータ型の値を返す) を除き、すべての数字関数はデータ型 SQL_FLOAT の値を返します。

数字関数引数

表 213. 数字関数引数

表示引数	定義
numeric_exp	これらの引数は、列の名前、別のスカラー関数の結果、または数値リテラルであり、基礎となるデータ型は SQL_NUMERIC、 SQL_DECIMAL、 SQL_TINYINT、 SQL_SMALLINT、 SQL_INTEGER、 SQL_BIGINT、 SQL_FLOAT、 SQL_REAL、 または SQL_DOUBLE で表すことができます。
float_exp	これらの引数は、列の名前、別のスカラー関数の結果、または数値リテラルであり、基礎となるデータ型は SQL_FLOAT で表すことができます。
integer_exp	これらの引数は、列の名前、別のスカラー関数の結果、または数値リテラルであり、基礎となるデータ型は SQL_TINYINT、 SQL_SMALLINT、 SQL_INTEGER、または SQL_BIGINT で表すことができます。

数字関数のリスト

表 214. 数字関数のリスト

関数	説明
ABS(numeric_exp) (ODBC 1.0)	*numeric_exp* の絶対値を返します。
ACOS(float_exp) (ODBC 1.0)	*float_exp* の逆余弦を、ラジアンで表した角度として返します。
ASIN(float_exp) (ODBC 1.0)	*float_exp* の逆正弦を、ラジアンで表した角度として返します。
ATAN(float_exp) (ODBC 1.0)	*float_exp* の逆正接を、ラジアンで表した角度として返します。
ATAN2(float_exp1, float_exp2) (ODBC 2.0)	*float_exp1* および *float_exp2* でそれぞれ指定された x 座標および y 座標の逆正接を、ラジアンで表した角度として返します。
CEILING(numeric_exp) (ODBC 1.0)	*numeric_exp* 以上の最小の整数を返します。戻り値は、入力パラメーターと同じデータ型です。

表 214. 数字関数のリスト (続き)

関数	説明
COS(float_exp) (ODBC 1.0)	*float_exp* の余弦を返します。ここで *float_exp* はラジアンで表した角度です。
COT(float_exp) (ODBC 1.0)	*float_exp* の余接を返します。ここで *float_exp* はラジアンで表した角度です。
DEGREES(numeric_exp) (ODBC 2.0)	*numeric_exp* ラジアンから変換された度数を返します。
EXP(float_exp) (ODBC 1.0)	*float_exp* の指数値を返します。
FLOOR(numeric_exp) (ODBC 1.0)	*numeric_exp* 以下の最大の整数を返します。戻り値は入力パラメーターと同じデータ型です。
LOG(float_exp) (ODBC 1.0)	*float_exp* の自然対数を返します。
LOG10(float_exp) (ODBC 2.0)	*float_exp* の基底 10 の対数を返します。
MOD(integer_exp1, integer_exp2) (ODBC 1.0)	*integer_exp2* で除算された *integer_exp1* の剰余 (モジュラス) を返します。
PI( ) (ODBC 1.0)	パイの定数値を浮動小数点値として返します。
POWER(numeric_exp, integer_exp)	*numeric_exp* の値の *integer_exp* 乗を返します。
RADIANS(numeric_exp) (ODBC 2.0)	*numeric_exp* 度から変換されたラジアン数を返します。
ROUND(numeric_exp, integer_exp) (ODBC 2.0)	小数点以下 *integer_exp* 桁に丸められた *numeric_exp* を返します。 *integer_exp* が負の数の場合、 *numeric_exp* は小数点以上 |*integer_exp*| 桁に丸められます。

表 214. 数字関数のリスト (続き)

関数	説明
SIGN(numeric_exp) (ODBC 1.0)	*numeric_exp* の標識つまり符号を返します。 *numeric_exp* がゼロより小さいと、-1 が返されます。 *numeric_exp* がゼロの場合は、0 が返されます。 *numeric_exp* がゼロより大きいと、1 が返されます。
SIN(float_exp) (ODBC 1.0)	*float_exp* の正弦を返します。ここで *float_exp* はラジアンで表した角度です。
SQRT(float_exp) (ODBC 1.0)	*float_exp* の平方根を返します。
TAN(float_exp) (ODBC 1.0)	*float_exp* の正接を返します。ここで *float_exp* はラジアンで表した角度です。
TRUNCATE(numeric_exp, integer_exp) (ODBC 2.0)	小数点以下 *integer_exp* 桁に切り捨てられた *numeric_exp* を返します。 *integer_exp* が負の数の場合、*numeric_exp* は、小数点以上 |*integer_exp*| 桁に切り捨てられます。

日時関数

このセクションでは、ODBC スカラー関数セットに含まれる日時関数をリストします。

アプリケーションは、SQL_TIMEDATE_FUNCTIONS 情報タイプを指定して SQLGetInfo を呼び出し、ドライバーがサポートしている日時関数を判別することができます。

日時関数引数

表 215. 日時関数引数

表示引数	定義
timestamp_exp	これらの引数は、列の名前、別のスカラー関数の結果、もしくは *ODBC_time_escape*、*ODBC_date_escape*、または *ODBC_timestamp_escape* であり、基礎となるデータ型は SQL_CHAR、 SQL_VARCHAR、 SQL_TYPE_TIME、 SQL_TYPE_DATE、 または SQL_TYPE_TIMESTAMP で表すことができます。
date_exp	これらの引数は、列の名前、別のスカラー関数の結果、もしくは *ODBC_date_escape* または *ODBC_timestamp_escape* であり、基礎となるデータ型は SQL_CHAR、 SQL_VARCHAR、 SQL_TYPE_DATE、または SQL_TYPE_TIMESTAMP で表すことができます。

表 215. 日時関数引数 (続き)

表示引数	定義
time_exp	これらの引数は、列の名前、別のスカラー関数の結果、もしくは *ODBC_time_escape* または *ODBC_timestamp_escape* であり、基礎となるデータ型は SQL_CHAR、 SQL_VARCHAR、SQL_TYPE_TIME、または SQL_TYPE_TIMESTAMP で表すことができます。

