FUJITSU

Fujitsu
Enterprise Postgres 16

アプリケーション開発ガイド

Windows/Linux

J2UL-2958-01Z0(00)
2024年7月

まえがき

本書の目的

本書は、Fujitsu Enterprise Postgresのアプリケーション開発者ガイドです。

本書の読者

本書は、Fujitsu Enterprise Postgresを利用したアプリケーションを開発される方を対象としています。本書では、Fujitsu Enterprise Postgresが提供するインタフェースのうち、PostgreSQLを拡張したインタフェースについて説明しています。

なお、本書は、以下についての一般的な知識があることを前提に書かれています。

L

- PostgreSQL
- SQL
- Linux

W

- PostgreSQL
- SQL
- Windows

本書の構成

本書の構成と内容は以下のとおりです。

第1章 アプリケーション開発機能の概要

Fujitsu Enterprise Postgresでのアプリケーション開発の概要を説明しています。

第2章 JDBCドライバ

JDBCドライバの利用方法について説明しています。

第3章 ODBCドライバ

ODBCドライバの利用方法について説明しています。

第4章 .NET Data Provider

.NET Data Providerの利用方法について説明しています。

第5章 C言語用ライブラリ(libpq)

C言語アプリケーションの利用方法について説明しています。

第6章 C言語による埋め込みSQL

C言語による埋め込みSQLの利用方法について説明しています。

第7章 COBOL言語による埋め込みSQL

COBOL言語による埋め込みSQLの利用方法について説明しています。

第8章 SQL リファレンス

SQLリファレンスを記載しています。

第9章 Oracleデータベースとの互換性

Oracleデータベースとの互換機能について説明しています。

第10章 アプリケーションの接続先切り替え機能

アプリケーションの接続先切り替え機能について説明しています。

第11章 Vertical Clustered Index(VCI)を利用した検索

カラム型インデックスVertical Clustered Index(VCI)を利用した検索について説明しています。

付録A アプリケーション開発における注意事項

アプリケーション開発における注意事項について説明しています。

付録B Oracleデータベースとの機能差異や記述差異に伴う移行手順

Oracleデータベースとの互換性に記載している範囲におけるFujitsu Enterprise Postgresへの移行手順を説明しています。

付録C Oracleデータベースとの互換機能が利用するテーブル

Oracleデータベースとの互換機能が利用するテーブルについて説明しています。

付録D ECOBPG - COBOL言語による埋め込みSQL

COBOL言語による埋め込みSQLを利用したアプリケーション開発について説明しています。

付録E 定量制限

定量制限について説明しています。

付録F リファレンス

各インタフェースのリファレンスを記載しています。

輸出管理規制について

本ドキュメントを輸出または第三者へ提供する場合は、お客様が居住する国および米国輸出管理関連法規等の規制をご確認のうえ、必要な手続きをおとりください。

出版年月および版数

2024年　7月　　初版

著作権

目　次

第1章 アプリケーション開発機能の概要

Fujitsu Enterprise Postgresが提供するアプリケーション開発用のインタフェースは、PostgreSQLと完全互換です。

Fujitsu Enterprise Postgresは、PostgreSQLのインタフェースに加え、さらに以下の拡張インタフェースを提供します。

- 各国語データのサポート

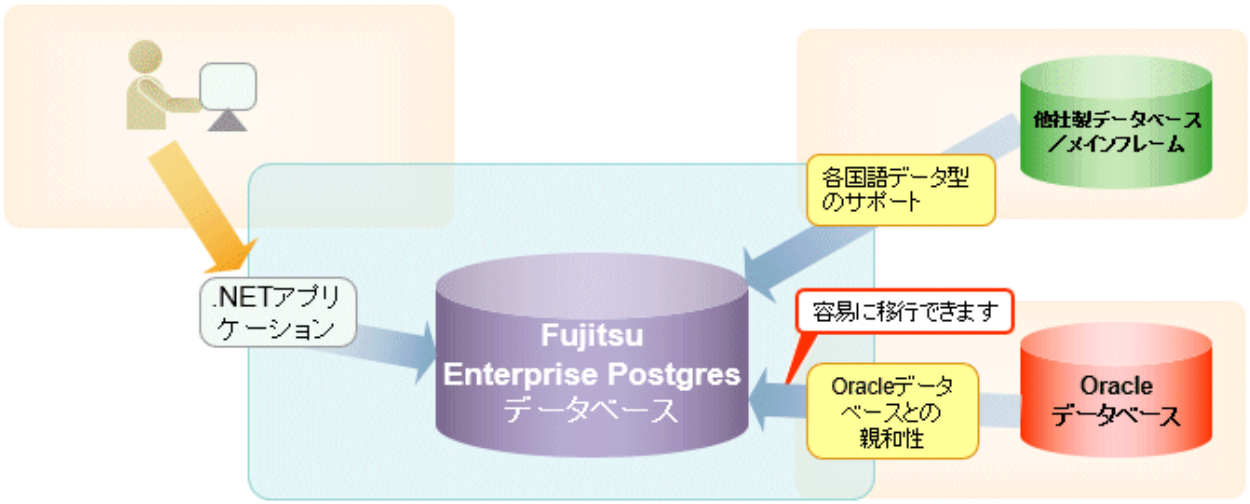
  メインフレームや他社製データベースからの移行性を確保するため、各国語文字列をサポートするデータ型を提供します。
  各国語文字列は、クライアントアプリケーションの各言語から利用できます。

  詳細は、“1.1 各国語データのサポート”を参照してください。

- Oracleデータベースとの互換性

  Oracleデータベースとの互換機能を提供します。互換機能を利用することにより、既存アプリケーションの改修を局所化し、Openインタフェースへ容易に移行できます。

  詳細は、“1.2 Oracleデータベースとの互換性”を参照してください。



- アプリケーションの接続先切り替え機能

  冗長化構成の複数のサーバに対して、接続対象がどのサーバであるかを意識せずにサーバへの接続を可能とする、アプリケーションの接続切り替え機能を提供します。

  詳細は、“1.3 アプリケーションの接続先切り替え機能”を参照してください。

- Vertical Clustered Index(VCI)を利用した検索

  大量の行に対する集計処理において、以下の機能を提供することで、検索処理の高速化を実現します。

  ― カラム型のインデックスVertical Clustered Index(VCI)

  ― データのメモリレジデント機能

  本機能は、Advanced Editionのみで使用できます。

  詳細は、“第11章 Vertical Clustered Index(VCI)を利用した検索”を参照してください。

1.1 各国語データのサポート

各国語文字を扱うデータ型としてNCHAR型を提供しています。

また、Fujitsu Enterprise PostgresのpgAdminでは、NCHAR型が使用できます。

ポイント

- NCHAR型はデータベースの文字セットがUTF-8の場合のみ使用できます。
- CHAR型が使用できる箇所(関数引数など)はNCHAR型も使用できます。
- アプリケーションでデータベースのNCHAR型のデータを扱う場合、データの形式はデータベースのCHAR型のデータと同じです。そのため、アプリケーションでデータベースのNCHAR型の列に格納されたデータを扱う場合は、データベースのCHAR型の列に格納されたデータと同様に使用できます。

注意

NCHAR型データをCHAR型にキャストするためには、以下の注意が必要です。

- 長さが異なるNCHAR型データの比較を行う場合、CHAR型データとして処理するため長さが短い方のNCHAR型データにASCII表記の空白が埋められます。
- 文字セットによっては、1.5倍～2倍にデータサイズが大きくなることがあります。
- 列の別名に“varchar”を付ける場合にはAS句で指定してください。

1.1.1 定数

記述形式

{ N | n }'[*各国語文字* [...]]'

一般規則

各国語文字列定数は、'N'または'n'および単一引用符(')と単一引用符(')で括られた任意の各国語文字の並びです。例えば、N'あいうえお' です。

データ型は各国語文字列型です。

1.1.2 データ型

記述形式

{ NATIONAL CHARACTER | NATIONAL CHAR | NCHAR } [VARYING][(*長さ*)]

NCHAR型の列のデータ型は以下となります。

データ型指定形式	指定の意味
NATIONAL CHARACTER(n) NATIONAL CHAR(n) NCHAR(n)	固定長で長さn文字の各国語文字列 (n)を省略すると(1)と同じになります。 nは、0より大きい正の整数です。
NATIONAL CHARACTER VARYING(n) NATIONAL CHAR VARYING(n) NCHAR VARYING(n)	可変長で長さ最大n文字までの各国語文字列 (n)を省略すると、どのような長さの各国語文字列でも受け付けます。 nは、0より大きい正の整数です。

一般規則

NCHARは、各国語文字列型のデータ型です。長さには文字数を指定します。

各国語文字列型の長さは次のようになります。

- VARYINGが指定されていないとき、各国語文字列の長さは固定長であり、長さで指定した値となります。
- VARYINGが指定されているとき、各国語文字列の長さは可変長となります。
  このとき、下限は0、上限は長さで指定した値となります。
- NATIONAL CHARACTERとNATIONAL CHARおよびNCHARは同じ意味です。

格納される各国語文字列が宣言された上限よりも短い場合、NCHARの値は各国語文字の空白文字が埋められ、NCHAR VARYINGの値はそのまま格納されます。

格納できる文字の上限は約1ギガバイトです。

1.1.3 関数と演算子

比較演算子

比較演算子にNCHAR型およびNCHAR VARYING型を指定する場合、NCHAR型およびNCHAR VARYING型同士でのみ比較可能です。

文字列関数と演算子

CHAR型が指定可能な文字列関数と演算子はすべてNCHAR型も指定可能です。また、文字列関数と演算子の動作はCHAR型と同一となります。

パターンマッチ(LIKE、SIMILAR TO正規表現、POSIX正規表現)

NCHAR型およびNCHAR VARYING型に対してパターンマッチを行う際に指定するパターンは、パーセント記号‘%’と下線文字‘_’を指定します。

下線文字‘_’は任意の各国語文字1文字との一致を意味し、パーセント記号‘%’は0文字以上の各国語文字の並びとの一致を意味します。

1.2 Oracleデータベースとの互換性

Oracleデータベースとの互換性を強化するため、以下の機能を拡張しています。

- 問合せ(外部結合演算子(+)、DUAL表)
- 関数(DECODE、SUBSTR、NVL)
- パッケージ(DBMS_OUTPUT、UTL_FILE、DBMS_SQL)

Oracleデータベース互換機能の詳細は、“第9章 Oracleデータベースとの互換性”を参照してください。

1.3 アプリケーションの接続先切り替え機能

アプリケーションの接続先切り替え機能とは、冗長化構成の複数のサーバに対して、接続対象のサーバがどのサーバであるかを意識せずに接続することができるようにする機能です。

アプリケーションの接続先切り替え機能の詳細は、“第10章 アプリケーションの接続先切り替え機能”を参照してください。

1.3.1 データベース多重化機能との連携

データベース多重化機能を利用する場合、アプリケーションの接続先切り替え機能により、アプリケーションは、データベースが多重化されていることを意識することなくデータベースへの接続が行えます。

参照

データベース多重化機能については、“クラスタ運用ガイド(データベース多重化編)”を参照してください。

1.4 アプリケーションの互換に関する注意事項

Fujitsu Enterprise Postgresがバージョンアップする場合、機能改善や機能拡張にともなってアプリケーションに影響するような変更が発生することがあります。

したがって、アプリケーションの開発を行う場合は、新しいバージョンのFujitsu Enterprise Postgresにアップグレードした場合でも互換性を維持できるように、以下の注意が必要です。

- 実行結果の確認方法に関する注意
- システムカタログの参照方法に関する注意
- 関数の使用方法に関する注意

1.4.1 実行結果の確認方法に関する注意

アプリケーション内で使用するSQL文や、開発で利用するコマンドの実行結果は、メッセージ中に出力されるSQLSTATEを参照して確認してください。

参照

メッセージの出力内容については、“メッセージ集”を参照してください。

SQLSTATEについては、“PostgreSQL Documentation”の“Appendixes”の“PostgreSQL Error Codes”を参照してください。

1.4.2 システムカタログの参照方法に関する注意

Fujitsu Enterprise Postgresのシステムの情報やデータベースのオブジェクトの情報を取得するために、システムカタログが利用できます。

しかし、システムカタログは、今後のFujitsu Enterprise Postgresのバージョンアップにともない変更されることがあります。また、システムカタログはFujitsu Enterprise Postgresに固有の情報を返却するものが多く存在します。

そのため、可能な限り、標準SQLで定義されている情報スキーマ(information_schema)を参照するようにしてください。ただし、列が追加されることもあるため、選択リストに“*”を指定した問い合わせを使用しないようにすることが必要です。

参照

情報スキーマについては、“PostgreSQL Documentation”の“Client Interfaces”の“The Information Schema”を参照してください。

情報スキーマにはない固有の情報を取得するには、システムカタログを参照する必要がありますが、この場合は、システムカタログを参照するビューを定義し、アプリケーションではシステムカタログを直接参照せずに、ビューを参照するようにしてください。ただし、ビューの定義では、ビュー名の後ろに明示的に列名を指定して定義する必要があります。

以下はビューの定義例と使用例です。

例

```
CREATE VIEW my_tablespace_view(spcname) AS SELECT spcname FROM pg_tablespace;
SELECT * FROM my_tablespace_view V1, pg_tables T1 WHERE V1.spcname = T1.tablespace;
```

これにより、システムカタログに変更が入った場合、アプリケーションを変更することなく、ビューを変更するだけで対応することができます。

以下は、ビューを再定義して変更が入らなかったかのように対処している例です。

列名 spcname がspacename に変更されたことにともない、システムカタログ pg_tablespaceを再定義しています。

 例

```
DROP VIEW my_tablespace_view;
CREATE VIEW my_tablespace_view(spcname) AS SELECT spacename FROM pg_tablespace;
```

1.4.3 関数の使用方法に関する注意

Fujitsu Enterprise Postgresがデフォルトで提供する関数を利用することで、様々な演算や操作、情報の取得をSQL文で行うことができます。

しかし、システムに関する情報を取得する関数や、統計情報に関する関数など、Fujitsu Enterprise Postgresの内部に関係する関数は、今後のFujitsu Enterprise Postgresのバージョンアップにともない変更されることがあります。

そのため、これらの関数を使用する際には、新たに関数を定義して、アプリケーションでは新しく定義した関数を使用するようにしてください。

以下は関数の定義例と使用例です。

 例

```
CREATE FUNCTION my_func(relid regclass) RETURNS bigint LANGUAGE SQL AS 'SELECT pg_relation_size(relid)';
SELECT my_func(2619);
```

これにより、関数に変更が入った場合、アプリケーションを変更することなく、関数を再定義するだけで対応することができます。

以下は、関数を再定義して変更が入らなかったかのように対処している例です。

引数が追加された関数 pg_relation_sizeを再定義しています。

 例

```
DROP FUNCTION my_func(regclass);
CREATE FUNCTION my_func(relid regclass) RETURNS bigint LANGUAGE SQL AS 'SELECT pg_relation_size(relid,$$main$
$)';
```

第2章 JDBCドライバ

JDBCドライバの利用方法について説明します。

2.1 開発環境

JDBCドライバを使用したアプリケーションの開発、および実行環境について説明します。

2.1.1 JDKまたはJREとの組合せ

JDBCドライバが動作可能なJDKまたはJREとの組合せについては、“導入ガイド(クライアント編)”を参照してください。

2.2 セットアップ

JDBCドライバを使用するための環境設定、および通信データの暗号化方法について説明します。

2.2.1 環境設定

JDBCドライバの実行環境として、環境変数CLASSPATHの設定が必要です。

JDBCドライバファイルの名前は以下のとおりです。

```
postgresql-jdbc42.jar
```

JDK 8、JRE 8、JDK 11、JRE 11、JDK 17またはJRE 17の、どの環境を使用する場合でも同じです。

以降では、環境変数CLASSPATHの設定例を説明します。

なお、“<*x*>”は製品のバージョンを示します。

L

- Linuxの場合
  - 設定例(TCシェル)

```
setenv CLASSPATH /opt/fsepv<x>client64/jdbc/lib/postgresql-jdbc42.jar:${CLASSPATH}
```

  - 設定例(bash)

```
CLASSPATH=/opt/fsepv<x>client64/jdbc/lib/postgresql-jdbc42.jar:$CLASSPATH;export CLASSPATH
```

W

- Windows(32ビット)の場合
  - 設定例

```
set CLASSPATH=C:¥Program Files¥Fujitsu¥fsepv<x>client32¥JDBC¥lib¥postgresql-jdbc42.jar;%CLASSPATH%
```

- Windows(64ビット)の場合
  - 設定例(Fujitsu Enterprise Postgres Client 32bitをインストールした場合)

```
set CLASSPATH=C:¥Program Files (x86)¥Fujitsu¥fsepv<x>client32¥JDBC¥lib¥postgresql-jdbc42.jar;
%CLASSPATH%
```

  - 設定例(Fujitsu Enterprise Postgres Client 64bitをインストールした場合)

```
set CLASSPATH=C:¥Program Files¥Fujitsu¥fsepv<x>client64¥JDBC¥lib¥postgresql-jdbc42.jar;%CLASSPATH%
```

2.2.2 メッセージの言語およびアプリケーションが使用する符号化方式の設定

JDBCドライバを利用する場合は、JDBCドライバが自動的にクライアント側の符号化方式をUTF-8に設定するので、符号化方式を設定することはできません。

参照

符号化方式については、PostgreSQL Documentationの“Server Administration”の“Automatic Character Set Conversion Between Server and Client”を参照してください。

言語の設定

アプリケーション実行環境の言語設定は、データベースサーバのメッセージロケールの設定と合わせる必要があります。

言語の設定は、システムプロパティ“user.language”で設定します。

例

システムプロパティを指定したJavaコマンドの起動例

```
java -Duser.language=ja TestClass1
```

2.2.3 通信データを暗号化する場合の設定

通信データの暗号化機能を利用してデータベースサーバと接続する場合は、以下のように設定してください。

通信データを暗号化してサーバに接続する設定

通信データを暗号化する場合のアプリケーションの作成方法を説明します。

暗号化する場合、sslパラメータのプロパティを“true”に設定します。sslパラメータのデフォルトは“false”です。

sslを“true”に設定すると、sslmodeは内部的に“verify-full”として扱われます。

例

ー 設定例1

```
String url = "jdbc:postgresql://sv1/test";
Properties props = new Properties();
props.setProperty("user","fsepuser");
props.setProperty("password","secret");
props.setProperty("ssl","true");
props.setProperty("sslfactory","org.postgresql.ssl.DefaultJavaSSLFactory");
Connection conn = DriverManager.getConnection(url, props);
```

ー 設定例2

```
String url = "jdbc:postgresql://sv1/test?
user=fsepuser&password=secret&ssl=true&sslfactory=org.postgresql.ssl.DefaultJavaSSLFactory";
Connection conn = DriverManager.getConnection(url);
```

さらに、データベースサーバの成りすましから防御するためには、Javaに含まれるkeytoolコマンドを使用して、CA証明書をJavaのキーストアにインポートする必要があります。また、その際はsslfactoryパラメータに“org.postgresql.ssl.DefaultJavaSSLFactory”を指定してください。

詳細については、JDKのドキュメント、または、Oracle社のWebサイトで参照してください。

注意

アプリケーションの接続先切り替え機能を利用するなど、DriverManagerクラスの接続文字列またはデータソースにsslmodeパラメータを指定する場合、sslパラメータの設定は不要です。sslパラメータを設定した場合、sslmodeパラメータの設定値が有効となります。

参照

通信データの暗号化についての詳細は、“PostgreSQL Documentation”の“Server Administration”の“Secure TCP/IP Connections with SSL”を参照してください。

2.3 データベースへの接続

データベースへの接続方法について説明します。

- DriverManagerクラスを使用する場合
- PGConnectionPoolDataSourceクラスを使用する場合
- PGXADataSourceクラスを使用する場合

注意

接続文字列の“protocolVersion”には、“V2”を指定しないでください。

2.3.1 DriverManagerクラスを使用する場合

DriverManagerクラスを使用してデータベースに接続するには、JDBCドライバをロードしてから、DriverManagerクラスのAPIにURIで表現された接続文字列を指定します。

JDBCドライバのロード

org.postgresql.Driver を指定します。

接続文字列

URI接続方式は、以下の方法で行ってください。

```
jdbc:postgresql://host:port/database?
user=user&password=password1&loginTimeout=loginTimeout&socketTimeout=socketTimeout
```

引数	説明
host	接続先のホスト名を指定します。 省略した場合は、localhostとなります。
port	データベースサーバのポート番号を指定します。 省略した場合は、27500となります。
database	データベース名を指定します。
user	データベースへ接続するユーザー名を指定します。 省略した場合は、そのアプリケーションを実行しているユーザーのオペレーティングシステム上の名前と同じです。
password	パスワードによる認証を必要とした場合に、パスワードを指定します。
loginTimeout	接続時のタイムアウト時間を指定します。

引数	説明
	単位は秒で0～2147483647の値を指定します。0、および不当な値を指定した場合、省略した場合は無制限です。 指定された時間内にコネクションが接続できなかった場合はエラーとなります。
socketTimeout	サーバとの通信時のタイムアウト時間を指定します。 単位は秒で0～2147483647の値を指定します。0、および不当な値を指定した場合、省略した場合は無制限です。 指定された時間内にサーバからのデータが受信できなかった場合は、エラーとなります。

例

アプリケーションの記述例

```
import java.sql.*;
…
Class.forName("org.postgresql.Driver");
String url = "jdbc:postgresql://sv1:27500/mydb?
user=myuser&password=myuser01&loginTimeout=20&socketTimeout=20";
Connection con = DriverManager.getConnection(url);
```

2.3.2 PGConnectionPoolDataSourceクラスを使用する場合

データソースを使用してデータベースに接続するには、データソースのプロパティに接続情報を指定します。

メソッドの説明

引数	説明
setServerName	接続先のホスト名を指定します。 省略した場合は、localhostとなります。
setPortNumber	データベースサーバのポート番号を指定します。 省略した場合は、27500となります。
setDatabaseName	データベース名を指定します。
setUser	データベースのユーザー名を指定します。 デフォルトは、そのアプリケーションを実行しているユーザーのオペレーティングシステム上の名前と同じです。
setPassword	サーバがパスワードによる認証を必要とした場合に使用されるパスワードを指定します。
setLoginTimeout	接続時のタイムアウト時間を指定します。 単位は秒で0～2147483647の値を指定します。0、および不当な値を指定した場合、省略した場合は無制限です。 指定された時間内にコネクションが接続できなかった場合はエラーとなります。
setSocketTimeout	サーバとの通信時のタイムアウト時間を指定します。 単位は秒で0～2147483647の値を指定します。0、および不当な値を指定した場合、省略した場合は無制限です。 指定された時間内にサーバからのデータが受信できなかった場合は、エラーとなります。

例

アプリケーションの記述例

```
import java.sql.*;
import org.postgresql.ds.PGConnectionPoolDataSource;
…
PGConnectionPoolDataSource source = new PGConnectionPoolDataSource();
source.setServerName("sv1");
source.setPortNumber(27500);
source.setDatabaseName("mydb");
source.setUser("myuser");
source.setPassword("myuser01");
source.setLoginTimeout(20);
source.setSocketTimeout(20);
…
Connection con = source.getConnection();
```

2.3.3 PGXADataSourceクラスを使用する場合

データソースを使用してデータベースに接続するには、データソースのプロパティに接続情報を指定します。

メソッドの説明

引数	説明
setServerName	接続先のホスト名を指定します。 省略した場合は、localhostとなります。
setPortNumber	データベースサーバのポート番号を指定します。 省略した場合は、27500となります。
setDatabaseName	データベース名を指定します。
setUser	データベースへ接続するユーザー名を指定します。 省略した場合は、そのアプリケーションを実行しているユーザーのオペレーティングシステム上の名前と同じです。
setPassword	パスワードによる認証を必要とした場合に、パスワードを指定します。
setLoginTimeout	接続時のタイムアウト時間を指定します。 単位は秒で0～2147483647の値を指定します。0、および不当な値を指定した場合、省略した場合は無制限です。 指定された時間内にコネクションが接続できなかった場合はエラーとなります。
setSocketTimeout	サーバとの通信時のタイムアウト時間を指定します。 単位は秒で0～2147483647の値を指定します。0、および不当な値を指定した場合、省略した場合は無制限です。 指定された時間内にサーバからのデータが受信できなかった場合は、エラーとなります。

例

アプリケーションの記述例

```
import java.sql.*;
import org.postgresql.xa.PGXADataSource;
…
PGXADataSource source = new PGXADataSource();
```

```
source.setServerName("sv1");
source.setPortNumber(27500);
source.setDatabaseName("mydb");
source.setUser("myuser");
source.setPassword("myuser01");
source.setLoginTimeout(20);
source.setSocketTimeout(20);
…
Connection con = source.getConnection();
```

2.4 アプリケーション開発

Fujitsu Enterprise Postgresと接続するアプリケーションの開発時に必要なデータ型に関して説明します。

2.4.1 アプリケーションのデータ型とデータベースのデータ型の関係

アプリケーションのデータ型と、データベースのデータ型の対応を以下に示します。

サーバのデータ型	javaのデータ型	java.sql.Typesで規定されるデータ型
character	String	java.sql.Types.CHAR
national character	String	java.sql.Types.NCHAR
character varying	String	java.sql.Types.VARCHAR
national character varying	String	java.sql.Types.NVARCHAR
text	String	java.sql.Types.VARCHAR
bytea	byte[]	java.sql.Types.BINARY
smallint	short	java.sql.Types.SMALLINT
integer	int	java.sql.Types.INTEGER
bigint	long	java.sql.Types.BIGINT
smallserial	short	java.sql.Types.SMALLINT
serial	int	java.sql.Types.INTEGER
bigserial	long	java.sql.Types.BIGINT
real	float	java.sql.Types.REAL
double precision	double	java.sql.Types.DOUBLE
numeric	java.math.BigDecimal	java.sql.Types.NUMERIC
decimal	java.math.BigDecimal	java.sql.Types.DECIMAL
money	String	java.sql.Types.OTHER
date	java.sql.Date	java.sql.Types.DATE
time with time zone	java.sql.Time	java.sql.Types.TIME
time without time zone	java.sql.Time	java.sql.Types.TIME
timestamp without time zone	java.sql.Timestamp	java.sql.Types.TIMESTAMP
timestamp with time zone	java.sql.Timestamp	java.sql.Types.TIMESTAMP
interval	org.postgresql.util.PGInterval	java.sql.Types.OTHER
boolean	boolean	java.sql.Types.BIT
bit	boolean	java.sql.Types.BIT
bit varying	org.postgresql.util.Pgobject	java.sql.Types.OTHER

サーバのデータ型	javaのデータ型	java.sql.Typesで規定されるデータ型
oid	long	java.sql.Types.BIGINT
xml	java.sql.SQLXML	java.sql.Types.SQLXML
array	java.sql.Array	java.sql.Types.ARRAY
uuid	java.util.UUID	java.sql.Types.OTHER
point	org.postgresql.geometric.Pgpoint	java.sql.Types.OTHER
box	org.postgresql.geometric.Pgbox	java.sql.Types.OTHER
lseg	org.postgresql.geometric.Pglseg	java.sql.Types.OTHER
path	org.postgresql.geometric.Pgpath	java.sql.Types.OTHER
polygon	org.postgresql.geometric.PGpolygon	java.sql.Types.OTHER
circle	org.postgresql.geometric.PGcircle	java.sql.Types.OTHER
json	org.postgresql.util.PGobject	java.sql.Types.OTHER
ネットワークアドレス型 (inet,cidr,macaddr, macaddr8)	org.postgresql.util.PGobject	java.sql.Types.OTHER
テキスト検索に関する型 (tsvector,tsquery)	org.postgresql.util.PGobject	java.sql.Types.OTHER
列挙型	org.postgresql.util.PGobject	java.sql.Types.OTHER
複合型	org.postgresql.util.PGobject	java.sql.Types.OTHER
範囲型	org.postgresql.util.PGobject	java.sql.Types.OTHER

すべてのサーバのデータ型に対して、ResultSetオブジェクトのgetString()メソッドを使用することができますが、この方法は同じデータ型に対して常に同じフォーマットの文字列を返すことを保証しません。

サーバのデータ型に合わせたgetterメソッド(例：getInt(),getTimestamp())によって得たjavaのデータ型のオブジェクトのtoString()メソッドを使用することで、JDBCの仕様に準拠したフォーマットの文字列を得ることができます。

2.4.2 ステートメントキャッシュ機能

ステートメントキャッシュ機能は、SQL文をコネクション単位でキャッシュすることで、次に同じ文字列のSQL文を実行するときに文の解析と作成を省略します。これによって、繰り返し実行されるループやメソッドの中で同じ文字列のSQL文を実行するような場合に性能が向上します。また、コネクションプーリング機能と組み合わせた場合にも性能の向上が期待できます。

キャッシュ登録の制御

ステートメントキャッシュ機能が有効の場合に、PreparedStatementクラスのsetPoolable(boolean)メソッドで、SQL文をキャッシュするかどうかを設定できます。

設定する値は、以下のとおりです。

false

ステートメントキャッシュ機能が有効でも、SQL文はキャッシュされません。

true

ステートメントキャッシュ機能が有効の場合、SQL文をキャッシュします。

2.4.3 データベース多重化運用時のアプリケーション作成

データベース多重化運用時のアプリケーション作成において考慮する事項を説明します。

参照

- データベース多重化運用についての詳細は、“クラスタ運用ガイド(データベース多重化編)”を参照してください。

- クラスタソフトウェアと連携したフェイルオーバ運用での、アプリケーション作成における考慮事項については、“クラスタ運用ガイド(PRIMECLUSTER編)”の“アプリケーションの作成”を参照してください。

2.4.3.1 アプリケーションの接続先切り替えが発生した場合のエラーと対処

データベース多重化運用時に、アプリケーションの接続先の切り替えが発生した場合、明示的にコネクションを切断し、コネクションの再接続またはアプリケーションを再実行してください。

以下に切り替え発生時のエラーと対処を示します。

状態		エラー情報(注)	対処
サーバダウン または Fujitsu Enterprise Postgresシステムダウン	アクセス中にダウンしたとき	57P01 08006 08007	切替え完了後、コネクションの再接続、またはアプリケーションを再実行してください。
	ダウン中にアクセスしたとき	08001	
スタンバイサーバへの切替え	アクセス中に切替えたとき	57P01 08006 08007	
	切替え中にアクセスしたとき	08001	

注: SQLExceptionのgetSQLState()の返却値となります。

第3章 ODBCドライバ

ODBCドライバを利用したアプリケーション開発について説明します。

3.1 開発環境

ODBCドライバを利用したアプリケーションは、ODBCインタフェースに対応したAccess、Excelなどのアプリケーション、およびVisual Basicなどのツールを利用して開発できます。

開発する環境については、これらのODBCインタフェースに対応したプログラム言語のマニュアルを参照してください。

Fujitsu Enterprise Postgresでは、ODBC 3.5をサポートしています。

3.2 セットアップ

Fujitsu Enterprise PostgresでODBCドライバを使用したアプリケーションを使用するためには、ODBCドライバであるPsqlODBCのセットアップが必要です。PsqlODBCはFujitsu Enterprise Postgresのクライアントパッケージに含まれます。

ODBCドライバの登録と、ODBCデータソースの登録方法を説明します。

3.2.1 ODBCドライバの登録

Linuxプラットフォームで ODBCドライバを使用する場合、以下の手順でODBCドライバを登録してください。

1. ODBCドライバマネージャ(unixODBC)およびlibtoolのインストール

   参考

   - Fujitsu Enterprise Postgresでは、unixODBCのVersion 2.3以降をサポートしています。

     以下のサイトから、unixODBCをダウンロードしてご利用ください。

     http://www.unixodbc.org/

   - unixODBCを実行するために、libtool 2.2.6以降のインストールが必要です。

     以下のサイトから、libtoolをダウンロードしてご利用ください。

     http://www.gnu.org/software/libtool/

   [注意]

   - ODBCドライバの動作についてはサポートします。
   - unixODBCの動作についてはサポート対象外です。

2. ODBCドライバの登録

   ODBCドライバマネージャ(unixODBC)のodbcinst.iniファイルを編集します。

   参考

   [odbcinst.iniファイルの格納先]

   ```
   <unixODBCインストールディレクトリ>/etc/odbcinst.ini
   ```

以下の内容を設定してください。

定義名	意味	設定値
[ドライバ名]	ODBCドライバの名前	ODBCドライバの名前を設定します。 アプリケーションの種類に応じて、以下の2つの文字列を選択し、これらを空白なしで連結した文字列を“[ ]”で囲んで設定してください。 ・ アプリケーションのアーキテクチャ “Fujitsu Enterprise Postgres<*Fujitsu Enterprise Postgres クライアント機能のバージョン*>x64” ・ アプリケーションが使用する符号化方式 ― Unicodeの場合(UTF-8のみ使用できます) “unicode” ― Unicode以外の場合 “ansi” 例: アプリケーションが使用する符号化方式がUnicodeの場合 “[Fujitsu Enterprise Postgres <*Fujitsu Enterprise Postgres クライアント機能のバージョン*>x64unicode]”
Description	ODBCドライバの説明	カレントのデータソースの補足の説明を指定します。任意の説明を設定してください。
Driver64	ODBCドライバのパス(64ビット)	ODBCドライバ(64ビット)のパスを設定します。 ・ 符号化方式がUnicodeの場合 *<Fujitsu Enterprise Postgres クライアント機能のインストールディレクトリ>*/odbc/lib/psqlodbcw.so ・ 符号化方式がUnicode以外の場合 *<Fujitsu Enterprise Postgres クライアント機能のインストールディレクトリ>*/odbc/lib/psqlodbca.so
FileUsage	データソースファイルの使用方法	1を指定してください。
Threading	コネクションプーリングのアトミック性の確保レベル	2を指定してください。

例

例内の“<*x*>”は、製品のバージョンを示します。

```
[Fujitsu Enterprise Postgres<x>x64unicode]
Description = Fujitsu Enterprise Postgres <x> x64 unicode driver
Driver64    = /opt/fsepv<x>client64/odbc/lib/psqlodbcw.so
FileUsage   = 1
Threading   = 2
```

3.2.2 ODBCデータソースの登録(Windows(R)の場合)

ODBCデータソースの登録方法を説明します。Windows(R)プラットフォームで、ODBCデータソースを登録する方法は以下の2通りの方法があります。

3.2.2.1 GUIを使用して登録する

[ODBC データソース アドミニストレータ]を起動し、ODBCデータソースを登録する方法を説明します。

ODBCデータソースの登録は、以下の手順で行います。

1. [ODBC データ ソースアドミニストレーター]を起動します。

   [ODBC データ ソースアドミニストレーター]はコントロールパネルの[管理ツール]にあります。

   注意

   64ビットのWindows(R)において、32ビットアプリケーション用のデータソースを登録する場合は、以下にある32ビット用のODBCアドミニストレータ(odbcad32.exe)を実行します。

   ```
   %SYSTEMDRIVE%¥WINDOWS¥SysWOW64¥odbcad32.exe
   ```

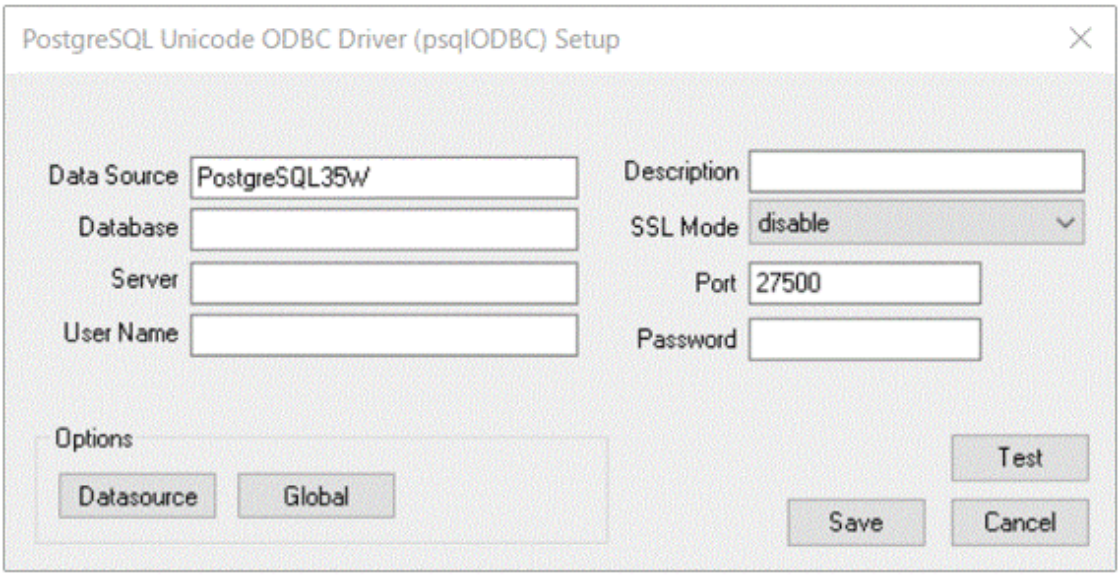
2. ODBCデータソースを現在のユーザーのみが使用する場合、[ユーザー DSN]タブを選択してください。同一のコンピュータを利用するすべてのユーザーが利用する場合は、[システム DSN]タブを選択してください。

3. [追加]ボタンをクリックします。

4. [データ ソースの新規作成]画面で、使用可能なODBCドライバの一覧の中から、以下のいずれかのドライバを選択し、[完了]ボタンをクリックします。“*x*”は、Fujitsu Enterprise Postgresクライアント機能のバージョンを示します。

   - Fujitsu Enterprise Postgres Unicode *x*
     アプリケーションの符号化方式として、Unicodeを使用する場合に選択します。
   - Fujitsu Enterprise Postgres ANSI *x*
     アプリケーションの符号化方式として、Unicode以外を使用する場合に選択します。

5. [PostgreSQL Unicode ODBC Driver (psqlODBC) Setup]画面が表示されますので、必要な項目を入力または選択します。また、必要な項目をすべて入力または選択したあと、[Save]ボタンをクリックします。



以下の内容を設定してください。

定義名	設定値
Data Source	ODBCドライバマネージャに登録するデータソース名を指定します。アプリケーションはここで指定した名前を選択してFujitsu Enterprise Postgresのデータベースに接続します。本パラメータは省略できません。 32バイト以内の以下の文字が指定できます。 ・ 各国語文字 ・ 英数字 ・ “_”、“<”、“>”、“+”、“`”、“|”、“~”、“‘”、“&”、“”””、“#”、“$”、“%”、“-”、“^”、“:”、“/”、“.”
Description	カレントのデータソースの補足の説明を指定します。255バイト以内の文字が指定できます。 ・ 各国語文字 ・ 英数字
Database	接続先のデータベース名を指定します。
SSLMode	通信を暗号化する場合に指定してください。デフォルトは無効に設定されています。 SSLModeの設定値は以下のとおりです。 ・ disable：非SSLで接続します。 ・ allow：非SSLで接続し、失敗したらSSLで接続します。 ・ prefer：SSLで接続し、失敗したら非SSLで接続します。 ・ require：必ずSSLで接続します。 ・ verify-ca：SSLで接続し、信頼できるCAから発行された証明書を使用します。(注1)