注: ODBC 3.0 では、SQL-92 との整合性のために、CURRENT_DATE、CURRENT_TIME、および CURRENT_TIMESTAMP の日時スカラー関数が追加されました。

日時関数のリスト

表 216. 日時関数のリスト

関数	説明
CURRENTTIME [(time_precision)] (ODBC 3.0)	現在の現地時間を時刻値として返します。 time_precision 引数により、戻り値の秒精度が決まります。
CURRENT_TIMESTAMP [(timestamp_precision)] (ODBC 3.0)	現在の現地日付および現地時間をタイム・スタンプ値として返します。 timestamp_precision 引数により、返されるタイム・スタンプの秒精度が決まります。
CURDATE( ) (ODBC 1.0)	現在日付を返します。
CURTIME( ) (ODBC 1.0)	現在の現地時間を返します。
DAYNAME(date_exp) (ODBC 2.0)	*date_exp* の日部分に対するデータ・ソース固有の曜日の名称 (例えば、データ・ソースで英語が使用されている場合は Sunday から Saturday または Sun. から Sat.、ドイツ語が使用されている場合は Sonntag から Samstag) を含む文字ストリングを返します。
DAYOFMONTH(date_exp) (ODBC 1.0)	*date_exp* の月の何日目かを示す値を 1 から 31 までの範囲の整数値として返します。
DAYOFWEEK(date_exp) (ODBC 1.0)	*date_exp* の週フィールドに基づき、曜日を 1 から 7 までの範囲の整数値として返します。1 は日曜日を表します。

表 216. 日時関数のリスト (続き)

関数	説明
DAYOFYEAR(date_exp) (ODBC 1.0)	*date_exp* の年フィールドに基づき、年間通算日を 1 から 366 までの範囲の整数値として返します。

EXTRACT(extract_field FROM extract_source)
 	
(ODBC 3.0)
	extract_source の *extract_field* 部分を返します。
	extract_source 引数は、日時またはインターバルの式です。
	extract_field 引数は、以下のキーワードのうちの 1 つです。
	`YEAR`
	`MONTH`
	`DAY`
	`HOUR`
	`MINUTE`
	`SECOND`
	戻り値の精度は、インプリメンテーションで定義されます。SECOND が指定されていない限り、位取りは 0 ですが、指定されている場合は、位取りは extract_source フィールドの小数秒精度以上です。

HOUR(time_exp) (ODBC 1.0)	*time_exp* の時フィールドに基づき、時刻を 0 から 23 までの範囲の整数値として返します。
MINUTE(time_exp) (ODBC 1.0)	*time_exp* の分フィールドに基づき、分を 0 から 59 までの範囲の整数値として返します。
MONTH(date_exp) (ODBC 1.0)	*date_exp* の月フィールドに基づき、月を 1 から 12 までの範囲の整数値として返します。
MONTHNAME(date_exp) (ODBC 2.0)	*date_exp* の月部分に対するデータ・ソース固有の月の名称 (例えば、データ・ソースで英語が使用されている場合は January から December または Jan. から Dec.、ドイツ語が使用されている場合は Januar から Dezember) を含む文字ストリングを返します。
NOW( ) (ODBC 1.0)	現在の日付および時刻をタイム・スタンプ値として返します。
QUARTER(date_exp) (ODBC 1.0)	*date_exp* の四半期を 1 から 4 までの範囲の整数値として返します。 1 は 1 月 1 日から 3 月 31 日までを表します。
SECOND(time_exp) (ODBC 1.0)	*time_exp* の秒を 0 から 59 までの範囲の整数値として返します。

表 216. 日時関数のリスト (続き)

関数	説明
TIMESTAMPADD(interval, integer_exp, timestamp_exp) (ODBC 2.0)	interval 型の *integer_exp* インターバルを *timestamp_exp* に加算して計算したタイム・スタンプを返します。 interval の有効値は、以下のキーワードです。 SQL_TSI_FRAC_SECOND SQL_TSI_SECOND SQL_TSI_MINUTE SQL_TSI_HOUR SQL_TSI_DAY SQL_TSI_WEEK SQL_TSI_MONTH SQL_TSI_QUARTER SQL_TSI_YEAR 小数秒は、10 億分の 1 秒 (ナノ秒) で表されます。例えば、以下の SQL ステートメントは、各従業員の名前および一周年記念日を返します。 SELECT NAME, {fn TIMESTAMPADD(SQL_TSI_YEAR, 1, HIRE_DATE)} FROM EMPLOYEES *timestamp_exp* に時刻値を指定し、 interval に日、週、月、四半期、または年を指定した場合、*timestamp_exp* の日付部分は、結果のタイム・スタンプを計算する前は現在日付に設定されます。 *timestamp_exp* に日付値を指定し、interval に小数秒、秒、分、または時間を指定した場合、*timestamp_exp* の時刻部分は、結果のタイム・スタンプを計算する前は 0 に設定されます。 アプリケーションは、SQL_TIMEDATE_ADD_INTERVALS オプションを使用して SQLGetInfo を呼び出し、データ・ソースがどのインターバルをサポートしているかを判別します。

表 216. 日時関数のリスト (続き)

関数	説明

TIMESTAMPDIFF(interval, timestamp_exp1, timestamp_exp2)
 	
(ODBC 2.0)
	timestamp_exp1 と *timestamp_exp2* の間の *interval* 型の単位間隔の数 (整数) を返します。
	
	アプリケーションが古い TIMESTAMPDIFF セマンティクスに従っている場合は、solid.ini ファイルの SQL セクションに以下の構成設定をすることにより、古い動作をエミュレートすることができます。
	
	`[SQL]`
	`EmulateOldTIMESTAMPDIFF=YES`
	
	古いセマンティクスでは、*timestamp_exp2* が *timestamp_exp1* よりどれだけ大きいかを示す interval 型のインターバル数の整数を返すことに注意してください。
	
	interval の有効値は、以下のキーワードです。
	
	`SQL_TSI_FRAC_SECOND`
	`SQL_TSI_SECOND`
	`SQL_TSI_MINUTE`
	`SQL_TSI_HOUR`
	`SQL_TSI_DAY`
	`SQL_TSI_WEEK`
	`SQL_TSI_MONTH`
	`SQL_TSI_QUARTER`
	`SQL_TSI_YEAR`
	