定義名	設定値
	・ verify-full：SSLで接続し、信頼できるCAから発行された証明書を使用してサーバのホスト名が証明書と一致するかを検証します。(注1)
Server	接続したいデータベースが存在するデータベースサーバのホスト名を63バイト以内で指定します。 本パラメータは省略できません。
Port	リモートアクセスで使用するポート番号を指定します。 省略値は、27500です。
Username(注2)	データベースにアクセスするユーザーを指定します。
Password(注2)	データベースにアクセスするユーザーのパスワードを指定します。

注1) “verify-ca”または“verify-full”を指定する場合、CA証明書ファイルをOSのシステム環境変数PGSSLROOTCERTで以下のように指定してください。

例）

```
変数名：PGSSLROOTCERT
変数値：CA証明書ファイル
```

注2) セキュリティのため、“Username”および“Password”は、アプリケーションで指定してください。

3.2.2.2 コマンドを使用して登録する

コマンドを使用してODBCデータソースを登録する方法を説明します。

ODBCデータソースの登録には、Microsoft社が提供する以下のツールを利用します。

- ODBCConf.exe
- Add-OdbcDsn

ツールの使用方法の詳細は、Microsoft社のMSDNライブラリを参照してください。

ODBCConf.exeを利用する場合

ODBCConf.exeはWindows(R)のすべてのプラットホームでサポートされているツールです。

指定形式

```
ODBCConf.exe /A {データソースタイプ "ODBCドライバ名" "オプション名=値[|オプション名=値…]"} [/Lv ファイル名]
```

指定形式の詳細とパラメータについては、Microsoft社のMSDNライブラリを参照してください。

説明:

以下の内容を設定してください。

定義名	設定値
データソースタイプ	データソースのタイプを指定します。 ・ “CONFIGSYSDSN”：システムのデータソースを作成します。管理者権限のあるユーザーのみ指定できます。データソースは、同一のコンピュータを利用するすべてのユーザーが利用可能です。 ・ “CONFIGDSN”：ユーザーのデータソースを作成します。データソースは、現在のユーザーのみ利用可能です。

定義名	設定値
	注意 “CONFIGSYSDSN”を指定する場合、管理者モードのコマンドプロンプトでコマンドを実行する必要があります。
ODBCドライバ名	使用するODBCドライバ名を指定します。以下のいずれかを指定してください。 ・“Fujitsu Enterprise Postgres Unicode *<Fujitsu Enterprise Postgres クライアント機能のバージョン>*” アプリケーションの符号化方式として、Unicodeを使用する場合に指定します。 ・“Fujitsu Enterprise Postgres ANSI *<Fujitsu Enterprise Postgres クライアント機能のバージョン>*” アプリケーションの符号化方式として、Unicode以外を使用する場合に指定します。
オプション名	以下の項目の設定が必要です。 ・“DSN”:データソース名を指定します。 ・“Servername”:データベースサーバのホスト名を指定します。 ・“Port”:データベースに接続するためのポート番号を指定します。 ・“Database”:データベース名を指定します。 以下の項目は、任意で指定してください。 ・“UID”:ユーザーID ・“Password”:パスワード ・“SSLMode”:通信を暗号化する場合に指定してください。デフォルトは無効に設定されています。SSLModeの設定方法は、“3.2.2.1 GUIを使用して登録する”の手順5.の表にあるSSLModeの説明を参照してください。
ファイル名	データソースを作成するとき、プロセス情報をファイルに出力することができます。省略可能です。

例

例内の“*<x>*”は、製品のバージョンを示します。

```
ODBCConf.exe /A {CONFIGSYSDSN "Fujitsu Enterprise Postgres Unicode <x>" "DSN=odbcconf1|Servername=sv1|
Port=27500|Database=db01|SSLMode=verify-ca"} /Lv log.txt
```

注意

セキュリティのため、ユーザーID(UID)およびパスワード(Password)は、アプリケーションで指定してください。

Add-OdbcDsnを利用する場合

Add-OdbcDsnは、PowerShellコマンドインタフェースで使用します。

指定形式

```
Add-OdbcDsn データソース名 -DriverName "ODBCドライバ名" -DsnType データソースタイプ -Platform OSアーキテクチャ -SetPropertyValue @("オプション名=値" [,"オプション名=値"…])
```

指定形式の詳細とパラメータについては、Microsoft社のMSDNライブラリを参照してください。

説明:

以下の内容を設定してください。

定義名	設定値
データソース名	データソース名として任意の名前を指定できます。
ODBCドライバ名	使用するODBCドライバ名を指定します。以下のいずれかを指定してください。 • "Fujitsu Enterprise Postgres Unicode <*Fujitsu Enterprise Postgres クライアント機能のバージョン*>" アプリケーションの符号化方式として、Unicodeを使用する場合に指定します。 • "Fujitsu Enterprise Postgres ANSI <*Fujitsu Enterprise Postgres クライアント機能のバージョン*>" アプリケーションの符号化方式として、Unicode以外を使用する場合に指定します。
データソースタイプ	データソースのタイプを指定します。 • "System":システムのデータソースを作成します。管理者権限のあるユーザーのみ指定できます。データソースは、同一のコンピュータを利用するすべてのユーザーが利用可能です。 • "User":ユーザーのデータソースを作成します。データソースは、現在のユーザーのみ利用可能です。 **注意** "System"を指定する場合、管理者モードのコマンドプロンプトでコマンドを実行する必要があります。
OSアーキテクチャ	システムのOSアーキテクチャを指定します。 • "32-bit":32ビットシステム • "64-bit":64ビットシステム
オプション名	以下の項目の設定が必要です。 • "Servername":データベースサーバのホスト名を指定します。 • "Port":データベースに接続するためのポート番号を指定します。 • "Database":データベース名を指定します。 以下の項目は、任意で指定してください。 • "SSLMode":通信を暗号化する場合に指定してください。デフォルトは無効に設定されています。SSLModeの設定方法は、"3.2.2.1 GUIを使用して登録する"の手順5.の表にあるSSLModeの説明を参照してください。 **注意** Add-OdbcDsnを利用する場合、オプション名として"UID"、"Password"を設定できません。ODBCConf.exeを利用する場合だけ利用できます。

例

例内の“<*x*>”は、製品のバージョンを示します。

```
Add-OdbcDsn odbcps1 -DriverName "Fujitsu Enterprise Postgres Unicode <x>" -DsnType System -Platform 32-bit -
SetPropertyValue @("Servername=sv1", "Port=27500", "Database=db01","SSLMode=verify-ca")
```

L

3.2.3 ODBCデータソースの登録(Linuxの場合)

Linuxプラットフォームで ODBCデータソースを登録する方法を説明します。

1. データソースの登録

   データソースの定義ファイルodbc.iniを編集します。

   参考

   [ODBCドライバマネージャ(unixODBC)のインストールディレクトリにあるファイルを編集する]

   ```
   <unixODBCインストールディレクトリ>/etc/odbc.ini
   ```

   または

   [HOMEディレクトリ配下に新しいファイルを作成する]

   ```
   ~/.odbc.ini
   ```

   ポイント

   <unixODBCインストールディレクトリ>配下を編集した場合、当該システムにログインするユーザーすべての共通設定として使用されます。HOMEディレクトリ(~/)配下に作成した場合、当該ユーザーのみが使用できる設定として使用されます。

   以下の内容を設定してください。

定義名	設定値
[データソース名]	ODBCデータソースに付与する名前を設定します。
Description	ODBCデータソース定義の説明を設定します。任意の説明を設定してください。
Driver	ODBCドライバの名前を設定します。この値は変更しないでください。 アプリケーションの種類に応じて、以下の2つの文字列を選択し、これらを空白なしで連結した文字列を設定してください。 ・ アプリケーションのアーキテクチャ “Fujitsu Enterprise Postgres<*Fujitsu Enterprise Postgres クライアント機能のバージョン*>x64” ・ アプリケーションが使用する符号化方式 － Unicodeの場合(UTF-8のみ使用できます) “unicode”

定義名	設定値
	ー Unicode以外の場合 “ansi” 例: アプリケーションが使用する符号化方式がUnicodeの場合 “Fujitsu Enterprise Postgres<*Fujitsu Enterprise Postgres クライアント機能のバージョン*>x64unicode”
Database	接続するデータベース名を指定します。
Servername	データベースサーバのホスト名を指定します。
Username	データベースに接続するユーザーIDを指定します。
Password	データベースに接続するユーザーのパスワードを指定します。
Port	データベースサーバのポート番号を指定します。 省略した場合は、27500となります。
SSLMode	通信の暗号化方法を指定します。SSLModeの設定値は以下のとおりです。 ・ disable:非SSLで接続します。 ・ allow:非SSLで接続し、失敗したらSSLで接続します。 ・ prefer:SSLで接続し、失敗したら非SSLで接続します。 ・ require:必ずSSLで接続します。 ・ verify-ca:SSLで接続し、信頼できるCAから発行された証明書を使用します。(注) ・ verify-full:SSLで接続し、信頼できるCAから発行された証明書を使用してサーバのホスト名が証明書と一致するかを検証します。(注)
ReadOnly	データベースを読み込み専用にするかどうかを指定します。 ・ 1:読み込み専用にします ・ 0:読み込み専用にしません

注) “verify-ca”または“verify-full”を指定する場合、CA証明書ファイルを環境変数PGSSLROOTCERTで以下のように指定してください。

例)

```
export PGSSLROOTCERT=<CA証明書ファイルの格納ディレクトリ>/root.crt
```

例

例内の“<*x*>”は、製品のバージョンを示します。

```
[MyDataSource]
Description    = Fujitsu Enterprise Postgres
Driver         = Fujitsu Enterprise Postgres<x>x64unicode
Database       = db01
Servername     = sv1
Port           = 27500
ReadOnly       = 0
```

注意

セキュリティのため、ユーザーID(UserName)およびパスワード(Password)は、アプリケーションで指定してください。

2. 環境変数の設定

   ODBCドライバを使用するアプリケーションを実行するためには、環境変数LD_LIBRARY_PATHに、以下をすべて設定してください。

   - *<unixODBCのインストールディレクトリ(注)>*/lib
   - *<libtoolのインストールディレクトリ(注)>*/lib

   注: unixODBCとlibtoolについて、インストールディレクトリを指定せずにインストールした場合は、/usr/localにインストールされます。

3.2.4 メッセージの言語およびアプリケーションが使用する符号化方式の設定

アプリケーション実行環境の言語の設定、およびアプリケーションが使用する符号化方式の設定について説明します。

言語の設定

アプリケーション実行環境の言語設定は、データベースサーバのメッセージロケールの設定と合わせる必要があります。

アプリケーションが出力するメッセージの中には、アプリケーション側のメッセージに、データベースサーバから送られたメッセージを埋め込む場合があります。このとき、アプリケーション側のメッセージは、アプリケーション側のメッセージロケールに従い、データベースサーバから送られるメッセージは、データベースサーバ側のメッセージロケールに従います。そのため、両方のメッセージロケールが一致していない場合には、言語や符号化方式が混在します。符号化方式が一致しない場合には、文字化けが発生します。

L

- Linuxの場合

  プロセスのロケールのLC_MESSAGESカテゴリが、データベースサーバのメッセージロケールと一致するように設定してください。環境変数を用いるなどいくつかの方法があります。詳細は、setlocale関数のオペレーティングシステムに付属するドキュメントを参照してください。

  例

  setlocale関数で“ja_JP.UTF-8”の指定例

```
setlocale(LC_ALL,"ja_JP.UTF-8");
```

  LC_ALLに指定することにより、LC_MESSAGEカテゴリに設定を適用しています。

W

- Windows(R)の場合

  OSのロケールをデータベースサーバのメッセージロケールの設定と合わせてください。

符号化方式の設定

アプリケーションに埋め込まれ、データベースに渡される符号化方式と、実行時のクライアント符号化方式の設定は同じにしてください。データベースサーバ側で正しく符号化方式を変換できなくなります。

アプリケーションの符号化方式は、以下のいずれかの方法で設定してください。

- 実行時の環境変数PGCLIENTENCODINGに設定する。
- 接続文字列のclient_encodingキーワードに設定する。
- PQsetClientEncoding関数を使って設定する。

参照

設定できる符号化方式を表す文字列は、“PostgreSQL Documentation”の“Server Administration”の“Supported Character Sets”を参照してください。

例えば、Unicode、8ビットの場合は、“UTF8”という文字列を設定します。

例

環境変数“PGCLIENTENCODING”に設定する場合

L

クライアントの符号化方式が“UTF8”の場合の設定例(Bash)

```
> PGCLIENTENCODING=UTF8; export PGCLIENTENCODING
```

W

クライアントの符号化方式が“UTF8”の場合の設定例

```
> set PGCLIENTENCODING=UTF8
```

注意

コマンドプロンプトに結果を出力する際、文字化けする場合があります。文字化けした場合は、コマンドプロンプトのフォントの設定を見直してください。

3.3 データベースへの接続

Access、ExcelまたはVisual Basicなど、ODBCインタフェースに対応したプログラム言語のマニュアルを参照してください。

3.4 アプリケーション開発

ODBCドライバを使用したアプリケーションの開発方法について説明します。

W

3.4.1 アプリケーションのコンパイル(Windows(R)の場合)

Access、ExcelまたはVisual Basicなど、ODBCインタフェースに対応したプログラム言語のマニュアルを参照してください。

注意

clコマンドは以下のいずれかのコードページで記述されたプログラムを入力として期待しているため、これらのコードページに変換してからコンパイルおよびリンクを行ってください(詳細は、Microsoftのマニュアルを参照してください)。

- ANSIコンソールコードページ(例:日本語の場合は Shift-JIS)
- BOM(Byte Order Mark)付き、またはBOMなしのUTF-16リトルエンディアン
- BOM付き、またはBOMなしのUTF-16ビッグエンディアン
- BOM付きのUTF-8

また、clコマンドは、プログラム中の文字列などをANSIコンソールコードページに変換してモジュールを生成するため、データベースサーバとの間で送受信されるデータはANSIコンソールコードページになります。したがって、ANSIコンソールコードページに該当する符号化方式をクライアントの符号化方式に設定してください。

クライアントの符号化方式の設定方法は、“PostgreSQL Documentation”の“Server Administration”の“Character Set Support”を参照してください。
（例: 日本語において環境変数を使用する場合、PGCLIENTENCODINGにSJISを設定する）

3.4.2 アプリケーションのコンパイル(Linuxの場合)

アプリケーションのコンパイル時は、以下のオプションを指定します。

表3.1 インクルードファイルとライブラリパス

オプションの種類	オプションの指定方法
インクルードファイルのパス	-I<unixODBC64bitのインクルードファイルの格納ディレクトリ>
ライブラリのパス	-L<unixODBC64bitのライブラリの格納ディレクトリ>

表3.2 ODBCライブラリ

ライブラリの種類	ライブラリ名
動的ライブラリ	libodbc.so

注意

64ビットアプリケーションを作成する場合には“-m64”の指定が必要です。

例

ODBCアプリケーションのコンパイル例を説明します。

```
gcc -m64 -I/usr/local/include(注) -L/usr/local/lib(注) -lodbc testproc.c -o testproc
```

注) unixODBCのインストール先を指定せずにソースからビルドおよびインストールした場合の例です。インストール先を指定した場合には、インストール先のディレクトリを設定してください。

3.4.3 データベース多重化運用時のアプリケーション作成

データベース多重化運用時のアプリケーション作成において考慮する事項を説明します。

参照

- データベース多重化運用についての詳細は、“クラスタ運用ガイド(データベース多重化編)”を参照してください。
- クラスタソフトウェアと連携したフェイルオーバ運用での、アプリケーション作成における考慮事項については、“クラスタ運用ガイド(PRIMECLUSTER編)”の“アプリケーションの作成”を参照してください。

3.4.3.1 アプリケーションの接続先切り替えが発生した場合のエラーと対処

データベース多重化運用時に、アプリケーションの接続先の切り替えが発生した場合、明示的にコネクションを切断し、コネクションの再接続またはアプリケーションを再実行してください。

以下に切り替え発生時のエラーと対処を示します。

状態		エラー情報(注)	対処
サーバダウン または Fujitsu Enterprise Postgresシステムダウン	アクセス中にダウンしたとき	57P01 08S01	切替え完了後、コネクションの再接続、またはアプリケーションを再実行してください。
	ダウン中にアクセスしたとき	08001	
スタンバイサーバへの切替え	アクセス中に切替えたとき	57P01 08S01	
	切替え中にアクセスしたとき	08001	

注: SQLSTATEの返却値となります。

第4章 .NET Data Provider

.NETアプリケーションを作成するための設定方法について説明します。

4.1 開発環境

.NET Data Providerは、以下の環境で動作可能です。

.NETアプリケーションの開発および動作に必要な環境	.NET 8.0 .NET 7.0 .NET 6.0 .NET Standard 2.0 .NET Standard 2.1
.NET環境で動作するアプリケーションの統合開発環境	Visual Studio 2022 Visual Studio 2019 Visual Studio 2017 Visual Studio 2015
利用可能な開発言語	C# Visual Basic .NET

4.2 セットアップ

.NET Data ProviderおよびEntity Framework Coreのセットアップ方法について説明します。

4.2.1 .NET Data Providerのセットアップ

Fujitsu Enterprise Postgresの.NET Data Providerは、ローカルのNuGetパッケージとしてインストール可能です。インストールは、以下の手順で行います。

NuGetパッケージの格納場所

Npgsql NuGetパッケージは、以下に格納されています。“<*x*>”は製品のバージョンを示します。

L
- Linuxの場合

```
<Fujitsu Enterprise Postgresクライアント機能のインストールディレクトリ>/DOTNET/Npgsql.<x>.0.0.nupkg
```

W
- Windows(R)の場合

```
<Fujitsu Enterprise Postgresクライアント機能のインストールディレクトリ>¥DOTNET¥Npgsql.<x>.0.0.nupkg
```

ローカルパッケージソースの追加

NuGetローカルパッケージソースがない場合は追加します。

L
- Linuxの場合

上記のNugetパッケージ格納場所をローカルパッケージソースとして追加します。

```
dotnet nuget add source <Fujitsu Enterprise Postgresクライアント機能のインストールディレクトリ>/DOTNET
-n <ソース名>
```

W
- Windows(R)の場合

1. Visual Studioを起動し、[ツール] - [オプション] - [NuGet パッケージ マネージャー]をクリックし、[パッケージソース]を選択します。
2. 右上隅の[+]ボタンをクリックし、[名前]フィールドに“ローカル パッケージ ソース”を入力します。

3. [...]ボタンをクリックし上記のフォルダを検索します。このフォルダを選択し、[OK]ボタンをクリックします。

NuGetパッケージのインストール

ローカルパッケージソースからNuGetパッケージをインストールします。

L ― Linuxの場合

プロジェクトファイルにパッケージ参照を追加します。

```
dotnet add <プロジェクトファイルのパス> package Npgsql
```

W ― Windows(R)の場合

1. Visual Studioを起動し、[ツール] - [NuGet パッケージ マネージャー] - [ソリューションの NuGet パッケージの管理]をクリックします。
2. 右上隅にある[パッケージ ソース]から“ローカルパッケージソース”を選択します。
3. ローカルパッケージソースを設定すると、このローカルロケーションにある利用可能なすべてのNuGetパッケージが表示されます。“Npgsql”を選択し、このパッケージをインストールするプロジェクトを選択します。
4. [インストール]をクリックします。

4.2.2 .NET Data Provider タイププラグインのセットアップ

Fujitsu Enterprise Postgresの.NET Data Providerには、より多くのデータ型マッピングをサポートする4つのタイププラグインが同梱されています(例えば、Npgsql.NodaTimeによる日付時刻サポート)。タイププラグインは、NpgsqlがPostgreSQLの値をCLRの型にマッピングする方法を変更します。

タイププラグインはローカルのNuGetパッケージとしてインストール可能です。以下の4つのパッケージをインストールできます。“<*x*>”は製品のバージョンを示します。

- Npgsql.NodaTime.<*x*>.0.0.nupkg
- Npgsql.Json.NET.<*x*>.0.0.nupkg
- Npgsql.NetTopologySuite.<*x*>.0.0.nupkg:空間型(注)
- Npgsql.GeoJSON.<*x*>.0.0.nupkg:空間型(注)

注) 空間型プラグインでは、サーバにPostGIS拡張がインストールされている必要があります。

また、各プラグインについては、“タイププラグインについて”および“各タイププラグインに関する注意事項”を参照してください。

タイププラグインのインストールは、以下の手順で行います。

NuGetパッケージの格納場所

タイププラグインのNuGetパッケージは、以下に格納されています。“<*x*>”は製品のバージョンを示します。

L ― Linuxの場合

```
<Fujitsu Enterprise Postgresクライアント機能のインストールディレクトリ>/DOTNET/Npgsql.*.<x>.0.0.nupkg
```

W ― Windows(R)の場合

```
<Fujitsu Enterprise Postgresクライアント機能のインストールディレクトリ>¥DOTNET¥Npgsql.*.<x>.0.0.nupkg
```

ローカルパッケージソースの追加

NuGetローカルパッケージソースが存在しない場合は、追加します。

L ― Linuxの場合

上記のNugetパッケージ格納場所をローカルパッケージソースとして追加します。

```
dotnet nuget add source <Fujitsu Enterprise Postgresクライアント機能のインストールディレクトリ>/DOTNET
-n <ソース名>
```

W — Windows(R)の場合

1. Visual Studioを起動し、[ツール] - [オプション] - [NuGet パッケージ マネージャー]をクリックし、[パッケージ ソース]を選択します。
2. 右上隅の[+]ボタンをクリックし、[名前]フィールドに“ローカル パッケージ ソース”を入力します。
3. [...]ボタンをクリックし、上のフォルダを検索します。このフォルダを選択し、[OK]ボタンをクリックします。

NuGetパッケージのインストール

ローカルのパッケージソースからNuGetパッケージをインストールします。

L — Linuxの場合

プロジェクトファイルにパッケージ参照を追加します(例:Npgsql.NodaTime)。

```
dotnet add <プロジェクトファイルのパス> package Npgsql.NodaTime
```

W — Windows(R)の場合

1. Visual Studioを起動し、[ツール] - [NuGet パッケージ マネージャー] - [ソリューションの NuGet パッケージの管理]をクリックします。
2. 右上隅にある[パッケージ ソース]から“ローカルパッケージソース”を選択します。
3. ローカルパッケージソースを設定すると、このローカルロケーションにある利用可能なすべてのNuGetパッケージが表示されます。インストールするプラグイン(例:Npgsql.NodaTime)を選択し、このパッケージをインストールするプロジェクトを選択します。
4. [インストール]をクリックします。

4.2.3 Entity Framework Coreのセットアップ

Entity Framework Coreは、NuGetパッケージファイルとして提供されています。ローカルにインストールするには、以下の手順で行います。

NuGetパッケージの格納場所

EntityFramework Core NuGetパッケージは、以下に格納されています。

L — Linuxの場合

```
<Fujitsu Enterprise Postgresクライアント機能のインストールディレクトリ>/DOTNET/
Npgsql.EntityFrameworkCore.PostgreSQL.8.0.2.nupkg
```

W — Windows(R)の場合

```
<Fujitsu Enterprise Postgresクライアント機能のインストールディレクトリ>¥DOTNET
¥Npgsql.EntityFrameworkCore.PostgreSQL.8.0.2.nupkg
```

ローカルパッケージソースの追加

NuGetローカルパッケージソースがない場合は追加します。

L — Linuxの場合

上記のNugetパッケージ格納場所をローカルパッケージソースとして追加します。

```
dotnet nuget add source <Fujitsu Enterprise Postgresクライアント機能のインストールディレクトリ>/DOTNET
-n <ソース名>
```

W

－ Windows(R)の場合

1. Visual Studioを起動し、[ツール] - [オプション] - [NuGet パッケージ マネージャー]をクリックし、[パッケージ ソース]を選択します。
2. 右上隅の[+]ボタンをクリックし、[名前]フィールドに“ローカル パッケージ ソース”を入力します。
3. [...]ボタンをクリックし上記のフォルダを検索します。このフォルダを選択し、[OK]ボタンをクリックします。

NuGetパッケージのインストール

ローカルパッケージソースからNuGetパッケージをインストールします。

L

－ Linuxの場合

プロジェクトファイルにパッケージ参照を追加します。

```
dotnet add <プロジェクトファイルのパス> package Npgsql.EntityFrameworkCore.PostgreSQL
```

W

－ Windows(R)の場合

1. Visual Studioを起動し、[ツール] - [NuGet パッケージ マネージャー] - [ソリューションの NuGet パッケージの管理]をクリックします。
2. 右上隅にある[パッケージ ソース]から“ローカルパッケージソース”を選択します。
3. ローカルパッケージソースを設定すると、このローカルロケーションにある利用可能なすべてのNuGetパッケージが表示されます。“Npgsql.EntityFrameworkCore.PostgreSQL”を選択し、このパッケージをインストールするプロジェクトを選択します。
4. [インストール]をクリックします。

4.2.4 Npgsql.OpenTelemetryのセットアップ

Fujitsu Enterprise Postgresの.NET Data Providerには、オブザーバビリティフレームワークであるOpenTelemetryによるトレーシングをサポートするプラグインが同梱されています。これにより、Npgsqlにトレースデータを発行させることが可能になります。

OpenTelemetryプラグインはローカルのNuGetパッケージとして提供されています。ローカルにインストールするには、以下の手順で行います。

NuGetパッケージの格納場所

OpenTelemetryのNuGetパッケージは、以下に格納されています。“<*x*>”は製品のバージョンを示します。

L

－ Linuxの場合

```
<Fujitsu Enterprise Postgresクライアント機能のインストールディレクトリ>/DOTNET/
Npgsql.OpenTelemetry.<x>.0.0.nupkg
```

W

－ Windows(R)の場合

```
<Fujitsu Enterprise Postgresクライアント機能のインストールディレクトリ>¥DOTNET
¥Npgsql.OpenTelemetry.<x>.0.0.nupkg
```

ローカルパッケージソースの追加

NuGetローカルパッケージソースが存在しない場合は、追加します。

L

－ Linuxの場合

上記のNugetパッケージ格納場所をローカルパッケージソースとして追加します。

```
dotnet nuget add source <Fujitsu Enterprise Postgresクライアント機能のインストールディレクトリ>/DOTNET
-n <ソース名>
```

W

－ Windows(R)の場合

1. Visual Studioを起動し、[ツール] - [オプション] - [NuGet パッケージ マネージャー]をクリックし、[パッケージ ソース]を選択します。
2. 右上隅の[+]ボタンをクリックし、[名前]フィールドに“ローカル パッケージ ソース”を入力します。
3. [...]ボタンをクリックし、上のフォルダを検索します。このフォルダを選択し、[OK]ボタンをクリックします。

NuGetパッケージのインストール

ローカルのパッケージソースからNuGetパッケージをインストールします。

L

－ Linuxの場合

プロジェクトファイルにパッケージ参照を追加します。

```
dotnet add <プロジェクトファイルのパス> package Npgsql.OpenTelemetry
```

W

－ Windows(R)の場合

1. Visual Studioを起動し、[ツール] - [NuGet パッケージ マネージャー] - [ソリューションの NuGet パッケージの管理]をクリックします。
2. 右上隅にある[パッケージ ソース]から“ローカルパッケージソース”を選択します。
3. ローカルパッケージソースを設定すると、このローカルロケーションにある利用可能なすべてのNuGetパッケージが表示されます。“Npgsql.OpenTelemetry”を選択し、このパッケージをインストールするプロジェクトを選択します。
4. [インストール]をクリックします。

4.3 データベースへの接続

データベースへの接続方法について説明します。

- NpgsqlConnectionを使用する場合
- NpgsqlConnectionStringBuilderを使用する場合

4.3.1 NpgsqlConnectionを使用する場合

接続文字列を指定してデータベースに接続します。

例

アプリケーションの記述例

```
using Npgsql;

NpgsqlConnection conn = new NpgsqlConnection("Server=sv1;Port=27500;Database=mydb;
Username=myuser;Password=myuser01; Timeout=20;CommandTimeout=20;");
```

データベースへの接続文字列については、“4.3.3 接続文字列”を参照してください。

4.3.2 NpgsqlConnectionStringBuilderを使用する場合

接続情報をNpgsqlConnectionStringBuilderオブジェクトのプロパティに指定して接続文字列を生成します。

例

アプリケーションの記述例

```
using Npgsql;

NpgsqlConnectionStringBuilder sb = new NpgsqlConnectionStringBuilder();
sb.Host = "sv1";
sb.Port = 27500;
sb.Database = "mydb";
sb.Username = "myuser";
sb.Password = "myuser01";
sb.Timeout = 20;
sb.CommandTimeout = 20;
NpgsqlConnection conn = new NpgsqlConnection(sb.ConnectionString);
```

データベースへの接続文字列については、“4.3.3 接続文字列”を参照してください。

4.3.3 接続文字列

コネクションには、接続するデータベースへの接続先情報として以下を指定します。

```
Server=127.0.0.1;Port=27500;Database=mydb;Username=myuser;Password=myuser01;...;
       (1)             (2)             (3)           (4)              (5)       (6)
```

(1) 接続するサーバのホスト名またはIPアドレスを指定します。必ず指定してください。

(2) データベースサーバのポート番号を指定します。省略した場合は、既定値の27500となります。

(3) 接続するデータベース名を指定します。

(4) データベースに接続するユーザー名を指定します。

(5) データベースに接続するユーザー名のパスワードを指定します。

(6) その他の接続情報の指定方法については、以下の表を参照してください。

.NET Data Provider(Npgsql)で接続文字列に指定可能なキーワードを以下に示します。

なお、Oracleデータベース互換機能を利用する場合は、注意が必要な設定があります。“9.2.3 アプリケーション開発用のインタフェースとの連携時の注意事項”を参照してください。

基本接続

キーワード	説明	既定値
Host	接続するサーバのホスト名を指定します。複数のホストを指定することもできます。ホスト名は必ず指定してください。	
Port	サーバのTCPポート番号を指定します。	27500
Database	接続するデータベース名を指定します。	Usernameと同じ
Username	データベースに接続するユーザー名を指定します。統合セキュリティを使用する場合は必要ありません。	PGUSER
Password	データベースに接続するユーザー名のパスワードを指定します。統合セキュリティを使用する場合は必要ありません。	PGPASSWORD
Passfile	パスワードを取得する、パスワードファイル(PGPASSFILE)のパスを指定します。	PGPASSFILE

セキュリティと暗号化

キーワード	説明	既定値
SSL Mode	サーバのサポートに応じて、SSLを使用するかどうかを、以下の値から1つ指定します。 Disable：SSL接続は行われません。 Allow：サーバが要求する場合のみSSL接続を行います。 Prefer：可能であればSSL接続を行います。 Require：SSL接続を行います。このとき接続サーバの信頼性は検証しません。 VerifyCA：SSL接続を行います。このとき接続サーバの信頼性を検証します。 VerifyFull: SSL接続を行います。このとき接続サーバの信頼性を検証し、それが指定したサーバであることを確認します。	Prefer
SSL Certificate	サーバに送信されるクライアント証明書の場所を指定します。	PGSSLCERT
SSL Key	サーバに送信されるクライアント証明書のクライアントキーの場所を指定します。	PGSSLKEY
SSL Password	クライアント証明書のキーのパスワードを指定します。	
Root Certificate	サーバ証明書の検証に使用されるCA証明書の場所を指定します。	PGSSLROOTCERT
Check Certificate Revocation	認証時に証明書失効リストを確認するかどうかを指定します。	false
Integrated Security	統合セキュリティ(GSS/SSPI)を使用してログインするかどうかを指定します。	false
Persist Security Info	接続中または接続状態になったことがある場合に、パスワードなどのセキュリティ上重要な情報が接続の一部として返されないかどうかを示すブール値を取得または設定します。	false
Kerberos Service Name	認証に使用されるKerberosサービス名を指定します。	postgres
Include Realm	認証に使用されるKerberosレルムを指定します。	
Include Error Detail	有効にすると、データベースのエラーと通知の詳細がPostgresException.DetailとPostgresNotice.Detailに含まれます。これらには機密情報を含めることができます。	false
Log Parameters	有効にすると、コマンドの実行時にパラメータ値がログに記録されます。	false

プーリング

キーワード	説明	既定値
Pooling	接続プーリングを使用するかどうかを指定します。	true
Minimum Pool Size	接続プールの最小サイズです。	0
Maximum Pool Size	接続プールの最大サイズです。	100
Connection Idle Lifetime	すべての接続の数がMinimum Pool Sizeを超えた場合に、プール内 のアイドル状態の接続を切断するまでの待機時間(秒)を指定します。	300

キーワード	説明	既定値
Connection Pruning Interval	存続期間を超えたアイドル接続のプルーニングを試行するまでに、プールが待機する秒数を指定します。Connection Idle Lifetimeを参照してください。	10
ConnectionLifetime	接続の合計最大寿命を秒数で指定します。この値を超えた接続は、プールから返されるのではなく破棄されます。これは、実行中のサーバとオンラインになったばかりのサーバとの間でロードバランシングを強制的に実行するクラスタ構成で役立ちます。	0 (disabled)

タイムアウトとキープアライブ

キーワード	説明	既定値
Timeout	接続の確立中に、試行を終了してエラーを生成するまで待機する時間(秒)を指定します。	15
Command Timeout	コマンドの実行中に、試行を終了してエラーを生成するまでの待機時間(秒)を指定します。無限大の場合は0に設定します。	30
Internal Command Timeout	内部コマンドを実行しようとしたときに、試行を終了してエラーを生成するまでの待機時間(秒)を指定します 。-1はCommandTimeoutを使用し、0はタイムアウトなしを意味します。	-1
Cancellation Timeout	タイムアウトまたはキャンセルされたクエリのキャンセル要求に対する応答を読み取ろうとしている間に、試行を終了してエラーを生成するまでの待機時間(ミリ秒)を指定します。-1は待機をスキップし、0は無限待機を意味します。	2000
Keepalive	接続の非アクティブ状態がここで指定した秒数を超えると、Npgsql が キープアライブクエリを送信します。	0 (disabled)
Tcp Keepalive	オーバーライドが指定されていない場合に、システムデフォルトでTCPキープアライブを使用するかどうかを指定します。	false
Tcp Keepalive Time	接続の非アクティブ時間がここで指定したミリ秒数を超えると、TCPキープアライブクエリが送信されます。このオプションの使用は推奨されません。Keepaliveの使用を推奨します。これはWindowsのみでサポートされています。	0 (disabled)
Tcp Keepalive Interval	確認応答が受信されない場合に連続するキープアライブパケットが送信される間隔(ミリ秒)を指定します。Tcp KeepAlive Timeも0以外にする必要があります。これはWindowsのみでサポートされています。	Tcp Keepalive Timeの値

パフォーマンス

キーワード	説明	既定値
Max Auto Prepare	任意の時点で自動的に準備できるSQL文の最大数です。この値を超えると、最後に使用された文が再利用されます。0を指定すると、自動準備が無効になります。	0
Auto Prepare Min Usages	SQL文の使用回数がここで指定した値を超えると、SQL文が自動的に 準備されます。	5
Read Buffer Size	Npgsqlが読み込み時に使用する内部バッファのサイズです。データベースから大量のデータを転送する場合に、この値を大きくするとパフォーマンスが向上することがあります。	8192
Write Buffer Size	Npgsqlが書き込み時に使用する内部バッファのサイズを決定します。大量のデータをデータベースに転送する場合に、この値を大きくするとパフォーマンスが向上することがあります。	8192

キーワード	説明	既定値
Socket Receive Buffer Size	ソケット受信バッファのサイズです。	システム依存
Socket Send Buffer Size	ソケット送信バッファのサイズです。	システム依存
No Reset On Close	trueのとき、接続がプールに戻されるときに接続状態をリセットしません。場合によっては、これによりリーク状態となる代わりにパフォーマンスが向上します。ベンチマークでパフォーマンスが向上した場合にのみ使用してください。	false

フェイルオーバと負荷分散

キーワード	説明	既定値
Target Session Attributes	優先して接続するサーバタイプを指定します。 any：任意の正常な接続を許容します。 primary：プライマリサーバに接続します。 stanby：スタンバイサーバに接続します。 prefer-primary：まずプライマリサーバに接続を試行しますが、リストされたホストにプライマリサーバがない場合はanyモードで再試行します。 prefer-standby：まずスタンバイサーバに接続を試行しますが、リストされたホストにスタンバイサーバがない場合はanyモードで再試行します。 read-write：デフォルトのトランザクションモードが読み取り/書き込みのサーバに接続します。 read-only：デフォルトのトランザクションモードが読み取り専用のサーバに接続します。	PGTARGETSESSIONATTRS, Any
Load Balance Hosts	ラウンドロビン方式による複数ホスト間の負荷分散を行うかどうかを指定します。	false
Host Recheck Seconds	ホストのキャッシュされた状態が有効と見なされる期間を制御します。	10
Enable Fdw Acs	接続ルーティング機能を使用するとき、データノードの接続先のタイプを切り替えるかどうかをonまたはoffで指定します。"Target Session Attributes=prefer-primary"が指定されているとき、これをonに指定することはできません。	off

その他

キーワード	説明	既定値
Options	有効なデータベース接続オプションを指定します。 (例:Options=-c synchronous_commit=local)	PGOPTIONS
Application Name	接続開始時にバックエンドに送信されるアプリケーション名を指定します。	
Enlist	コネクションを、トランザクションスコープで宣言したトランザクションに参加させるかどうかを指定します。	true
Search Path	スキーマ検索パスを設定します。	なし
Client Encoding	client_encodingパラメータを取得または設定します。	PGCLIENTENCODING

キーワード	説明	既定値
Encoding	データベースの文字列データのエンコード/デコードに使用する.NETエンコーディングを取得または設定します。	UTF8
Timezone	セッションのタイムゾーンを取得または設定します。	PGTZ
EF Template Database	Entity Frameworkでデータベースを作成するときに指定するデータベーステンプレートです。	template1
EF Admin Database	Entity Frameworkでデータベースを作成および削除するときに指定する管理データベースです。	template1
Load Table Composites	独立した複合型だけでなく、表の複合型定義をロードするかどうかを指定します。	false
Array Nullability Mode	オブジェクトインスタンスとして要求されたときに値型の配列を返す方法を設定します。使用できる値は以下です。 Never：常にnull入力不可の配列として返します。 Always：常にnull入力可能な配列として返します。 PerInstance：返される配列の方は実行時に決定されます。	Never