	小数秒は、10 億分の 1 秒 (ナノ秒) で表されます。例えば、以下の SQL ステートメントは、各従業員の名前および勤続年数を返します。
	
	`SELECT NAME, {fn`
	`TIMESTAMPDIFF(SQL_TSI_YEAR,`
	`{fn CURDATE()}, HIRE_DATE)}`
	`FROM EMPLOYEES`
	
	どちらかのタイム・スタンプ式に時刻値を指定し、interval に日、週、月、四半期、または年を指定した場合、そのタイム・スタンプの日付部分は、タイム・スタンプ間の差を計算する前は現在日付に設定されます。
	
	どちらかのタイム・スタンプ式に日付値を指定し、interval に小数秒、秒、分、または時間を指定した場合、そのタイム・スタンプの時刻部分は、タイム・スタンプ間の差を計算する前は 0 に設定されます。
	
	アプリケーションは、SQL_TIMEDATE_DIFF_INTERVALS オプションを使用して SQLGetInfo を呼び出し、データ・ソースがどのインターバルをサポートしているかを判別します。

WEEK(date_exp) (ODBC 1.0)	*date_exp* の週フィールドに基づき、年間通算週を 1 から 53 までの範囲の整数値として返します。

表 216. 日時関数のリスト (続き)

関数	説明
YEAR(date_exp) (ODBC 1.0)	*date_exp* の年フィールドに基づき、年を整数値として返します。範囲は、データ・ソースに依存します。

システム関数

このセクションでは、ODBC スカラー関数セットに含まれるシステム関数をリストします。

アプリケーションは、SQL_SYSTEM_FUNCTIONS 情報タイプを指定して SQLGetInfo を呼び出し、ドライバーがサポートしているシステム関数を判別することができます。

システム関数引数

表 217. システム関数引数

表示引数	定義
exp	これらの引数は、列の名前、別のスカラー関数の結果、またはリテラルであり、基礎となるデータ型は SQL_NUMERIC、SQL_DECIMAL、 SQL_TINYINT、 SQL_SMALLINT、SQL_INTEGER、 SQL_BIGINT、 SQL_FLOAT、SQL_REAL、 SQL_DOUBLE、 SQL_TYPE_DATE、SQL_TYPE_TIME、 または SQL_TYPE_TIMESTAMP で表すことができます。
value	これらの引数はリテラル定数であり、基礎となるデータ型は SQL_NUMERIC、 SQL_DECIMAL、 SQL_TINYINT、SQL_SMALLINT、 SQL_INTEGER、 SQL_BIGINT、SQL_FLOAT、 SQL_REAL、 SQL_DOUBLE、SQL_TYPE_DATE、 SQL_TYPE_TIME、 または SQL_TYPE_TIMESTAMP で表すことができます。
integer_exp	これらの引数は、列の名前、別のスカラー関数の結果、または数値リテラルであり、基礎となるデータ型は SQL_TINYINT、SQL_SMALLINT、 SQL_INTEGER、または SQL_BIGINT で表すことができます。

戻り値は、ODBC データ型として表されます。

システム関数のリスト

表 218. システム関数のリスト

関数	説明
DATABASE( ) (ODBC 1.0)	接続ハンドルに対応するデータベースの名前を返します。(データベース名は、SQL_CURRENT_QUALIFIER 接続オプションを指定して SQLGetConnectOption を呼び出すことでも入手できます。)
IFNULL(exp, value) (ODBC 1.0)	exp が NULL の場合、value が返されます。exp が NULL でない場合、exp が返されます。value のデータ型は、exp のデータ型と互換性のあるものである必要があります。
USER( ) (ODBC 1.0)	DBMS 内のユーザーの名前を返します。(ユーザーの許可名は、情報タイプの SQL_USER_NAME を指定した SQLGetInfo によって入手することもできます。) これは、ログイン時と異なる場合があります。

明示的なデータ型変換

明示的なデータ型変換は、 SQL データ型定義に関連して指定されます。

明示的なデータ型変換関数用の ODBC 構文では、変換の制限がありません。1 つのデータ型から別のデータ型への特定の変換の妥当性検査は、各ドライバー固有のインプリメンテーションによります。ドライバーは ODBC 構文をネイティブの構文に変換するので、ODBC 構文で有効であるとしても、データ・ソースによってサポートされていない変換は拒否します。アプリケーションは、ODBC 関数の SQLGetInfo を呼び出し、データ・ソースがサポートしている変換について問い合わせることができます。

CONVERT 関数のフォーマットは以下のとおりです。

CONVERT(*value_exp*, *data_type*)

この関数は、指定された data_type に変換された、value_exp によって指定された値を返します。data_type は以下のキーワードのうちの 1 つです。

- SQL_BIGINT
- SQL_SMALLINT
- SQL_BINARY
- SQL_DATE
- SQL_CHAR
- SQL_TIME
- SQL_DECIMAL
- SQL_TIMESTAMP
- SQL_DOUBLE
- SQL_TINYINT

- SQL_FLOAT
- SQL_VARBINARY
- SQL_INTEGER
- SQL_VARCHAR
- SQL_LONGVARBINARY
- SQL_WCHAR
- SQL_LONGVARCHAR
- SQL_WLONGVARCHAR
- SQL_NUMERIC
- SQL_WVARCHAR
- SQL_REAL

明示的なデータ型変換関数用の ODBC 構文では、変換フォーマットの指定がサポートされていません。基礎となるデータ・ソースが明示的なフォーマットの指定をサポートしている場合、ドライバーはデフォルト値を指定するか、またはフォーマット指定をインプリメントする必要があります。