4.4 アプリケーション開発

4.4.1 データ型

Fujitsu Enterprise Postgresでは、様々なデータ型を使用することができます。

サポートするデータ型一覧

データ型
boolean
smallint
integer
bigint
real
double precision
numeric
money
text
character varying
character
national character varying
national character
citext
json
jsonb
xml
uuid
bytea

データ型
timestamp without time zone
timestamp with time zone
date
time without time zone(注1)
time with time zone(注1)
interval(注2)
cidr
inet
macaddr
tsquery
tsvector
bit(1)
bit(n)
bit varying
point
lseg
path
polygon
line
circle
box
hstore
oid
xid
cid
oidvector
name
(internal) char
geometry (PostGIS)
record
composite types
range types
multirange types (PG14)
enum types
array types

注1) 以下の例のとおり、time with time zoneおよびtime without time zoneの値を表示すると、日付データが補填された形で表示されます。しかし、実データは時刻のみのデータで構成されるため、表示以外に注意すべきことはありません。

例:

【表(t1)の構成】

col1(time with time zone)	col2(time without time zone)
1/01/0001 10:21:30 +08:00	10:21:30
1/01/0001 23:34:03 +08:00	23:34:03
1/01/0001 17:23:54 +08:00	17:23:54

time with time zoneは日付データに固定値“2/01/0001”を表示し、time without time zoneは日付データのみを表示します。

```
SELECT *
  FROM t1;
 col1                       |    col2
----------------------------+--------------------
 1/01/0001 10:21:30 +08:00 | 10:21:30
 1/01/0001 23:34:03 +08:00 | 23:34:03
 1/01/0001 17:23:54 +08:00 | 17:23:54
```

注2) 形式はd.hh:mm:ssで、dは整数、hh:mm:ssの最大値は23.59.59(23時59分59秒)です。

4.4.2 アプリケーションのデータ型とデータベースのデータ型の関係

SQLのデータ型に対して、使用できるデータタイプは以下のとおりです。

読み取りマッピング

PostgreSQLの データ型	標準.NET データ型	非標準.NET データ型
boolean	bool	
smallint	short	byte, sbyte, int, long, float, double, decimal
integer	int	byte, short, long, float, double, decimal
bigint	long	long, byte, short, int, float, double, decimal
real	float	double
double precision	double	
numeric	decimal	byte, short, int, long, float, double, BigInteger
money	decimal	
text	string	char[]
character varying	string	char[]
character	string	char[]
national character varying	string	char[]
national character	string	char[]
citext	string	char[]
json	string	char[]
jsonb	string	char[]
xml	string	char[]
uuid	Guid	
bytea	byte[]	
timestamp without time zone	DateTime (Unspecified)	

PostgreSQLの データ型	標準.NET データ型	非標準.NET データ型
timestamp with time zone	DateTime (Utc)(注1)	DateTimeOffset (Offset=0)(注2)
date	DateTime	DateOnly
time without time zone	TimeSpan	TimeOnly
time with time zone	DateTimeOffset	
interval	TimeSpan(注3)	NpgsqlInterval
cidr	NpgsqlCidr	
inet	IPAddress	NpgsqlInet
macaddr	PhysicalAddress	
tsquery	NpgsqlTsQuery	
tsvector	NpgsqlTsVector	
bit(1)	bool	BitArray
bit(n)	BitArray	
bit varying	BitArray	
point	NpgsqlPoint	
lseg	NpgsqlLSeg	
path	NpgsqlPath	
polygon	NpgsqlPolygon	
line	NpgsqlLine	
circle	NpgsqlCircle	
box	NpgsqlBox	
hstore	Dictionary<string, string>	
oid	uint	
xid	uint	
cid	uint	
oidvector	uint[]	
name	string	char[]
(internal) char	char	byte, short, int, long
geometry (PostGIS)	PostgisGeometry	
record	object[]	
composite types	T	
range types	NpgsqlRange<TElement>	
multirange types (PG14)	NpgsqlRange<TElement>[]	
enum types	TEnum	
array types	Array (of element type)	

注1) Npgsql.EnableLegacyTimestampBehaviorが有効な場合、timestamp with time zoneを読み込むとUtcではなくLocal DateTimeが返されます。

注2) Npgsql.EnableLegacyTimestampBehaviorが有効な場合、timestamp with time zoneをDateTimeOffsetとして読み込むと、Npgsqlが実行されているサーバーのタイムゾーンに基づいたローカルオフセットが返されます。

注3) PostgreSQLのintervalのうち、月または年の要素を持つものはTimeSpanとして読み込むことができません。NodaTimeのPeriodタイプ、またはNpgsqlIntervalの使用を検討してください。

書き込みマッピング

PostgreSQLのデータ型	標準.NETデータ型	非標準.NETデータ型	NpgsqlDbType	DbType
boolean	bool		Boolean	Boolean
smallint	short, byte, sbyte		Smallint	Int16
integer	int		Integer	Int32
bigint	long		Bigint	Int64
real	float		Real	Single
double precision	double		Double	Double
numeric	decimal, BigInteger		Numeric	Decimal, VarNumeric
money		decimal	Money	Currency
text	string, char[], char		Text	String, StringFixedLength, AnsiString, AnsiStringFixedLength
character varying		string, char[], char	Varchar	
character		string, char[], char	Char	
citext		string, char[], char	Citext	
json		string, char[], char	Json	
jsonb		string, char[], char	Jsonb	
xml		string, char[], char	Xml	
uuid	Guid		Uuid	
bytea	byte[]	ArraySegment<byte>	Bytea	Binary
timestamp without time zone	DateTime (Utc)(注1), DateTimeOffset		TimestampTz	DateTime(注2), DateTimeOffset
timestamp with time zone	DateTime (Local/ Unspecified)(注1)		Timestamp	DateTime2
date	DateOnly	DateTime	Date	Date
time without time zone	TimeOnly	TimeSpan	Time	Time
time with time zone		DateTimeOffset	TimeTz	
interval	TimeSpan	NpgsqlInterval	Interval	
cidr		ValueTuple<IPAddress, int>, IPAddress	Cidr	

PostgreSQLのデータ型	標準.NETデータ型	非標準.NETデータ型	NpgsqlDbType	DbType
inet	IPAddress	ValueTuple<IPAddress, int>	Inet	
macaddr	PhysicalAddress		MacAddr	
tsquery	NpgsqlTsQuery		TsQuery	
tsvector	NpgsqlTsVector		TsVector	
bit		bool, BitArray, string	Bit	
bit varying	BitArray	bool, BitArray, string	Varbit	
point	NpgsqlPoint		Point	
lseg	NpgsqlLSeg		LSeg	
path	NpgsqlPath		Path	
polygon	NpgsqlPolygon		Polygon	
line	NpgsqlLine		Line	
circle	NpgsqlCircle		Circle	
box	NpgsqlBox		Box	
hstore	IDictionary<string, string>		Hstore	
oid		uint	Oid	
xid		uint	Xid	
cid		uint	Cid	
oidvector		uint[]	Oidvector	
name		string, char[], char	Name	
(internal) char		byte	InternalChar	
composite types	Pre-mapped type		Composite	
range types	NpgsqlRange<TSubtype>		Range | NpgsqlDbType	
enum types	Pre-mapped type		Enum	
array types	T[], List<T>		Array | NpgsqlDbType	

注1)UTC DateTimeはtimestamp with time zoneとして書き込まれ、Local/Unspecified DateTimesはtimestamp without time zoneとして書き込まれます。Npgsql.EnableLegacyTimestampBehaviorが有効な場合、DateTimeは常にtimestamp without time zoneとして書き込まれます。

注2) Npgsql.EnableLegacyTimestampBehaviorが有効な場合、DbType.DateTimeはtimestamp without time zoneにマップされます。

4.4.3 データベース多重化運用時のアプリケーション作成

データベース多重化運用時のアプリケーション作成において考慮する事項を説明します。

参照

- データベース多重化運用についての詳細は、“クラスタ運用ガイド(データベース多重化編)”を参照してください。

・ クラスタソフトウェアと連携したフェイルオーバ運用での、アプリケーション作成における考慮事項については、“クラスタ運用ガイド(PRIMECLUSTER編)”の“アプリケーションの作成”を参照してください。

4.4.3.1 アプリケーションの接続先切り替えが発生した場合のエラーと対処

データベース多重化運用時に、アプリケーションの接続先の切り替えが発生した場合、明示的にコネクションを切断し、コネクションの再接続またはアプリケーションを再実行してください。

以下に切り替え発生時のエラーと対処を示します。

状態		エラー情報	対処
サーバダウン または Fujitsu Enterprise Postgresシステムダウン	アクセス中にダウンしたとき	57P01(注) 空文字列(注)	切替え完了後、コネクションの再接続、またはアプリケーションを再実行してください。
	ダウン中にアクセスしたとき	空文字列(注)	
スタンバイサーバへの切替え	アクセス中に切替えたとき	57P01(注) 空文字列(注)	
	切替え中にアクセスしたとき	空文字列(注)	

注: PostgresExceptionの属性SqlStateの返却値となります。

4.4.4 注意事項

タイププラグインについて

・ 以下のタイププラグインをサポートしています。

－ NodaTime

PostgrSQLの日付/時刻データ型をNodatimeとして扱うためのプラグイン。

－ Json.NET

PostgreSQLのJSONデータ型（json型およびjsonb型）を読み書きするときにNpgsqlがNewtonSoft JsonNETライブラリを使用できるようします。これにより、JSONデータ型のまま扱うことができます。

－ NetTopologySuite

PostGISのデータ型をNetTopologySuiteとして扱うためのプラグイン。NpgsqlがPostGIS空間タイプをNetTopologySuite(.NETの主要な空間ライブラリ)に直接マッピングできます。

－ GeoJSON

PostGISのデータ型をGeoJSON.NETとして扱うためのプラグイン。NpgsqlがGeoJSONライブラリを介してPostGIS空間タイプをGeoJSONとして読み書きできます。

・ アプリケーションでプラグインをセットアップするには、プラグインに依存関係を追加します(プロジェクトにインストールされたときに自動的に行われます)。以下のコードは、Npgsql.NodaTimeプラグインの例です。

```
using Npgsql;
// Place this at the beginning of your program to use NodaTime everywhere (recommended)
NpgsqlConnection.GlobalTypeMapper.UseNodaTime();
// Or to temporarily use NodaTime on a single connection only:
conn.TypeMapper.UseNodaTime();
```

プラグインがセットアップされると、以下のようにNodaTimeオブジェクトを読み書きすることができます。

```
// Write NodaTime Instant to PostgreSQL "timestamp without time zone"
using (var cmd = new NpgsqlCommand(@"INSERT INTO mytable (my_timestamp) VALUES (@p)", conn))
{
    cmd.Parameters.Add(new NpgsqlParameter("p", Instant.FromUtc(2011, 1, 1, 10, 30)));
    cmd.ExecuteNonQuery();
}

// Read timestamp back from the database as an Instant
using (var cmd = new NpgsqlCommand(@"SELECT my_timestamp FROM mytable", conn))
using (var reader = cmd.ExecuteReader())
{
    reader.Read();
    var instant = reader.GetFieldValue<Instant>(0);
}
```

- タイププラグインの修正を適用するためには、下記のいずれかを実施します。
  - タイププラグインのアンインストール (“4.5.2 .NET Data Provider タイププラグインのアンインストール”を参照) を実施後、タイププラグインをセットアップ (“4.2.2 .NET Data Provider タイププラグインのセットアップ”を参照) する。
  - ソリューションのpackagesディレクトリからタイププラグインのディレクトリを削除したのち、nuget restoreコマンドにより復元する。
- タイププラグインを使用したアプリケーションを配布する際、配布物の中にタイププラグインが含まれます。そのため、タイププラグインに関する修正を適用したのち、アプリケーションを再度ビルドし、更新されたアプリケーションを再配布する必要があります。

各タイププラグインに関する注意事項

各タイププラグインに関する注意事項を説明します。

NodaTime

PostgreSQLのデータ型とNodatimeのデータ型のマッピングを説明します。

マッピングテーブル

PostgreSQL の データ型	既定のNodaTime 型	追加されるNodaTime 型	備考
timestamp with time zone	Instant	ZonedDateTime(注1), OffsetDateTime(注1)	データベース内のUTCタイムスタンプです。UTC ZonedDateTimeおよびOffsetDateTimeのみがサポートされます。
timestamp without time zone	LocalDateTime(注2)		未知または暗黙のタイムゾーンのタイムスタンプです。
date	LocalDate		タイムゾーンやオフセット情報を持たない単純な日付です。
time without time zone	LocalTime		タイムゾーンやオフセット情報を持たない単純な時刻です。
time with time zone	OffsetTime		時刻とオフセットを格納する型です。一般的に使用は推奨されません。
interval	Period	Duration	秒未満の単位から年までの時間間隔です。[NodaTime Duration] は、日とそれより短い間隔でサポートされていま

PostgreSQL の データ型	既定のNodaTime 型	追加されるNodaTime 型	備考
			すが、年や月ではサポートされていません (これらには絶対期間がないため)。期間は、任意の間隔単位で使用できます。
tszrange	Interval	NpgsqlRange<Instant> など	2つの時間インスタンス(開始と終了)間の時間間隔です。
strange	NpgsqlRange<LocalDateTime>		未知または暗黙のタイムゾーンの2つのタイムスタンプ間の時間間隔です。
daterange	DateInterval	NpgsqlRange<LocalDate> など	2つの日付の間隔です。

注1) Npgsql.EnableLegacyTimestampBehaviorが有効な場合、ZonedDateTimeまたはOffsetDateTimeの書き込みまたは読み取りは自動的にUTCに変換されます。

注2) Npgsql.EnableLegacyTimestampBehaviorが有効な場合 、timestamp without time zoneはデフォルトでLocalDateTimeではなくInstantにマップされます。

Json.NET

JSONプラグインがセットアップされると、ユーザーは透過的にCLRオブジェクトをJSON値として読み書きでき、プラグインはそれらを自動的にシリアル化/逆シリアル化します。

以下にコード例を示します。

```
// Write arbitrary CLR types as JSON
using (var cmd = new NpgsqlCommand(@"INSERT INTO mytable (my_json_column) VALUES (@p)", conn))
{
    cmd.Parameters.Add(new NpgsqlParameter("p", NpgsqlDbType.Jsonb) { Value = MyClrType });
    cmd.ExecuteNonQuery();
}

// Read arbitrary CLR types as JSON
using (var cmd = new NpgsqlCommand(@"SELECT my_json_column FROM mytable", conn))
using (var reader = cmd.ExecuteReader())
{
    reader.Read();
    var someValue = reader.GetFieldValue<MyClrType>(0);
}
```

NetTopologySuite (空間型)

デフォルトでは、プラグインはGeometryServiceProvider.InstanceのDefaultCoordinateSequenceFactoryによって提供される縦座標のみを処理します。GeometryServiceProviderが自動的に初期化されると、X座標とY座標が処理されます。動作を変更するには、次の例のようにhandleOrdinatesパラメーターを指定します。

```
conn.TypeMapper.UseNetTopologySuite(handleOrdinates: Ordinates.XYZ);
```

M座標を処理するには、GeometryServiceProvider.InstanceをDotSpatialAffineCoordinateSequenceFactory に設定されたcoordinateSequenceFactoryを使用して、新しいNtsGeometryServicesインスタンスに初期化する必要があります。

```
// Place this at the beginning of your program to use the specified settings everywhere (recommended)
GeometryServiceProvider.Instance = new NtsGeometryServices(
    new DotSpatialAffineCoordinateSequenceFactory(Ordinates.XYM),
    new PrecisionModel(PrecisionModels.Floating),
    -1);
```

```
// Or specify settings for Npgsql only
conn.TypeMapper.UseNetTopologySuite(
    new DotSpatialAffineCoordinateSequenceFactory(Ordinates.XYM));
```

ジオメトリ値の読み書き

データベースからPostGIS値を読み込むと、Npgsqlは適切なNetTopologySuite型(Point、LineStringなど)を自動的に返します。Npgsqlは、パラメーター内のNetTopologySuite型を自動的に認識し、対応するPostGIS型タイプをデータベースに自動的に送信します。以下のコードは、データベースとNetTopologySuite型のやりとりを示しています。

```
var point = new Point(new Coordinate(1d, 1d));
conn.ExecuteNonQuery("CREATE TEMP TABLE data (geom GEOMETRY)");
using (var cmd = new NpgsqlCommand("INSERT INTO data (geom) VALUES (@p)", conn))
{
    cmd.Parameters.AddWithValue("@p", point);
    cmd.ExecuteNonQuery();
}

using (var cmd = new NpgsqlCommand("SELECT geom FROM data", conn))
using (var reader = cmd.ExecuteReader())
{
    reader.Read();
    Assert.That(reader[0], Is.EqualTo(point));
}
```

NpgsqlDbType.Geometryを設定して、パラメータの型を明示的に指定することもできます。

ジオグラフィ(測地座標)サポート

PostGISには、ジオメトリ(デカルト座標の場合)とジオグラフィ(測地座標または球面座標の場合)の2つのタイプがあります。ジオメトリとジオグラフィの違いについては、PostGISドキュメントを参照してください。ジオグラフィは長距離の計算を行う場合にはるかに正確ですが、計算コストが高くなり、ジオメトリでサポートされる空間操作の小さなサブセットのみをサポートします。

Npgsqlは同じNetTopologySuiteタイプを使用してジオメトリとジオグラフィの両方を表現します。Pointタイプはデカルト空間またはジオグラフィ空間のいずれかのポイントを表します。通常、PostgreSQLは必要に応じて型をキャストするため、この区別について考慮は不要です。ただし、Npgsqlはデフォルトでデカルトジオメトリを送信することに注意してください。NpgsqlDbType.Geographyを指定することにより、代わりにジオグラフィを送信するようにNpgsqlに指示するオプションがあります。

```
using (var cmd = new NpgsqlCommand("INSERT INTO data (geog) VALUES (@p)", conn))
{
    cmd.Parameters.AddWithValue("@p", NpgsqlDbType.Geography, point);
    cmd.ExecuteNonQuery();
}
```

デフォルトでジオグラフィを使用したい場合は、プラグインをセットアップするときにそれを指定することもできます。

```
NpgsqlConnection.GlobalTypeMapper.UseNetTopologySuite(geographyAsDefault: true);
```

GeoJSON (空間型)

GeoJSONプラグインの使用方法は、NetTopologuSuiteと同じです。

クエリビルダーに関する注意事項

- 名前付きパラメータのプレフィックスは、「@」を設定してください。

- 引用符(“) で囲んでも大文字のオブジェクト名を使用することはできません。
  クエリビルダーではなく、[SQLステートメントの入力]画面で、引用符(“) で囲んだ大文字のオブジェクト名を含むSQL文を入力して使用してください。
- 以下のSQL文をFilterに指定して正しいSQL文を生成できません。
  - PostgreSQLの固有演算子(<<、:: など)を使ったSQL文
  - 「AS、FROM、IN、OVER」などのキーワードがある関数を使ったSQL文

    例: extract(field from timestamp)、RANK( ) OVER
  - SQL規約に規定されている関数と関数名は同じであるが、引数の異なる関数を使ったSQL文

メタデータに関する注意事項

- GetSchemaTableでメタデータを取得する前にExecuteReaderメソッドを実行する場合、ExecuteReaderメソッドの引数にCommandBehavior.KeyInfoを指定する必要があります。

例

```
NpgsqlDataReader  ndr=cmd.ExecuteReader(CommandBehavior.KeyInfo);
DataTable dt = dr.GetSchemaTable();
```