引数 value_exp は、列の名前、別のスカラー関数の結果、もしくは数値リテラルまたはストリング・リテラルです。以下は、CURDATE スカラー関数の出力を文字ストリングに変換する例です。

```
{ fn CONVERT( { fn CURDATE() }, SQL_CHAR) }
```

ODBC では、スカラー関数からの戻り値にデータ型を必要としません (多くの場合、関数がデータ・ソース固有であるからです)。アプリケーションは、データ型変換を強制的に行う際、可能であれば CONVERT スカラー関数を使用する必要があります。

以下の 2 つの例で、CONVERT 関数の使用方法を説明します。これらの例では、型が SQL_SMALLINT の EMPNO 列および型が SQL_CHAR の EMPNAME 列を持つ EMPLOYEES と呼ばれる表の存在が前提となっています。

アプリケーションで以下を指定した場合、

```
SELECT EMPNO FROM EMPLOYEES WHERE {fn CONVERT(EMPNO,SQL_CHAR)}LIKE '1%'
```

solidDB ODBC ドライバーは、この要求を以下のように変換します。

```
SELECT EMPNO FROM EMPLOYEES WHERE CONVERT_CHAR(EMPNO) LIKE '1%'
```

SQL-92 CAST 関数

ODBC CONVERT 関数に相当する SQL-92 の関数が、CAST 関数です。

この互いに同等な関数の構文は、以下のとおりです。

```
{ fn CONVERT (value_exp, data_type)} /* ODBC */
CAST (value_exp AS data_type) /* SQL 92 */
```

CAST 関数は、FIPS Transitional レベルでサポートします。CAST 関数内のデータ型変換について詳しくは、SQL-92 仕様書を参照してください。

アプリケーションが CAST 関数をサポートしているかどうかを判別するには、SQL_SQL_CONFORMANCE 情報タイプを指定して SQLGetInfo を呼び出します。情報タイプに対する戻り値が以下の場合、CAST 関数はサポートされています。

- SQL_SC_FIPS127_2_TRANSITIONAL
- SQL_SC_SQL92_INTERMEDIATE
- SQL_SC_SQL92_FULL

戻り値が SQL_SC_ENTRY または 0 の場合、SQL_SQL92_VALUE_EXPRESSIONS 情報タイプを指定して SQLGetInfo を呼び出します。SQL_SVE_CAST ビットが設定されていれば、CAST 関数はサポートされています。

付録 G. タイムアウト制御

solidDB では、アクションによってはタイムアウトになる場合があります。タイムアウトは、メイン・サーバー、クライアント・ドライバー、1 次サーバーまたは 2 次サーバー、もしくはマスター・サーバーまたはレプリカ・サーバーによってアクティブにすることができます。

タイムアウトにはデフォルトのファクトリー値が存在し、通常は異なる .ini パラメーターにより設定することができます。いくつかの始動時のデフォルト値は、SQL を使用するかドライバー・インターフェースおよび接続ストリング・パラメーターを使用することにより、異なる制御によって動的に変更することができます。

クライアント・タイムアウト

このトピックでは、データベース・クライアントに関連したタイムアウトについて説明します。

ログイン・タイムアウト

このタイムアウトは、ドライバーがログイン (SQLConnect) の成功を待つ秒数です。デフォルト値はドライバーに依存します。値 (または ODBC の ValuePtr) が 0 の場合、タイムアウトは無効となり、接続試行は無期限に待機することになります。指定したタイムアウトがデータ・ソース内の最大ログイン・タイムアウトを超えた場合、ドライバーはその値を代用し SQLSTATE 01S02 (オプション値の変更) を返します。

このタイムアウトは、TCP プロトコルのみに適用されます。

表 219. ログイン・タイムアウト

INI パラメーター	SQL によるオーバーライド	ドライバー	接続ストリング
		ODBC: SQL_ATTR_LOGIN_TIMEOUT (秒単位) SQL_ATTR_LOGIN_TIMEOUT_MS (ミリ秒単位、非標準) JDBC: メソッド (JDBC 2.0) DriverManger.setLoginTimeout(seconds); 接続プロパティー (非標準) ″solid_login_timeout_ms″ (ミリ秒)	-c milliseconds

表 219. ログイン・タイムアウト (続き)

INI パラメーター	SQL による オーバーライド	ドライバー	接続ストリング
(クライアント・サイド) [Com] ConnectTimeout (ミリ秒単位) または Connect -c オプション (ミリ秒単位)			

タイムアウトのエラー・コードおよびメッセージ:

ODBC:

HYT00 タイムアウトの期限切れ

接続タイムアウト

このタイムアウトは、ドライバーが接続上の要求の完了を待つ秒数 (またはミリ秒数) です。このタイムアウトは、照会実行やログインには関連付けられていません。タイムアウトになると、ドライバーは solidDB サーバーから切断されます。

ドライバーは、照会実行やログインに関連しない場合にタイムアウトになることが可能であれば SQLSTATE HYT00 (タイムアウトの期限切れ) を返します。値 (または ODBC の ValuePtr) が 0 (デフォルト値) の場合、タイムアウトは発生しません。

このタイムアウトは、以下に挙げた以外のすべての ODBC 関数 (ODBC 3.5 仕様) に適用されます。

```
SQLDrivers
              SQLDataSources  SQLGetEnvAttr
                 SQLSetEnvAttr
```

表 220. 接続タイムアウト

INI パラメーター	SQL によるオーバーライド	ドライバー	接続ストリング
(サーバー・サイド) [Com] Listen -r オプション (ミリ秒単位) (クライアント・サイド) [Com] ClientReadTimeout (ミリ秒単位) または Connect (-r オプション) (ミリ秒単位)		ODBC: SQL_ATTR_CONNECTION_TIMEOUT (秒単位) SQL_ATTR_CONNECTION_TIMEOUT_MS (ミリ秒単位、非標準) JDBC: 非標準: 接続プロパティー ″solid_connection_timeout_ms″ (ミリ秒) またはメソッド: SolidConnection.setConnectionTimeout() (ミリ秒単位)	-r milliseconds

注: このタイムアウトはサーバー上でもインプリメントされています。つまり、サーバーは未処理の要求をキャンセルし、クライアントを切断します。

タイムアウトのエラー・コードおよびメッセージ:

ODBC:

HYT01 接続タイムアウトの期限切れ

以下も参照してください。

```
SOLID Server Error 14518:
Connection to the server is broken, connection lost.

SOLID Communication Error 21328 and SOLID Session Error 20024:
Timeout while resolving host name.

SOLID Communication Error 21329 and SOLID Session Error 20025:
Timeout while connecting to a remote host.
```

照会タイムアウト

このタイムアウトは、ドライバーが SQL ステートメントの実行を待つ秒数です。値 (または ODBC の ValuePtr) が 0 (デフォルト値) の場合、タイムアウトは発生しません。

指定したタイムアウトがデータ・ソース内の最大タイムアウトを超えた場合、または指定したタイムアウトが最小タイムアウトより小さい場合、SQLSetStmtAttr はその値を代用し SQLSTATE 01S02 (オプション値の変更) を返します。

このタイムアウトは、以下の ODBC 関数 (ODBC 3.5 仕様) に適用されます。

SQLBrowseConnect

SQLBulkOperations

SQLColumnPrivileges

SQLColumns

SQLConnect

SQLDriverConnect

SQLExecDirect

SQLExecute

SQLExtendedFetch

SQLForeignKeys

SQLGetTypeInfo

SQLParamData

SQLPrepare

SQLPrimaryKeys

SQLProcedureColumns

SQLProcedures

SQLSetPos

SQLSpecialColumns

SQLStatistics

SQLTablePrivileges

SQLTables

注: SELECT ステートメントがタイムアウトになった際にステートメントを再使用する場合、アプリケーションは SQLCloseCursor を呼び出す必要はありません。このステートメント属性に設定された照会タイムアウトは、同期モードおよび非同期モードのどちらでも有効です。

表 221. 照会タイムアウト

INI パラメーター	SQL によるオーバーライド	ドライバー	接続ストリング
		ODBC: SQL_ATTR_QUERY_TIMEOUT (秒単位) SQL_ATTR_QUERY_TIMEOUT_MS (ミリ秒単位、非標準)	

タイムアウトのエラー・コードおよびメッセージ:

ODBC:

HYT00 タイムアウトの期限切れ

サーバー・タイムアウト

このトピックでは、データベース・サーバーに関連したタイムアウトについて説明します。

SQL ステートメント実行タイムアウト

1 つの SQL ステートメントの実行に費やす時間をサーバーで制御することができます。時間が過ぎた場合、サーバーはステートメントを終了し、対応するエラー・コードを返します。このタイムアウトは、以下の呼び出し (ODBC 3.5 仕様) に適用されます。

- SQLExecute()
- SQLExecDirect()
- SQLPrepare()
- SQLForeignKeys()
- SQLColumns()
- SQLProcedureColumns()
- SQLSpecialColumns()
- SQLStatistics()
- SQLPrimaryKeys()
- SQLProcedures()
- SQLTables()
- SQLTablePrivileges()
- SQLColumnPrivileges()
- SQLGetTypeInfo()

このタイムアウトは、対応する JDBC 呼び出しにも適用されます。

表 222. SQL ステートメント実行タイムアウト

INI パラメーター	SQL によるオーバーライド	ドライバー	接続ストリング
	SET STATEMENT MAXTIME minutes	ODBC: SQL_ATTR_QUERY_TIMEOUT (秒単位) SQL_ATTR_QUERY_TIMEOUT_MS (ミリ秒単位、非標準) JDBC: statement.setQueryTimeout()	

タイムアウトのエラー・コードおよびメッセージ:

HYT00 タイムアウトの期限切れ

以下も参照してください。

```
SOLID Server Error 14518:
Connection to the server is broken, connection lost.

SOLID Server Error 14529:
The operation timed out.
```

ロック待機タイムアウト

このタイムアウトは、エンジンがロック解除を待つ時間を秒単位 (またはミリ秒単位) で指定します。タイムアウト・インターバルに達すると、solidDB はタイムアウトになったトランザクションを終了します。デフォルト値は 30 秒で、パラメーターのアクセス・モードは読み取り/書き込みです。

ロック待機タイムアウトは、デッドロックの解決に使用されます。その場合、デッドロックになっている最も古いトランザクションが中止されます。

表 223. ロック待機タイムアウト

INI パラメーター	SQL によるオーバーライド	ドライバー	接続ストリング
[General] LockWaitTimeOut seconds	SET LOCK TIMEOUT {seconds | milliseconds MS}		

タイムアウトのエラー・コードおよびメッセージ:

```
SOLID Database Error 10006:
Concurrency conflict, two transactions updated or deleted the same row.
```

オプティミスティック・ロック・タイムアウト

このタイムアウトでは、オプティミスティック・ロック・タイムアウトを指定します。オプティミスティック・ロックとは、オプティミスティック並行制御方式において SELECT FOR UPDATE による更新が常に成功するように実施される追加のロックです。デフォルトはゼロで、その場合、オプティミスティック・ロックは使用されず、トランザクション妥当性検査が早まるため、各ステートメントの後でトランザクションが中止される場合があります。タイムアウトをゼロ以外の値に設定した場合、SELECT FOR UPDATE はロック取得まで待つか、タイムアウトになり異常終了します。タイムアウトが設定されると、DELETE および UPDATE のステートメントもすべて影響を受けます。

表 224. オプティミスティック・ロック待機タイムアウト

INI パラメーター	SQL によるオーバーライド	ドライバー	接続ストリング
	SET OPTIMISTIC LOCK TIMEOUT {seconds | milliseconds MS}		

タイムアウトのエラー・コードおよびメッセージ:

```
SOLID Database Error 10006:
Concurrency conflict, two transactions updated or deleted the same row.
```

表ロック待機タイムアウト

トランザクションが表に対する排他ロックを取得することがあります。これは、ロック・エスカレーションまたは ALTER TABLE ステートメント実行の試行の結果、もしくは拡張レプリケーション・コマンドの副次作用です。表レベルの競合がある場合、この設定は、排他または共有ロックの解除までのトランザクションの待機時間を示します。単位は秒で、デフォルト値は 30 秒であり、パラメーターのアクセス・モードは読み取り/書き込みです。