更新系SQL文の自動生成に関する注意事項

以下の更新不可能な問合せを含むSQL文に対し、更新系のSQL文が生成され、そのSQL文が実行できない場合があります。

- 導出表を含む問合せ
- 選択リストに同じ列名を含む問合せ

更新系SQL文は以下の場合に自動生成されます。

- NpgsqlCommandBuilderで更新文を取得
- NpgsqlDataAdapterでデータ更新を実行

分散トランザクションに関する注意事項

トランザクションスコープを利用するアプリケーションは、Microsoft Distributed Transaction Coordinator(MSDTC)と連携することで、分散トランザクションとして動作することができます。この場合、以下の注意が必要です。

- データベースサーバに同時接続する各トランザクションに対して「PREPARE TRANSACTION」を発行できるようにするため、max_prepared_transactionsパラメータの値をmax_connectionパラメータの値より大きくしてください。
- トランザクションスコープ内の各トランザクションが、別々のコネクションを使って同じ資源にアクセスすると、データベースサーバでは異なるアプリケーションから要求があるように見え、デッドロックが発生する可能性がありますので注意してください。あらかじめトランザクションスコープにタイムアウト値を設定しておけば、デッドロックを解除することができます。

4.5 アンインストール

.NET Data ProviderおよびEntity Framework Coreのアンインストールについて説明します。

4.5.1 Npgsqlのアンインストール

Npgsqlは、プロジェクト単位でインストールされています。アンインストールは以下の手順で行います。

L

- Linuxの場合

プロジェクトファイルからパッケージ参照を削除します。

```
dotnet remove <プロジェクトファイルのパス> package Npgsql
```

W

- Windows(R)の場合

1. Visual Studioで、Npgsqlがインストールされているプロジェクトを開きます。
2. [ツール] - [NuGet パッケージ マネージャー] - [ソリューションの NuGet パッケージの管理]をクリックします。
3. Npgsqlがインストールされているすべてのプロジェクトを選択し、[アンインストール]をクリックします。
   または、プラグインパッケージが削除されている場合は、ソリューションエクスプローラで[依存関係]/ [NuGet]ノードを開き、アンインストールが必要なプラグインを削除します。

4.5.2 .NET Data Provider タイププラグインのアンインストール

.NET Data Provider タイププラグインは、プロジェクト単位でインストールされています。アンインストールは以下の手順で行います。

L

- Linuxの場合

プロジェクトファイルからパッケージ参照を削除します(例:Npgsql.NodaTime)。

```
dotnet remove <プロジェクトファイルのパス> package Npgsql.NodaTime
```

W

- Windows(R)の場合

1. Visual Studioで、削除するタイププラグインがインストールされているプロジェクトを開きます。
2. [ツール] - [NuGet パッケージ マネージャー] - [ソリューションの NuGet パッケージの管理]をクリックします。
3. Npgsql Pluginがインストールされているすべてのプロジェクトを選択し、[アンインストール]をクリックします。
   または、プラグインパッケージが削除されている場合は、ソリューションエクスプローラで[依存関係]/ [NuGet]ノードを開き、アンインストールが必要なプラグインを削除します。

4.5.3 Entity Framework Coreのアンインストール

Entity Framework Coreは、プロジェクト単位でインストールされています。アンインストールは以下の手順で行います。

L

- Linuxの場合

プロジェクトファイルからパッケージ参照を削除します。

```
dotnet remove <プロジェクトファイルのパス> package Npgsql.EntityFrameworkCore.PostgreSQL
```

W

- Windows(R)の場合

1. Visual StudioでEntity Framework Coreがインストールされているプロジェクトを開きます。
2. [ツール] - [NuGet パッケージ マネージャー] - [ソリューションの NuGet パッケージの管理]をクリックします。
3. Entity Framework Coreがインストールされているすべてのプロジェクトを選択し、[アンインストール]をクリックします。
   または、Entity Frameworkパッケージが削除されている場合は、ソリューションエクスプローラで[依存関係]/ [NuGet]ノードを開き、アンインストールが必要なパッケージを削除します。

4.5.4 Npgsql.OpenTelemetryのアンインストール

Npgsql.OpenTelemetryは、プロジェクト単位でインストールされています。アンインストールは以下の手順で行います。

L

・ Linuxの場合

プロジェクトファイルからパッケージ参照を削除します。

```
dotnet remove <プロジェクトファイルのパス> package Npgsql.OpenTelemetry
```

W

・ Windows(R)の場合

1. Visual StudioでNpgsql.OpenTelemetryがインストールされているプロジェクトを開きます。
2. [ツール] - [NuGet パッケージ マネージャー] - [ソリューションの NuGet パッケージの管理]をクリックします。
3. Npgsql.OpenTelemetryがインストールされているすべてのプロジェクトを選択し、[アンインストール]をクリックします。
   または、Npgsql.OpenTelemetryパッケージが削除されている場合は、ソリューションエクスプローラで[依存関係]/[NuGet]ノードを開き、アンインストールが必要なパッケージを削除します。

第5章 C言語用ライブラリ(libpq)

C言語用ライブラリの利用方法について説明します。

5.1 開発環境

開発、および実行するアーキテクチャのFujitsu Enterprise Postgres Clientパッケージをインストールしてください。

参照

C言語アプリケーションの開発に必要なCコンパイラについては、“導入ガイド(クライアント編)”を参照してください。

5.2 セットアップ

C言語ライブラリを利用する場合の環境設定、および通信データの暗号化方法について説明します。

5.2.1 環境設定

libpqを使用するアプリケーションを実行するためには、以下のように環境変数を設定してください。

L

Linuxの場合

- アプリケーション実行時に必要
  - PGLOCALEDIR
    <Fujitsu Enterprise Postgres クライアント機能のインストールディレクトリ>/share/locale

例

“*<x>*”は製品のバージョンを示します。

```
> PGLOCALEDIR=/opt/fsepv<x>client64/share/locale;export PGLOCALEDIR
```

W

Windows(R)の場合

- コンパイル/リンク時に必要
  - LIB
    <Fujitsu Enterprise Postgres クライアント機能のインストールディレクトリ>¥lib
  - INCLUDE
    <Fujitsu Enterprise Postgres クライアント機能のインストールディレクトリ>¥include
- アプリケーション実行時に必要
  - PATH
    <Fujitsu Enterprise Postgres クライアント機能のインストールディレクトリ>¥lib
  - PGLOCALEDIR
    <Fujitsu Enterprise Postgres クライアント機能のインストールディレクトリ>¥share¥locale

例

64ビットのオペレーティングシステムに、32ビットのクライアントパッケージをインストールしている場合の例です。なお、“<x>”は製品のバージョンを示します。

```
> SET PATH=%ProgramFiles(x86)%¥Fujitsu¥fsepv<x>client32¥lib;%PATH%
> SET LIB=%ProgramFiles(x86)%¥Fujitsu¥fsepv<x>client32¥lib;%LIB%
> SET INCLUDE=%ProgramFiles(x86)%¥Fujitsu¥fsepv<x>client32¥include;%INCLUDE%
> SET PGLOCALEDIR=%ProgramFiles(x86)%¥Fujitsu¥fsepv<x>client32¥share¥locale
```

5.2.2 メッセージの言語およびアプリケーションが使用する符号化方式の設定

アプリケーション実行環境の言語の設定、およびアプリケーションが使用する符号化方式の設定について説明します。

言語の設定

アプリケーション実行環境の言語設定は、データベースサーバのメッセージロケールの設定と合わせる必要があります。

アプリケーションが出力するメッセージの中には、アプリケーション側のメッセージに、データベースサーバから送られたメッセージを埋め込む場合があります。このとき、アプリケーション側のメッセージは、アプリケーション側のメッセージロケールに従い、データベースサーバから送られるメッセージは、データベースサーバ側のメッセージロケールに従います。そのため、両方のメッセージロケールが一致していない場合には、言語や符号化方式が混在します。符号化方式が一致しない場合には、文字化けが発生します。

L

- Linuxの場合

プロセスのロケールのLC_MESSAGESカテゴリが、データベースサーバのメッセージロケールと一致するように設定してください。環境変数を用いるなどいくつかの方法があります。詳細は、setlocale関数のオペレーティングシステムに付属するドキュメントを参照してください。

例

setlocale関数で“ja_JP.UTF-8”の指定例

```
setlocale(LC_ALL,"ja_JP.UTF-8");
```

LC_ALLに指定することにより、LC_MESSAGEカテゴリに設定を適用しています。

W

- Windows(R)の場合

OSのロケールをデータベースサーバのメッセージロケールの設定と合わせてください。

符号化方式の設定

アプリケーションに埋め込まれ、データベースに渡される符号化方式と、実行時のクライアント符号化方式の設定は同じにしてください。データベースサーバ側で正しく符号化方式を変換できなくなります。

アプリケーションの符号化方式は、以下のいずれかの方法で設定してください。

- 実行時の環境変数PGCLIENTENCODINGに設定する。
- 接続文字列のclient_encodingキーワードに設定する。
- PQsetClientEncoding関数を使って設定する。

参照

設定できる符号化方式を表す文字列は、“PostgreSQL Documentation”の“Server Administration”の“Supported Character Sets”を参照してください。

例えば、Unicode、8ビットの場合は、“UTF8”という文字列を設定します。

注意

コマンドプロンプトに結果を出力する際、文字化けする場合があります。文字化けした場合は、コマンドプロンプトのフォントの設定を見直してください。

5.2.3 通信データを暗号化する場合の設定

通信データの暗号化機能を利用してリモートアクセスを行う場合は、以下のいずれかの方法で設定してください。

環境変数により外部から設定する場合

環境変数PGSSLMODEに「require」、「verify-ca」、「verify-full」のいずれかを指定してください。

さらに、データベースサーバの成りすましを防御するためには、環境変数PGSSLROOTCERTおよびPGSSLCRLの各パラメータの設定が必要です。

参照

環境変数の詳細については、“PostgreSQL Documentation”の“Client Interfaces”の“Environment Variables”を参照してください。

接続URIに指定する場合

接続URIの“sslmode”パラメータに「require」、「verify-ca」、「verify-full」のいずれかを指定してください。

さらに、データベースサーバの成りすましから防御するためには、“sslcert”、“sslkey”、“sslrootcert”、“sslcrl”の各パラメータの設定も必要です。

参照

通信データの暗号化についての詳細は、“PostgreSQL Documentation”の“Server Administration”の“Secure TCP/IP Connections with SSL”を参照してください。

5.3 データベースへの接続

ポイント

接続サービスファイルを用いて接続先を指定することを推奨します。接続サービスファイルには、接続先情報やコネクションに対して設定する各種のチューニング情報を1セットとして名前(サービス名)を定義します。データベース接続時には、接続サービスファイルに定義されたサービス名を用いることで、接続情報の変更によるアプリケーションの修正が不要になります。

“PostgreSQL Documentation”の“Client Interfaces”の“The Connection Service File”を参照してください。

参照

“PostgreSQL Documentation”の“Client Interfaces”の“Database Connection Control Functions”を参照してください。

また、接続文字列に設定する情報については、“C言語による埋め込みSQL”の“6.3 データベースへの接続”を参照してください。

5.4 アプリケーション開発

参照

アプリケーションの作成方法については、“PostgreSQL Documentation”の“Client Interfaces”の“libpq - C Library”を参照してください。

ただし、C言語用ライブラリを使用する場合、以下の点がPostgreSQLのCライブラリ(libpq)と異なります。

5.4.1 アプリケーションのコンパイル

アプリケーションのコンパイル時は、以下のパスを指定します。

パスの指定方法はご利用のコンパイラのドキュメントを参照してください。

また、共有ライブラリを利用する場合、“A.6 共有ライブラリを利用するアプリケーションのビルド・実行方法”を参照してください。

L Linuxの場合

L 表5.1 インクルードファイルとライブラリのパス

パスの種類	パス名
インクルードファイルのパス	<Fujitsu Enterprise Postgres *クライアント機能のインストールディレクトリ*>/include
ライブラリのパス	<Fujitsu Enterprise Postgres *クライアント機能のインストールディレクトリ*>/lib

L 表5.2 C言語用ライブラリ(libpqライブラリ)

ライブラリの種類	ライブラリ名
動的ライブラリ	libpq.so
静的ライブラリ	libpq.a

W Windows(R)の場合

環境変数で、インクルードファイルとライブラリのパスが設定されている場合、コンパイル時に、下記のパスの指定は不要です。

W 表5.3 インクルードファイルとライブラリのパス

パスの種類	パス名
インクルードファイルのパス	<Fujitsu Enterprise Postgres *クライアント機能のインストールディレクトリ*>¥include
ライブラリのパス	<Fujitsu Enterprise Postgres *クライアント機能のインストールディレクトリ*>¥lib

W 表5.4 C言語用ライブラリ(libpqライブラリ)

ライブラリの種類	ライブラリ名
リンク用ライブラリ	libpq.lib
動的ライブラリ	libpq.dll

5.4.2 データベース多重化運用時のアプリケーション作成

データベース多重化運用時のアプリケーション作成において考慮する事項を説明します。

参照

- データベース多重化運用についての詳細は、“クラスタ運用ガイド(データベース多重化編)”を参照してください。
- クラスタソフトウェアと連携したフェイルオーバ運用での、アプリケーション作成における考慮事項については、“クラスタ運用ガイド(PRIMECLUSTER編)”の“アプリケーションの作成”を参照してください。

5.4.2.1 アプリケーションの接続先切り替えが発生した場合のエラーと対処

データベース多重化運用時に、アプリケーションの接続先の切り替えが発生した場合、明示的にコネクションを切断し、コネクションの再接続またはアプリケーションを再実行してください。

以下に切り替え発生時のエラーと対処を示します。

状態		エラー情報	対処
サーバダウン または Fujitsu Enterprise Postgresシステムダウン	アクセス中にダウンしたとき	PGRES_FATAL_ERROR(注1) 57P01(注2) NULL(注2)	切替え完了後、コネクションの再接続、またはアプリケーションを再実行してください。
	ダウン中にアクセスしたとき	CONNECTION_BAD(注3)	
スタンバイサーバへの切替え	アクセス中に切替えたとき	PGRES_FATAL_ERROR(注1) 57P01(注2) NULL(注2)	
	切替え中にアクセスしたとき	CONNECTION_BAD(注3)	

注1: PQresultStatus()の返却値となります。

注2: PQresultErrorField()のPG_DIAG_SQLSTATEの返却値となります。

注3: PQstatus()の返却値となります。

第6章 C言語による埋め込みSQL

C言語による埋め込みSQLを利用したアプリケーション開発について説明します。

6.1 開発環境

開発、および実行するアーキテクチャのFujitsu Enterprise Postgres Clientパッケージをインストールしてください。

参照

C言語アプリケーションの開発に必要なCコンパイラについては、“導入ガイド(クライアント編)”を参照してください。

注意

C++言語はサポートしていません。埋め込みSQL部分をC言語で記述しライブラリ化してから、C++から利用してください。

6.2 セットアップ

6.2.1 環境設定

C言語による埋め込みSQLを利用する場合は、C言語用ライブラリ(libpq)を利用する場合と同様の環境設定が必要です。C言語用ライブラリでの環境設定については、“C言語用ライブラリ(libpq)”の“5.2.1 環境設定”を参照してください。

また、プレコンパイラecpgに対する以下のパスを環境変数PATHに設定してください。

L Linuxの場合

```
<Fujitsu Enterprise Postgres クライアント機能のインストールディレクトリ>/bin
```

W Windows(R)の場合

```
<Fujitsu Enterprise Postgres クライアント機能のインストールディレクトリ>¥bin
```

6.2.2 メッセージの言語およびアプリケーションが使用する符号化方式の設定

メッセージの言語およびアプリケーションが使用する符号化方式の設定については、C言語用ライブラリを利用する場合と同様の環境設定が必要です。

ただし、埋め込みSQLでは、符号化方式の設定において、PQsetClientEncoding関数は使用できません。埋め込みSQLでSETコマンドを使用し、client_encodingに符号化方式を指定してください

C言語用ライブラリでの設定については、“C言語用ライブラリ(libpq)”の“5.2.2 メッセージの言語およびアプリケーションが使用する符号化方式の設定”を参照してください。

6.2.3 通信データを暗号化する場合の設定

通信データを暗号化する場合は、C言語用ライブラリ(libpq)を利用する場合と同様の設定が必要です。

C言語用ライブラリでの環境設定については、“C言語用ライブラリ(libpq)”の“5.2.3 通信データを暗号化する場合の設定”を参照してください。

6.3 データベースへの接続

ポイント

- 接続サービスファイルを用いて接続先を指定することを推奨します。接続サービスファイルには、接続先情報やコネクションに対して設定する各種のチューニング情報を1セットとして名前(サービス名)を定義します。データベース接続時には、接続サービスファイルに定義されたサービス名を用いることで、接続情報の変更によるアプリケーションの修正が不要になります。
  “PostgreSQL Documentation”の“Client Interfaces”の“The Connection Service File”を参照してください。
- 接続サービスファイルを利用するには、以下のいずれかの方法があります。
  - 文字列リテラルまたはホスト変数を使用して、以下のように記述する方法

    tcp:postgresql://?service=my_service
  - 環境変数PGSERVICEにサービス名を設定し、かつCONNECT TO DEFAULTを用いる方法

以下に示すCONNECT文を利用してデータベースサーバへの接続を作成します。

書式

```
EXEC SQL CONNECT TO target [AS connection-name] [USER user-name];
```

target

次のいずれかの形式で記述します。

- dbname@host:port
- tcp:postgresql://host:port/dbname[?options]
- unix:postgresql://host[:port][/dbname][?options]
  (UNIXドメインソケットを使用する場合の記述方法)
- 上記形式のいずれかを含むSQL規約の文字列定数
- 上記形式のいずれかを含む文字変数への参照
- DEFAULT

user-name

次のいずれかの形式で記述します。

- username
- username/password
- username IDENTIFIED BY password
- username USING password

引数に関する説明

引数	説明
dbname	データベース名を指定します。
host	接続先のホスト名を指定します。
port	データベースサーバのポート番号を指定します。 省略した場合は、27500となります。
connection-name	1つのプログラム内で複数の接続を処理する場合に、コネクションを識別するためのコネクション名を指定します。
username	データベースへ接続するユーザー名を指定します。

引数	説明
	省略した場合は、そのアプリケーションを実行しているユーザーのオペレーティングシステム上の名前と同じです。
password	パスワードによる認証を必要とした場合に、パスワードを指定します。
options	タイムアウト時間を指定する場合は、以下のパラメータを指定します。複数のパラメータを指定する場合は&で繋ぎます。各パラメータの指定値は以下のとおりです。 ・ connect_timeout 接続時のタイムアウト時間を指定します。 単位は秒で0～2147483647を指定します。0、不当な値を指定した場合、省略した場合は無制限です。1を指定した場合は2が指定されたものとして扱います。指定された時間内にコネクションが接続できなかった場合はエラーとなります。 ・ keepalives キープアライブを有効にします。 0を指定した場合はキープアライブが無効、それ以外の数値を指定した場合はキープアライブが有効となります。省略値はキープアライブが有効です。キープアライブにより、データベースとの接続が無効と判断された場合はエラーとなります。 ・ keepalives_idle データベースとの通信が行われていない場合、キープアライブメッセージの送信を開始するまでの時間を指定します。 － Linuxの場合 単位は秒で1～32767を指定します。省略した場合はシステムのデフォルト値を使用します。 － Windows(R)の場合 単位は秒で1～2147483647を指定します。それ以外の値を指定した場合、および省略した場合は7200が指定されたものとして扱います。 ・ keepalives_interval キープアライブメッセージの応答がない場合に、何秒後に再送するかを指定します。 － Linuxの場合 単位は秒で1～32767を指定します。省略した場合はシステムのデフォルト値を使用します。 － Windows(R)の場合 単位は秒で1～2147483647を指定します。それ以外の値を指定した場合、および省略した場合は1が指定されたものとして扱います。 ・ keepalives_count キープアライブメッセージの再送回数を指定します。 － Linuxの場合 1～127の値を指定します。省略した場合はシステムのデフォルト値を使用します。 － Windows(R)の場合 本パラメータの指定に関係なく、システムのデフォルト値を使用します。 ・ tcp_user_timeout 接続が確立した後、クライアントからサーバへの送信時にTCP再送処理が動作した場合に、切断とみなすまでの時間を指定します。 － Linuxの場合