具体的には、表レベル・ロックは、PESSIMISTIC キーワードが以下のコマンドで明示的に指定された場合に使用されます。

```
IMPORT SUBSCRIPTIONMESSAGE message_name EXECUTE
(NO EXECUTE オプションがある場合のみ)
MESSAGE message_name FORWARD
MESSAGE message_name GET REPLY
DROP SUBSCRIPTION.
```

表 225. 表ロック待機タイムアウト

INI パラメーター	SQL によるオーバーライド	ドライバー	接続ストリング
[General] TableLockWaitTimeout seconds			

タイムアウトのエラー・コードおよびメッセージ:

```
SOLID Database Error 10006:
Concurrency conflict, two transactions updated or deleted the same row.
```

トランザクション・アイドル・タイムアウト

このタイムアウトでは、アイドル・トランザクションが中止されるまでの時間を分単位で指定します。負の値またはゼロを指定すると無限になります。単位は分で、デフォルト値は 120 分であり、アクセス・モードは読み取り/書き込みです。

表 226. トランザクション・アイドル・タイムアウト

INI パラメーター	SQL によるオーバーライド	ドライバー	接続ストリング
[Srv] AbortTimeOut			

タイムアウトのエラー・コードおよびメッセージ:

```
SOLID Database Error 10026:
 Transaction is timed out.
```

接続アイドル・タイムアウト

このタイムアウトでは、(サーバーにより) 接続がドロップされるまでの継続したアイドル時間を分単位 (またはステートメントの場合、秒かミリ秒単位) で指定します。負の値またはゼロを指定すると無限値になります。パラメーターの単位は分で、デフォルト値は 480 分であり、アクセス・モードは読み取り/書き込みです。

表 227. 接続アイドル・タイムアウト

INI パラメーター	SQL によるオーバーライド	ドライバー	接続ストリング
[Srv] ConnectTimeOut (分)	SET IDLE TIMEOUT {seconds | milliseconds MS}	JDBC: 接続プロパティー (非標準): ″solid_idle_timeout_min″	

タイムアウトのエラー・コードおよびメッセージ:

```
SOLID Communication Error 21308:
Connection is broken (protocol read/write
operation failed with code internal code).
```

`solmsg.out` ファイルも参照してください。

SET IDLE TIMEOUT が設定されていて、トランザクションが指定時間アイドルだった場合、以下のエラーが表示されます。

```
SOLID Database Error 10026:
Transaction is timed out
```

HotStandby タイムアウト

このトピックでは、HotStandby サーバーに関連したタイムアウトについて説明します。

接続タイムアウト

接続タイムアウト値を指定することで、HotStandby 接続操作においてリモート・マシン接続の最大待機時間をミリ秒単位で指定することができます。ConnectTimeout パラメーターを使用するのは、以下の管理コマンドのサブセットに対してのみです。

```
hotstandby connect
hotstandby switch primary
hotstandby switch secondary
```

単位はミリ秒で、デフォルト値は 3000 であり、アクセス・モードは読み取り/書き込みです。

表 228. 接続タイムアウト

INI パラメーター	SQL による オーバーライド	ドライバー	接続ストリング
[HotStandby] ConnectTimeout milliseconds			

ping タイムアウト

このパラメーターは、サーバーが、相手のサーバーが故障またはアクセス不能であると判断するまでの待機時間を指定します。単位はミリ秒で、デフォルト値は 4000 であり、アクセス・モードは読み取り/書き込みです。

表 229. ping タイムアウト

INI パラメーター	SQL による オーバーライド	ドライバー	接続ストリング
[HotStandby] PingTimeout milliseconds			

付録 H. クライアント・サイド構成パラメーター

クライアント・サイド構成パラメーターは、solid.ini 構成ファイルに保管されており、クライアントの始動時に読み取られます。

ほとんどの場合、ファクトリー値設定のままで最善のパフォーマンスと操作容易性が得られますが、特別なケースではパラメーターを変更すると、パフォーマンスが向上します。構成ファイル solid.ini を編集することで、パラメーターを変更することができます。

クライアント・サイドの構成ファイルに設定するパラメーター値は、アプリケーションが SqlConnect ODBC 関数の呼び出しを発行するときに、毎回有効になります。プログラムの実行時にファイルの値を変更すると、その変更はその後に確立された接続に影響します。

solid.ini 構成ファイルを使用したクライアント・サイド・パラメーターの設定

このトピックでは、solid.ini 構成ファイルについて詳しく説明します。

solidDB は、始動時に構成ファイル solid.ini を開こうとします。このファイルが存在しない場合、solidDB は、パラメーターにファクトリー値を使用します。solid.ini ファイルが存在する場合でも、その中の特定のパラメーターに値が設定されていない場合、solidDB は、そのパラメーターにファクトリー値を使用します。ファクトリー値は、使用するオペレーティング・システムに依存する場合があります。

デフォルトでは、クライアントは現行作業ディレクトリーで solid.ini ファイルを検索しますが、通常、これはクライアントを始動したディレクトリーです。solidDB は、ファイルの検索時に以下の優先順位に従います (上から下)。

- SOLIDDIR 環境変数によって指定された場所 (この環境変数が設定されている場合)
- 現行作業ディレクトリー

クライアント・サイドの solid.ini ファイルのフォーマット設定のルール

クライアント・サイドの solid.ini ファイルのフォーマット設定には、サーバー・サイドの solid.ini ファイルの場合と同じルールが適用されます。詳しくは、「*IBM solidDB 管理者ガイド*」の『solid.ini ファイルのフォーマット設定のルール』のセクションを参照してください。

クライアント・サイドの solid.ini ファイル

```
[Com]
;データ・ソースのないこの接続ストリングを使用します。
Listen = tcp host1.acme.com 1315
```

```
[Client]
;SQLConnect で、この時間 (ms) の後タイムアウトになります。
ConnectTimeout = 5000

;ODBC ネットワーク要求で、この時間 (ms) の後タイムアウトになります。
ClientReadTimeout = 10000

[DataSources]
Primary_Server = tcp irix1 1315, The Primary Server
Secondary_Server = tcp irix2 1315, The Secondary Server
```

Client セクション

表 230. Client パラメーター

[Client]	説明	ファクトリー値
ExecRowsPerMessage	このパラメーターは、SELECT ステートメントでの SQLExecute 呼び出しの応答として、クライアント・ドライバーに送信 (プリフェッチ) する結果行の数を指定します。その後に、結果行は、最初の SQLFetch 呼び出しを発行したアプリケーションに返されます。デフォルト値は 2 で、単一行の結果をプリフェッチすることが可能です。SELECT ステートメントから返される行の数が通常多い場合は、これを適切な値に設定すれば、パフォーマンスを大幅に向上できます。 RowsPerMessage 構成パラメーターも参照してください。	サーバーが判断
NoAssertMessages	このパラメーターは、Windows プラットフォームのみに関係します。Yes に設定すると、Windows のランタイム・エラーのダイアログが表示されなくなります。	No

表 230. Client パラメーター (続き)

[Client]	説明	ファクトリー値
ODBCCharBinding	文字データのバインディング方式を定義します。 以下のオプションが使用できます。 • raw (バイナリー) • locale (現在のクライアント・ロケールが使用されます。) • locale:<locale name> (特定のコード・ページが使用されます。) <locale name> の規則は、オペレーティング・システムにより異なります。例えば、Linux 環境では、中国語 (簡体字)/中国のコード・ページ GB18030 のロケール名は zh_CN.gb18030 です。 Windows 環境では、フィンランド語/フィンランドの Latin1 コード・ページのロケール名は fin_fin.1252 です。 値「raw」は、バージョン 6.3 またはそれ以前の solidDB で使用しているバインディングをデータベースで使用する場合に利用できます。	locale
RowsPerMessage	SQLFetch 呼び出しが実行されたとき (およびプリフェッチ行がない場合)、1 件のネットワーク・メッセージにサーバーから返される行の数を指定します。 ExecRowsPerMessage 構成パラメーターも参照してください。	サーバーが判断
StatementCache	ステートメント・キャッシュとは、以前の準備済み SQL ステートメントを数件ほど格納する内部メモリーです。このパラメーターを使用すると、セッションごとにキャッシュに入れるステートメントの数を設定することができます。	6

Com セクション

表 231. Com パラメーター

[Com]	説明	ファクトリー値
ClientReadTimeout	このパラメーターは、接続 (読み取り) のタイムアウトをミリ秒単位で定義します。指定した時間の間に応答を受け取らない場合、ネットワーク要求は失敗します。0 の値を指定すると、タイムアウトは無限に設定されます。この値は、接続ストリング・オプション -r を指定するとオーバーライドすることができます。さらに、ODBC 属性 SQL_ATTR_CONNECTION_TIMEOUT と組み合わせてもオーバーライドできます。 注: TCP プロトコルの場合にのみ当てはまります。	60 000
Connect	Connect パラメーターは、クライアントがサーバーとの接続を確立する際に、デフォルトで接続するネットワーク名 (接続ストリング) を定義します。この値は、データ・ソース名が空の状態で SQLConnect() 呼び出しが発行されるときに使用されます。	tcp localhost 1964
ConnectTimeout	ConnectTimeout パラメーターは、ログインのタイムアウトをミリ秒単位で定義します。 この値は、接続ストリング・オプション -c を指定するとオーバーライドすることができます。さらに、ODBC 属性 SQL_ATTR_LOGIN_TIMEOUT を組み合わせてもオーバーライドできます。 注: TCP プロトコルの場合にのみ当てはまります。	OS 固有
ODBCHandleValidation	ODBCHandleValidation パラメーターは、ODBC ハンドル妥当性検査のオン/オフを切り替えます。 SQL_ATTR_HANDLE_VALIDATION ODBC 属性の詳細については、「*IBM solidDB プログラマー・ガイド*」の『*ODBC ハンドル妥当性検査*』のセクションも参照してください。	No

表 231. *Com* パラメーター *(続き)*

[Com]	説明	ファクトリー値
Trace	このパラメーターを yes に設定すると、確立済みのネットワーク接続のネットワーク・メッセージに関するトレース情報が、TraceFile パラメーターに指定したファイルに書き込まれます。TraceFile パラメーターのファクトリー値は、soltrace.out です。	no
TraceFile	Trace パラメーターを yes に設定した場合に、ネットワーク・メッセージに関するトレース情報が、この TraceFile パラメーターに指定したファイルに書き込まれます。	soltrace.out (トレースがサーバーまたはクライアントのいずれで開始されたかに応じて、いずれかの現行作業ディレクトリーに書き込まれます)

Data Sources

表 232. *Data Sources* パラメーター

[Data Sources]	説明	ファクトリー値	アクセス・モード
logical name = network name, Description	クライアント・アプリケーションの solid.ini ファイルの中で、これらのパラメーターを使用すると、solidDB サーバーに論理名を与えることができます。		N/A

索引

日本語, 数字, 英字, 特殊文字の順に配列されています。なお, 濁音と半濁音は清音と同等に扱われています。

[ア行]

[カ行]

[サ行]

[タ行]

[ナ行]

[ハ行]

[マ行]

[ラ行]

A

B

C

D

E

F

H

I

J

L

M

N

O

P

Q

R

S

T

U

W

Y

特記事項

Solid Information Technology Ltd. または International Business Machines Corporation の書面による明示的な許可がある場合を除き、本製品のいかなる部分も、いかなる方法においても使用することはできません。

本製品は、米国特許 6144941、 7136912、 6970876、 7139775、 6978396、 7266702、 7406489、および 7502796 により保護されています。