引数	説明
	単位はミリ秒で0～2147483647を指定します。0の場合はシステムのデフォルト値を使用します。省略した場合は0が指定されたものとして扱います。 － Windows(R)の場合 指定できません。

L

注意

tcp_user_timeoutパラメータに0以外を指定した場合、tcp_keepalives_idleパラメータおよびtcp_keepalives_intervalパラメータによる待機時間は無効となり、tcp_user_timeoutパラメータの指定値による待機時間となります。

アプリケーションの記述例

```
EXEC SQL CONNECT TO tcp:postgresql://sv1:27500/mydb?
connect_timeout=20&keepalives=1&keepalives_idle=20&keepalives_interval=5&keepalives_count=2 USER myuser/
myuser01;
```

6.4 アプリケーション開発

アプリケーションの作成方法については、“PostgreSQL Documentation”の“Client Interfaces”の“ECPG - Embedded SQL in C”を参照してください。

ただし、C言語による埋め込みSQLを使用する場合、以下の点がPostgreSQLのC言語による埋め込みSQL(ECPG)と異なります。

6.4.1 各国語データ型のサポート

ここでは、SQL埋め込みCプリプロセッサを用いて、各国語データ型を使用する方法を説明します。

NCHAR型に対応するC言語変数型は、下表のとおりです。また、ホスト変数の長さには、NCHAR型に指定した文字数×4+1の値を指定します。

データ型	ホスト変数型
NATIONAL CHARACTER(n)	NCHAR 変数名[n×4+1]
NATIONAL CHARACTER VARYING(n)	NVARCHAR 変数名[n×4+1]

参照

文字列型の使用方法については、“PostgreSQL Documentation”の“Client Interfaces”の“Handling Character Strings”を参照してください。

6.4.2 アプリケーションのコンパイル

C言語による埋め込みSQLソースファイルの名前には拡張子pgcを付けてください。

pgcファイルをecpgコマンドでプレコンパイルするとC言語のソースファイルが作成されるので、Cコンパイラを使用してコンパイルしてください。

プレコンパイルの例

```
ecpg testproc.pgc
```

SQL文に対して、オプティマイザヒントのブロックコメントを指定している場合は、ecpgコマンドに以下のオプションを指定します。

--enable-hint

オプティマイザヒントのブロックコメント(以降、ヒント句と呼びます)を有効にします。本オプションを指定しない場合、ecpgのプレコンパイルによりヒント句が取り除かれ、ヒント句が無効となります。

ヒント句が指定可能なSQL文は、SELECT、INSERT、UPDATEおよびDELETEです。

ヒント句が指定可能な位置は、SELECT、INSERT、UPDATE、DELETEまたはWITHのいずれかのキーワードの直後のみです。それ以外の場所に指定した場合は、構文エラーとなります。

ヒント句の指定例

```
EXEC SQL SELECT /*+ IndexScan(prod ix01) */ name_id INTO :name_id FROM prod WHERE id = 1;
```

参照

ヒント句については、以下を参照してください。

- pg_hint_planで実行計画を制御する

  https://www.fujitsu.com/jp/products/software/resources/feature-stories/postgres/article-index/pg-hint-plan/

注意

埋め込みSQLのソースファイルについて、以下の注意事項があります。

- SJISまたはUTF-16で表現されたマルチバイトコードをEXEC SQLで指定された文やホスト変数宣言に含めることはできません。
- BOM(Byte Order Mark)付きのUTF-8は、BOMをソースコードと認識してコンパイル時にエラーになるため、使用しないでください。
- ホスト変数名にマルチバイト文字は使用できません。
- TYPE名にマルチバイト文字を使用した場合には、定義はできますが使用できません。

プレコンパイルにより出力されたC言語のアプリケーションのコンパイル時は、以下のパスを指定します。

パスの指定方法はご利用のコンパイラのドキュメントを参照してください。

また、共有ライブラリを利用する場合、“A.6 共有ライブラリを利用するアプリケーションのビルド・実行方法”を参照してください。

L Linuxの場合

L 表6.1 インクルードファイルとライブラリのパス

パスの種類	パス名
インクルードファイルのパス	*<Fujitsu Enterprise Postgres クライアント機能のインストールディレクトリ>* /include
ライブラリのパス	*<Fujitsu Enterprise Postgres クライアント機能のインストールディレクトリ>*/lib

L

表6.2 C言語用ライブラリ

ライブラリの種類	ライブラリ名	備考
動的ライブラリ	libecpg.so	
	libpgtypes.so	pgtypesライブラリを使用する場合
静的ライブラリ	libecpg.a	
	libpgtypes.a	pgtypesライブラリを使用する場合

W

Windows(R)の場合

環境変数で、インクルードファイルとライブラリのパスが設定されている場合、コンパイル時に、下記のパスの指定は不要です。

W

表6.3 インクルードファイルとライブラリのパス

パスの種類	パス名
インクルードファイルのパス	*<Fujitsu Enterprise Postgres クライアント機能のインストールディレクトリ>*¥include
ライブラリのパス	*<Fujitsu Enterprise Postgres クライアント機能のインストールディレクトリ>*¥lib

W

表6.4 C言語用ライブラリ

ライブラリの種類	ライブラリ名	備考
リンク用ライブラリ	libecpg.lib	
	libpgtypes.lib	pgtypesライブラリを使用する場合
動的ライブラリ	libecpg.dll	
	libpgtypes.dll	pgtypesライブラリを使用する場合

注意

- Windows(R) におけるlibecpgライブラリは、release かつ multithreadedのオプションで作成されています。本ライブラリに含まれるECPGdebug関数を用いる場合には、本ライブラリを使用するすべてのプログラムにおいて、releaseフラグかつmultithreadedフラグを使用してコンパイルしてください。このときに、libecpg.dllを用いる場合にはdynamicフラグ、libecpg.libを用いる場合にはstaticフラグを使用してください。
  ECPGdebug関数については、“PostgreSQL Documentation”の“Client Interfaces”の“Library Functions”を参照してください。
- clコマンドは以下のいずれかのコードページで記述されたプログラムを入力として期待しているため、これらのコードページに変換してからコンパイルおよびリンクを行ってください(詳細は、Microsoftのマニュアルを参照してください)。
  - ANSIコンソールコードページ(例: 日本語の場合は Shift-JIS)
  - BOM(Byte Order Mark)付き、またはBOMなしのUTF-16リトルエンディアン
  - BOM付き、またはBOMなしのUTF-16ビッグエンディアン
  - BOM付きのUTF-8

  また、clコマンドは、プログラム中の文字列などをANSIコンソールコードページに変換してモジュールを生成するため、データベースサーバとの間で送受信されるデータはANSIコンソールコードページになります。したがって、ANSIコンソールコードページに該当する符号化方式をクライアントの符号化方式に設定してください。

  クライアントの符号化方式の設定方法は、“PostgreSQL Documentation”の“Server Administration”の“Character Set Support”を参照してください。
  (例: 日本語において環境変数を使用する場合、PGCLIENTENCODINGにSJISを設定する)

6.4.3 一括INSERT

一括INSERTについて説明します。

記述形式

```
EXEC SQL [ AT connection ] [ FOR { number_of_rows | ARRAY_SIZE } ]
    INSERT INTO table_name [ ( column_name [, ...] ) ]
    { VALUES ( { expr | DEFAULT } [, ...] ) [, ...] | query }
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...]
     INTO output_Host_var [ [ INDICATOR ] indicator_var ] [, ...] ];
```

説明

一括INSERTは複数行のデータを一括挿入します。

INSERT文のVALUES句にデータを格納した配列ホスト変数を指定することで、配列の各要素のデータを一括で挿入できます。INSERT文の直前にFOR句で挿入回数を指定して本機能を利用します。

FOR 句

FOR句には*number_of_rows*またはARRAY_SIZEを使って、挿入する回数を指定します。FOR句はINSERT文にのみ指定可能であり、他の更新文には指定できません。

number_of_rows、ARRAY_SIZE

指定された回数だけ、挿入処理が実行されます。ただし、1の場合には、アプリケーションの実行時にFOR句が省略されたものとみなします。この場合は、PostgreSQL DocumentationのINSERTの仕様に従います。

FOR句は、short型、int型、long long型のいずれかのホスト変数、またはリテラルで指定します。

配列のすべての要素をテーブルに挿入する場合は、ARRAY_SIZEを指定します。ARRAY_SIZEを指定する場合には、*expr*に1個以上の配列を指定してください。

*expr*に2個以上の配列を指定した場合には、ARRAY_SIZEは配列の中の要素数が最小の配列の要素数とみなされます。

*number_of_rows*またはARRAY_SIZEは、*expr*、*output_Host_var*、*indicator_val*で指定されたすべての配列のうちの最小の要素数を超える値でなければなりません。

FOR句の指定例を以下に示します。

```
int  number_of_rows = 10;
int id[25];
char name[25][10];

EXEC SQL FOR :number_of_rows    /* will process 10 rows */
INSERT INTO prod (name, id) VALUES (:name, :id);

EXEC SQL FOR ARRAY_SIZE          /* will process 25 rows */
INSERT INTO prod (name, id) VALUES (:name, :id);
```

expr

テーブルに挿入する値を指定します。配列ホスト変数、ホスト変数定数、文字列、ポインター変数を指定できます。構造体型の配列やポインター変数の配列は指定できません。

ポインター変数とARRAY_SIZEを同時に使用しないでください。ポインター変数が指す領域にある要素数を判断できないためです。

query

挿入する行を提供する問い合わせ(SELECT文)を指定します。*query*が返却する行数は1行でなければなりません。2行以上返却された場合にはエラーになります。ARRAY_SIZEと同時には使用できません。

output_Host_var、indicator_val

配列ホスト変数、またはポインター変数でなければなりません。

エラーメッセージ

一括INSERT使用時にエラーとなった場合、以下のメッセージが出力されます。

メッセージ

FOR句の値は正の整数でなければなりません

原因

*number_of_rows*に、0以下の値が指定されています。

対処

*number_of_rows*には、1以上の値を指定してください。

メッセージ

FOR句にARRAY_SIZEを指定する場合は、配列ホスト変数を使用する必要があります

原因

FOR句にARRAY_SIZEが指定されていますが、VALUES句に配列ホスト変数が指定されていません。

対処

ARRAY_SIZEを指定する場合、VALUES句には、配列ホスト変数を1つ以上指定してください

メッセージ

行番号%dにおいて、SELECT..INTO文が返却する行が多すぎます

原因

INSERT文中の“SELECT ... INTO”で2行以上のデータが返却されています。

対処

*number_of_rows*が2以上の場合、INSERT文中の“SELECT ... INTO”で返却可能な行の最大数は、1行です。

注意事項

一括INSERT使用時の注意事項を説明します。

- VALUES句には、構造体型の配列は指定できません。
- VALUES句には、ポインター変数の配列は指定できません。
- ECPGは、一つのINSERT文にWITH句の使用をサポートします。一括INSERTでは、WITH句は使用できません。
- 埋め込みSQLでは、ポインター変数のサイズを計算しません。複数要素が含まれるポインター変数を使用している場合、*number_of_rows*に、要素数以下の値を設定する必要があります。
- エラーが発生した場合には、一括INSERTのすべての処置がロールバックされるので、一行も挿入されていない状態になります。ただし、RETURNING句を使用し、かつ、挿入に成功した後の行の取得でエラーがあった場合には、挿入処理はロールバックされません。

使用例

一括INSERTの使用例を示します。

基本的な一括INSERT

```
int in_f1[4] = {1,2,3,4};
...
EXEC SQL FOR 3 INSERT INTO target (f1) VALUES (:in_f1);
```

挿入する行数がFOR句で“3”と指定されているので、配列の各要素のデータのうち、先頭から3要素がテーブルに挿入されます。targetテーブルは、以下のようになります。

```
 f1
----
  1
  2
  3
(3 rows)
```

また、挿入する行数をホスト変数として、FOR句に指定することができます。この場合も、上記と同様の実行結果となります。

```
int num = 3;
int in_f1[4] = {1,2,3,4};
...
EXEC SQL FOR :num INSERT INTO target (f1) VALUES (:in_f1);
```

定数値の挿入

挿入値として定数値を指定します。

```
EXEC SQL FOR 3 INSERT INTO target (f1,f2) VALUES (DEFAULT,'hello');
```

DEFAULTは、列“f1”に“0”を設定とした場合、targetテーブルは、以下のようになります。

```
f1 |  f2
---+-------
 0 | hello
 0 | hello
 0 | hello
(3 rows)
```

ARRAY_SIZEを指定する

ARRAY_SIZEは、配列のすべての要素を挿入する場合に指定します。

```
    int in_f1[4] = {1,2,3,4};
...
EXEC SQL FOR ARRAY_SIZE INSERT INTO target (f1) VALUES (:in_f1);
```

上記の例では、targetテーブルに4行のデータが挿入されます。

注意

VALUES句に、複数の配列のホスト変数が指定された場合、要素数が最も小さい配列の要素数が指定されたものとして、その行数分の値が挿入されます。以下に例を示します。

```
int in_f1[4] = {1,2,3,4};
char in_f3[3][10] = {"one", "two", "three"};
...
EXEC SQL FOR ARRAY_SIZE INSERT INTO target (f1,f3) VALUES (:in_f1,:in_f3);
```

上記の例では、配列の要素数は“4”と“3”ですが、ARRAY_SIZEには、最小の値である“3”が設定されるため、targetテーブルに3行のデータが挿入されます。したがって、テーブルの内容は以下のようになります。

```
f1 |  f3
----+-------
  1 | one
  2 | two
  3 | three
(3 rows)
```

ポインター変数の値を指定する

挿入値に、複数の要素を含むポインター変数の値を指定します。

```
int *in_pf1 = NULL;
in_pf1 = (int*)malloc(4*sizeof(int));
in_pf1[0]=1;
in_pf1[1]=2;
in_pf1[2]=3;
in_pf1[3]=4;
...
EXEC SQL FOR 4 INSERT INTO target (f1) values (:in_pf1);
```

上記の例では、targetテーブルに4行挿入されます。

問い合わせ(SELECT文)の結果を指定する

挿入値に、問い合わせ(SELECT文)の結果を指定します。

```
EXEC SQL FOR 4 INSERT INTO target(f1) SELECT age FROM source WHERE name LIKE 'foo';
```

上記の例で、問い合わせの結果、1行が返却されたと仮定すると、targetテーブルには、同じ行が4回挿入されます。

注意

FOR句に“2”以上を指定した場合、問い合わせの結果が2行以上の場合、INSERT文はエラーとなります。

FOR句に“1”を指定した場合、SELECT文で返却されるすべての行がテーブルに挿入されます。

```
EXEC SQL FOR 1 INSERT INTO target(f1) SELECT age FROM source;
```

この場合、FOR句に指定された“1”は、返却されたすべての行を指します。

RETURNING句を使用する

一括INSERTでは、通常のINSERT文と同様にRETURNING句を使用できます。以下に例を示します。

```
int out_f1[4];
int in_f1[4] = {1,2,3,4};
...
EXEC SQL FOR 3 INSERT INTO target (f1) VALUES (:in_f1) RETURNING f1 INTO :out_f1;
```

INSERT文の実行後、配列out_f1には、“1”、“2”および“3”の3つの要素が格納されます。

6.4.4 データベース多重化運用時のアプリケーション作成

データベース多重化運用時のアプリケーション作成において考慮する事項を説明します。

参照

- データベース多重化運用についての詳細は、“クラスタ運用ガイド(データベース多重化編)”を参照してください。
- クラスタソフトウェアと連携したフェイルオーバ運用での、アプリケーション作成における考慮事項については、“クラスタ運用ガイド(PRIMECLUSTER編)”の“アプリケーションの作成”を参照してください。

6.4.4.1 アプリケーションの接続先切り替えが発生した場合のエラーと対処

データベース多重化運用時に、アプリケーションの接続先の切り替えが発生した場合、明示的にコネクションを切断し、コネクションの再接続またはアプリケーションを再実行してください。

以下に切り替え発生時のエラーと対処を示します。

状態		エラー情報(注)	対処
サーバダウン または Fujitsu Enterprise Postgresシステムダウン	アクセス中にダウンしたとき	57P01 57P02 YE000 26000 40001	切替え完了後、コネクションの再接続、またはアプリケーションを再実行してください。
	ダウン中にアクセスしたとき	08001	
スタンバイサーバへの切替え	アクセス中に切替えたとき	57P01 57P02 YE000 26000 40001	
	切替え中にアクセスしたとき	08001	

注: SQLSTATEの返却値となります。

6.4.5 注意事項

マルチスレッドアプリケーション作成時の注意事項

C言語による埋め込みSQLにおけるDISCONNECT ALLは、プロセス内のすべてのコネクションを切断するため、コネクションを用いたすべての操作との間でスレッドアンセーフです。マルチスレッドアプリケーションでは使用しないでください。

第7章 COBOL言語による埋め込みSQL

COBOL言語による埋め込みSQLを利用したアプリケーション開発について説明します。

7.1 開発環境

開発、および実行するアーキテクチャのFujitsu Enterprise Postgres Clientパッケージをインストールしてください。

参照

COBOL言語アプリケーションの開発に必要なCOBOLコンパイラについては、“導入ガイド(クライアント編)”を参照してください。

7.2 セットアップ

7.2.1 環境設定

COBOL言語による埋め込みSQLを利用する場合は、C言語用ライブラリ(libpq)を利用する場合と同様の環境設定が必要です。C言語用ライブラリの環境設定については、“C言語用ライブラリ(libpq)”の“5.2.1 環境設定”を参照してください。

また、プレコンパイラecobpgに対する以下のパスを環境変数PATHに設定してください。

L Linuxの場合

```
<Fujitsu Enterprise Postgres クライアント機能のインストールディレクトリ>/bin
```

W Windows(R)の場合

```
<Fujitsu Enterprise Postgres クライアント機能のインストールディレクトリ>¥bin
```

7.2.2 メッセージの言語およびアプリケーションが使用する符号化方式の設定

メッセージの言語およびアプリケーションが使用する符号化方式の設定については、C言語用ライブラリを利用する場合と同様の環境設定が必要です。

ただし、埋め込みSQLでは、符号化方式の設定において、PQsetClientEncoding関数は使用できません。埋め込みSQLでSETコマンドを使用し、client_encodingに符号化方式を指定してください

C言語用ライブラリの設定については、“C言語用ライブラリ(libpq)”の“5.2.2 メッセージの言語およびアプリケーションが使用する符号化方式の設定”を参照してください。