本製品は、米国輸出規制品目分類番号 ECCN=5D992b に指定されています。

本書は米国 IBM が提供する製品およびサービスについて作成したものです。

本書に記載の製品、サービス、または機能が日本においては提供されていない場合があります。日本で利用可能な製品、サービス、および機能については、日本 IBM の営業担当員にお尋ねください。本書で IBM 製品、プログラム、またはサービスに言及していても、その IBM 製品、プログラム、またはサービスのみが使用可能であることを意味するものではありません。これらに代えて、IBM の知的所有権を侵害することのない、機能的に同等の製品、プログラム、またはサービスを使用することができます。ただし、IBM 以外の製品とプログラムの操作またはサービスの評価および検証は、お客様の責任で行っていただきます。

IBM は、本書に記載されている内容に関して特許権 (特許出願中のものを含む) を保有している場合があります。本書の提供は、お客様にこれらの特許権について実施権を許諾することを意味するものではありません。実施権についてのお問い合わせは、書面にて下記宛先にお送りください。

〒242-8502
神奈川県大和市下鶴間1623番14号
日本アイ・ビー・エム株式会社
法務・知的財産
知的財産権ライセンス渉外

以下の保証は、国または地域の法律に沿わない場合は、適用されません。 IBM およびその直接または間接の子会社は、本書を特定物として現存するままの状態で提供し、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任を負わないものとします。国または地域によっては、法律の強行規定により、保証責任の制限が禁じられる場合、強行規定の制限を受けるものとします。

この情報には、技術的に不適切な記述や誤植を含む場合があります。本書は定期的に見直され、必要な変更は本書の次版に組み込まれます。 IBM は予告なしに、随時、この文書に記載されている製品またはプログラムに対して、改良または変更を行うことがあります。

本書において IBM 以外の Web サイトに言及している場合がありますが、便宜のため記載しただけであり、決してそれらの Web サイトを推奨するものではありません。それらの Web サイトにある資料は、この IBM 製品の資料の一部ではありません。それらの Web サイトは、お客様の責任でご使用ください。

IBM は、お客様が提供するいかなる情報も、お客様に対してなんら義務も負うことのない、自ら適切と信ずる方法で、使用もしくは配布することができるものとします。

本プログラムのライセンス保持者で、(i) 独自に作成したプログラムとその他のプログラム (本プログラムを含む) との間での情報交換、および (ii) 交換された情報の相互利用を可能にすることを目的として、本プログラムに関する情報を必要とする方は、下記に連絡してください。

IBM Canada Limited
Office of the Lab Director
8200 Warden Avenue
Markham, Ontario
L6G 1C7
CANADA

本プログラムに関する上記の情報は、適切な使用条件の下で使用することができますが、有償の場合もあります。

本書で説明されているライセンス・プログラムまたはその他のライセンス資料は、IBM 所定のプログラム契約の契約条項、IBM プログラムのご使用条件、またはそれと同等の条項に基づいて、IBM より提供されます。

この文書に含まれるいかなるパフォーマンス・データも、管理環境下で決定されたものです。そのため、他の操作環境で得られた結果は、異なる可能性があります。一部の測定が、開発レベルのシステムで行われた可能性がありますが、その測定値が、一般に利用可能なシステムのものと同じである保証はありません。さらに、一部の測定値が、推定値である可能性があります。実際の結果は、異なる可能性があります。お客様は、お客様の特定の環境に適したデータを確かめる必要があります。

IBM 以外の製品に関する情報は、その製品の供給者、出版物、もしくはその他の公に利用可能なソースから入手したものです。IBM は、それらの製品のテストは行っておりません。したがって、他社製品に関する実行性、互換性、またはその他の要求については確証できません。IBM 以外の製品の性能に関する質問は、それらの製品の供給者にお願いします。

IBM の将来の方向または意向に関する記述については、予告なしに変更または撤回される場合があり、単に目標を示しているものです。

本書には、日常の業務処理で用いられるデータや報告書の例が含まれています。より具体性を与えるために、それらの例には、個人、企業、ブランド、あるいは製品などの名前が含まれている場合があります。これらの名称はすべて架空のものであり、名称や住所が類似する企業が実在しているとしても、それは偶然にすぎません。

著作権使用許諾:

本書には、様々なオペレーティング・プラットフォームでのプログラミング手法を例示するサンプル・アプリケーション・プログラムがソース言語で掲載されています。お客様は、サンプル・プログラムが書かれているオペレーティング・プラットフォームのアプリケーション・プログラミング・インターフェースに準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。このサンプル・プログラムは、あらゆる条件下における完全なテストを経ていません。従って IBM は、これらのサンプル・プログラムについて信頼性、利便性もしくは機能性があることをほのめかしたり、保証することはできません。

それぞれの複製物、サンプル・プログラムのいかなる部分、またはすべての派生的創作物にも、次のように、著作権表示を入れていただく必要があります。

© (お客様の会社名) (西暦年)。このコードの一部は、IBM Corp. のサンプル・プログラムから取られています。

商標

IBM、IBM ロゴ、ibm.com®、Solid、solidDB、InfoSphere™、DB2®、Informix®、および WebSphere は、International Business Machines Corporation の米国およびその他の国における商標です。これらおよび他の IBM 商標に、この情報の最初に現れる個所で商標表示 (® または ™) が付されている場合、これらの表示は、この情報が公開された時点で、米国において、IBM が所有する登録商標またはコモン・ロー上の商標であることを示しています。このような商標は、その他の国においても登録商標またはコモン・ロー上の商標である可能性があります。現時点での IBM の商標リストについては、「Copyright and trademark information」(www.ibm.com/legal/copytrade.shtml) をご覧下さい。

Java およびすべての Java 関連の商標およびロゴは Sun Microsystems, Inc.の米国およびその他の国における商標です。

Linux は、Linus Torvalds の米国およびその他の国における商標です。

Microsoft および Windows は、Microsoft Corporation の米国およびその他の国における商標です。

UNIX は、The Open Group の米国およびその他の国における登録商標です。

他の会社名、製品名およびサービス名等はそれぞれ各社の商標です。

Printed in Japan

SC88-8167-00

日本アイ・ビー・エム株式会社
〒103-8510 東京都中央区日本橋箱崎町19-21