7.2.3 通信データを暗号化する場合の設定

通信データを暗号化する場合は、C言語用ライブラリ(libpq)を利用する場合と同等の設定が必要です。

C言語用ライブラリでの環境設定については、“C言語用ライブラリ(libpq)”の“5.2.3 通信データを暗号化する場合の設定”を参照してください。

7.3 データベースへの接続

以下に示すCONNECT文を利用してデータベースサーバへの接続を作成します。

書式

```
EXEC SQL CONNECT TO target [AS connection-name] [USER user-name]END-EXEC.
```

target

次のいずれかの形式で記述します。

- dbname@host:port
- tcp:postgresql://host:port/dbname[?options]
- unix:postgresql://host[:port][/dbname][?options]
  (UNIXドメインソケットを使用する場合の記述方法)
- 上記形式のいずれかを含むSQL規約の文字列定数
- 上記形式のいずれかを含む文字変数への参照
- DEFAULT

注意

targetにDEFAULTを記述した場合、AS句およびUSER句は指定できません。

user-name

次のいずれかの形式で記述します。

- username
- username/password
- username IDENTIFIED BY password
- username USING password

引数に関する説明

引数	説明
dbname	データベース名を指定します。
host	接続先のホスト名を指定します。
port	データベースサーバのポート番号を指定します。 省略した場合は、27500となります。
connection-name	1つのプログラム内で複数の接続を処理する場合に、コネクションを識別するためのコネクション名を指定します。
username	データベースへ接続するユーザー名を指定します。 省略した場合は、そのアプリケーションを実行しているユーザーのオペレーティングシステム上の名前と同じです。
password	パスワードによる認証を必要とした場合に、パスワードを指定します。
options	タイムアウト時間を指定する場合は、以下のパラメータを指定します。複数のパラメータを指定する場合は&で繋ぎます。各パラメータの指定値は以下のとおりです。 ・ connect_timeout 接続時のタイムアウト時間を指定します。 単位は秒で0～2147483647を指定します。0、不当な値を指定した場合、省略した場合は無制限です。1を指定した場合は2が指定されたものとして扱います。指定された時間内にコネクションが接続できなかった場合はエラーとなります。 ・ keepalives キープアライブを有効にします。

引数	説明
	0を指定した場合はキープアライブが無効、それ以外の数値を指定した場合はキープアライブが有効となります。省略値はキープアライブが有効です。キープアライブにより、データベースとの接続が無効と判断された場合はエラーとなります。 ・ keepalives_idle データベースとの通信が行われていない場合、キープアライブメッセージの送信を開始するまでの時間を指定します。 － Linuxの場合 単位は秒で1～32767を指定します。省略した場合はシステムのデフォルト値を使用します。 － Windows(R)の場合 単位は秒で1～2147483647を指定します。それ以外の値を指定した場合、および省略した場合は7200が指定されたものとして扱います。 ・ keepalives_interval キープアライブメッセージの応答がない場合に、何秒後に再送するかを指定します。 － Linuxの場合 単位は秒で1～32767を指定します。省略した場合はシステムのデフォルト値を使用します。 － Windows(R)の場合 単位は秒で1～2147483647を指定します。それ以外の値を指定した場合、および省略した場合は1が指定されたものとして扱います。 ・ keepalives_count キープアライブメッセージの再送回数を指定します。 － Linuxの場合 1～127の値を指定します。省略した場合はシステムのデフォルト値を使用します。 － Windows(R)の場合 本パラメータの指定に関係なく、システムのデフォルト値を使用します。

アプリケーションの記述例

```
EXEC SQL CONNECT TO tcp:postgresql://sv1:27500/mydb?
connect_timeout=20&keepalives_idle=20&keepalives_interval=5&keepalives_count=2&keepalives=1 USER myuser/
myuser01 END-EXEC.
```

7.4 アプリケーション開発

アプリケーションの作成方法の詳細は、“付録D ECOBPG - COBOL言語による埋め込みSQL”を参照してください。

7.4.1 各国語データ型のサポート

ここでは、SQL埋め込みCOBOLプリプロセッサを用いて、各国語データ型を使用する方法を説明します。

各国語データ型に対応するCOBOL言語変数型は、下表のとおりです。また、ホスト変数の長さには、各国語データ型に指定する文字数を指定します。

各国語データ型	COBOL言語変数型
CHARACTER(n)	変数名 PIC N(n)

各国語データ型	COBOL言語変数型
NATIONAL CHARACTER(n)	
CHARACTER VARYING(n) NATIONAL CHARACTER VARYING(n)	変数名 PIC N(n) VARYING

各国語データ型に対応するCOBOL言語変数型を使用する場合は、環境変数ECOBPG_NCHARを指定する必要があります。

ECOBPG_NCHAR={ UTF16LE | UTF16BE | UTF32LE | UTF32BE | SJIS }

COBOL言語による埋込みSQLを使用したアプリケーションの各国語データ型に対応するCOBOL言語変数型のエンコードを指定します。

- UTF16LE:UTF-16のリトルエンディアン
- UTF16BE:UTF-16のビッグエンディアン
- UTF32LE:UTF-32のリトルエンディアン
- UTF32BE:UTF-32のビッグエンディアン
- SJIS:シフトSJIS

本環境変数を省略した場合はクライアントの符号化方式によって決定されます。

- UTF8の場合:UTF16 (エンディアンはクライアントシステムのエンディアンに従います)
- SJISの場合:SJIS

NetCOBOLによるコンパイル時に翻訳オプションでエンコードを指定した場合は、各国語データ型に指定したエンコードを環境変数ECOBPG_NCHARに指定する必要があります。

NetCOBOLの翻訳オプションと、指定する環境変数ECOBPG_NCHARの値の対応は、以下となります。

NetCOBOLの翻訳オプション	環境変数 ECOBPG_NCHAR
ENCODE(UTF8,UTF16,LE) RCS(UTF16,LE)	UTF16LE
ENCODE(UTF8,UTF16,BE) RCS(UTF16,BE)	UTF16BE
ENCODE(UTF8,UTF32,LE)	UTF32LE
ENCODE(UTF8,UTF32,BE)	UTF32BE
ENCODE(SJIS,SJIS)	SJIS
指定なし	指定不要

また、コンパイル後のアプリケーションのエンコードとアプリケーション実行時のロケールが異なる場合は、クライアントの符号化方式をアプリケーションのエンコードに指定する必要があります。

アプリケーションのエンコードと、アプリケーション実行時のロケールと、設定するクライアントの符号化方式の値の対応は、以下となります。

アプリケーションのエンコード	アプリケーション実行時のロケール	クライアントの符号化方式
UTF8	UTF8	UTF8
	SJIS	UTF8

アプリケーションのエンコード	アプリケーション実行時のロケール	クライアントの符号化方式
SJIS	UTF8	SJIS
	SJIS	SJIS

クライアントの符号化方式の設定方法については、“7.2.2 メッセージの言語およびアプリケーションが使用する符号化方式の設定”を参照してください。

以下に、各国語データ型のホスト変数の宣言例を示します。

```
01 DATA1 PIC N(10).
01 DATA2 PIC N(10) VARYING.
```

注意

- 各国語データ型のCOBOL言語変数には半角文字を格納しないでください。
- アプリケーションで取得する各国語データ型の列属性もCHAR型となります。
- NetCOBOLの機能である、ENCODING句によるエンコードの指定は使用できません。

7.4.2 アプリケーションのコンパイル

COBOL言語による埋め込みSQLソースファイルの名前には拡張子pcoを付けてください。

pcoファイルをecobpgコマンドでプレコンパイルするとCOBOL言語のソースファイルが作成されるので、COBOLコンパイラを使用してコンパイルしてください。

プレコンパイルの例

```
ecobpg testproc.pco
```

ecobpgコマンドのオプションにCOBOLコードの記法として、固定形式または可変形式を指定することができます。デフォルトは固定形式となります。

COBOLコードの記法およびオプションの指定方法については、“D.1 機能と操作における注意事項”と“D.12.1 ecobpg”を参照してください。

SQL文に対して、オプティマイザヒントのブロックコメントを指定している場合は、ecobpgコマンドに以下のオプションを指定します。

--enable-hint

オプティマイザヒントのブロックコメント(以降、ヒント句と呼びます)を有効にします。本オプションを指定しない場合、ecobpgのプレコンパイルによりヒント句が取り除かれ、ヒント句が無効となります。

ヒント句が指定可能なSQL文は、SELECT、INSERT、UPDATEおよびDELETEです。

ヒント句が指定可能な位置は、SELECT、INSERT、UPDATE、DELETEまたはWITHのいずれかのキーワードの直後のみです。それ以外の場所に指定した場合は、構文エラーとなります。

ヒント句の指定例

```
EXEC SQL SELECT /*+ IndexScan(prod ix01) */ name_id
INTO :name_id FROM prod WHERE id = 1 END-EXEC.
```

参照

ヒント句については、以下を参照してください。

ー pg_hint_planで実行計画を制御する

https://www.fujitsu.com/jp/products/software/resources/feature-stories/postgres/article-index/pg-hint-plan/

埋め込みSQLのソースファイルのエンコードと、プレコンパイル実行時のロケールが異なる場合は、ecobpgコマンドに以下のオプションを指定して、埋め込みSQLのソースファイルのエンコードを指定します。

-E エンコード

エンコードは“UTF8”、“SJIS”、“EUC_JP”のいずれかを設定することができます。

本オプションを省略した場合はロケールで処理します。

登録集ファイルのパス

ecobpgコマンドはエラーを取り扱うための集団項目sqlca_tを定義しており、以下のパスに格納された登録集ファイルに定義されています。

L Linuxの場合

登録集ファイルの名前	登録集ファイルの格納先
SQLCA-COBOL.cob	*<Fujitsu Enterprise Postgresクライアント機能のインストールディレクトリ>*/include

W Windows(R)の場合

登録集ファイルの名前	登録集ファイルの格納先
SQLCA-COBOL.cob	*<Fujitsu Enterprise Postgresクライアント機能のインストールディレクトリ>*¥include

ecobpgコマンドはCOBOLファイルを生成するときに、この登録集ファイルを複写するために、オプションを付けないCOPY文を挿入します。そのため、コンパイルするときに登録集ファイルの格納先のパスを指定してください。パスの指定方法は、使用するコンパイラの仕様に従ってください。

なお、拡張子cobが付かない同じ内容の登録集ファイルも用意されています。

参考

sqlca_tの詳細は、“D.7.2 sqlca”を参照してください。

使用するライブラリ

ecobpgによって生成されたアプリケーションは、ECPGのライブラリを介してPostgreSQLに接続します。ECPGのライブラリは、内部でlibpqのライブラリをロードします。

ライブラリのパス

ECPGライブラリの場所と名前は、“6.4.2 アプリケーションのコンパイル”、libpqライブラリの場所と名前は、“第5章 C言語用ライブラリ(libpq)”を参照してください。

また、共有ライブラリを利用する場合、“A.6 共有ライブラリを利用するアプリケーションのビルド・実行方法”を参照してください。

COBOLコンパイラは、様々なライブラリのリンク方法を提供しているので、パスやライブラリの指定方法は、使用するコンパイラの仕様に従ってください。

副プログラムのエントリ情報

動的プログラム構造を利用してECPGライブラリを利用する場合には、以下に格納されているエントリ情報を複写してください。詳細は、使用するコンパイラの仕様に従ってください。

L Linuxの場合

<Fujitsu Enterprise Postgres クライアント機能のインストールディレクトリ>/share/cobol_entry.info

W

Windows(R)の場合

```
<Fujitsu Enterprise Postgres クライアント機能のインストールディレクトリ>¥share¥cobol_entry.info
```

例

COBOL言語ライブラリを動的リンクするアプリケーションのコンパイル例を説明します。
なお、“<*x*>”は製品のバージョンを示します。

L

Linuxの場合

― Linux 64ビット版アプリケーションの場合

```
cobol -M  -o testproc -I/opt/fsepv<x>client64/include -L/opt/fsepv<x>client64/lib -lecpg -lpq
testproc.cob
```

― Linux 64ビット版アプリケーションの場合(DT_RUNPATHを設定する場合)

```
cobol -c -M -o testproc.o -I/opt/fsepv<x>client64/include testproc.cob
ld -dynamic-linker /lib64/ld-linux-x86-64.so.2 -rpath /opt/fsepv<x>client64/lib --enable-new-dtags -o
testproc /usr/lib64/crti.o /usr/lib64/crt1.o /usr/lib64/crtn.o testproc.o  -L/opt/fsepv<x>client64/
lib -lecpg -lpq -Bdynamic -L/opt/FJSVcbl64/lib -lrcobol -ldl -lc
```

W

Windows(R)の場合

64ビット版のオペレーティングシステムでコンパイルする例です。

― 64ビット版アプリケーションの場合

```
> SET LIB=%ProgramFiles%¥Fujitsu¥fsepv<x>client64¥lib;%LIB%
> cobol -I "%ProgramFiles%¥Fujitsu¥fsepv<x>client64¥include" -M testproc.cob
> link testproc.obj F4AGCIMP.LIB LIBCMT.LIB LIBECPG.LIB LIBPQ.LIB /OUT:testproc.exe
```

― 32ビット版アプリケーションの場合

[NetCOBOL V10.5以前の場合]

```
> SET LIB=%ProgramFiles(x86)%¥Fujitsu¥fsepv<x>client32¥lib;%LIB%
> cobol32 -I "%ProgramFiles(x86)%¥Fujitsu¥fsepv<x>client32¥include" -M testproc.cob
> link testproc.obj LIBC.LIB F3BICIMP.LIB LIBECPG.LIB LIBPQ.LIB /OUT:testproc.exe
```

[NetCOBOL V11.0以降の場合]

```
> SET LIB=%ProgramFiles(x86)%¥Fujitsu¥fsepv<x>client32¥lib;%LIB%
> cobol32 -I "%ProgramFiles(x86)%¥Fujitsu¥fsepv<x>client32¥include" -M testproc.cob
> link testproc.obj MSVCRT.LIB F3BICIMP.LIB LIBECPG.LIB LIBPQ.LIB /OUT:testproc.exe
```

7.4.3 一括INSERT

一括INSERTについて説明します。

記述形式

```
EXEC SQL [ AT connection ] [ FOR { number_of_rows | ARRAY_SIZE } ]
    INSERT INTO table_name [ ( column_name [, ...] ) ]
    { VALUES ( { expression | DEFAULT } [, ...] ) [, ...] | query }
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...]
    INTO output_host_var [ [ INDICATOR ] indicator_var ] [, ...] ] END-EXEC
```

説明

一括INSERTは複数行のデータを一括挿入します。

INSERT文のVALUES句にデータを格納した配列ホスト変数を指定することで、配列の各要素のデータを一括で挿入できます。INSERT文の直前にFOR句で挿入回数を指定して本機能を利用します。

FOR 句

FOR句には*number_of_rows*またはARRAY_SIZEを使って、挿入する回数を指定します。FOR句はINSERT文にのみ指定可能であり、他の更新文には指定できません。

number_of_rows、ARRAY_SIZE

指定された回数だけ、挿入処理が実行されます。ただし、1の場合には、アプリケーションの実行時にFOR句が省略されたものとみなします。この場合は、PostgreSQL DocumentationのINSERTの仕様に従います。

FOR句は、整数項目で定義されたホスト変数、またはリテラルで指定します。

配列のすべての要素をテーブルに挿入する場合は、ARRAY_SIZEを指定します。ARRAY_SIZEを指定する場合には、*expression*に1個以上の配列を指定してください。

*expression*に2個以上の配列を指定した場合には、ARRAY_SIZEは配列の中の要素数が最小の配列の要素数とみなされます。

*number_of_rows*またはARRAY_SIZEは、*expression*、*output_host_var*、*indicator_val*で指定されたすべての配列のうちの最小の要素数を超える値でなければなりません。

FOR句の指定例を以下に示します。

```
01 NUMBER-OF-ROWS PIC S9(9) COMP VALUE 10.
01 GROUP-ITEM.
05 ID1 PIC S9(9) OCCURS 25.
05 NAME PIC X(10) OCCURS 25.
* will process 10 rows
EXEC SQL FOR :NUMBER-OF-ROWS
INSERT INTO prod (name, id) VALUES (:NAME, :ID1) END-EXEC
* will process 25 rows
EXEC SQL FOR ARRAY_SIZE
INSERT INTO prod (name, id) VALUES (:NAME, :ID1) END-EXEC
```

expression

テーブルに挿入する値を指定します。配列ホスト変数、ホスト変数定数、文字列、ポインター変数を指定できます。構造体型の配列やポインター変数の配列は指定できません。

ポインター変数とARRAY_SIZEを同時に使用しないでください。ポインター変数が指す領域にある要素数を判断できないためです。

query

挿入する行を提供する問い合わせ(SELECT文)を指定します。*query*が返却する行数は1行でなければなりません。2行以上返却された場合にはエラーになります。ARRAY_SIZEと同時には使用できません。

output_host_var、indicator_val

配列ホスト変数、またはポインター変数でなければなりません。

エラーメッセージ

一括INSERT使用時にエラーとなった場合、以下のメッセージが出力されます。

メッセージ

FOR句の値は正の整数でなければなりません

原因

*number_of_rows*に、0以下の値が指定されています。

対処

*number_of_rows*には、1以上の値を指定してください。

メッセージ

FOR句にARRAY_SIZEを指定する場合は、配列ホスト変数を使用する必要があります

原因

FOR句にARRAY_SIZEが指定されていますが、VALUES句に配列ホスト変数が指定されていません。

対処

ARRAY_SIZEを指定する場合、VALUES句には、配列ホスト変数を1つ以上指定してください

メッセージ

行番号%dにおいて、SELECT..INTO文が返却する行が多すぎます

原因

INSERT文中の“SELECT ... INTO”で2行以上のデータが返却されています。

対処

*number_of_rows*が2以上の場合、INSERT文中の“SELECT ... INTO”で返却可能な行の最大数は、1行です。

注意事項

一括INSERT使用時の注意事項を説明します。

- VALUES句には、構造体型の配列は指定できません。
- VALUES句には、ポインター変数の配列は指定できません。
- ECOBPGは、一つのINSERT文にWITH句の使用をサポートします。一括INSERTでは、WITH句は使用できません。
- エラーが発生した場合には、一括INSERTのすべての処置がロールバックされるので、一行も挿入されていない状態になります。ただし、RETURNING句を使用し、かつ、挿入に成功した後の行の取得でエラーがあった場合には、挿入処理はロールバックされません。

使用例

一括INSERTの使用例を示します。

基本的な一括INSERT

```
01 GROUP-ITEM.
05 IN-F1 PIC S9(9) OCCURS 4.
MOVE 1 TO IN-F1(1)
MOVE 2 TO IN-F1(2)
MOVE 3 TO IN-F1(3)
MOVE 4 TO IN-F1(4)
...
EXEC SQL FOR 3 INSERT INTO target (f1) VALUES (:IN-F1) END-EXEC
```

挿入する行数がFOR句で“3”と指定されているので、配列の各要素のデータのうち、先頭から3要素がテーブルに挿入されます。targetテーブルは、以下のようになります。

```
f1
----
```

```
1
2
3
(3 rows)
```

また、挿入する行数をホスト変数として、FOR句に指定することができます。この場合も、上記と同様の実行結果となります。

```
01 NUM PIC S9(9) COMP VALUE 3.
01 GROUP-ITEM.
05 IN-F1 PIC S9(9) OCCURS 4.
MOVE 1 TO IN-F1(1)
MOVE 2 TO IN-F1(2)
MOVE 3 TO IN-F1(3)
MOVE 4 TO IN-F1(4)
...
EXEC SQL FOR :NUM INSERT INTO target (f1) VALUES (:IN-F1) END-EXEC
```

定数値の挿入

挿入値として定数値を指定します。

```
EXEC SQL FOR 3 INSERT INTO target (f1,f2) VALUES (DEFAULT,'hello') END-EXEC
```

DEFAULTは、列“f1”に“0”を設定とした場合、targetテーブルは、以下のようになります。

```
f1 | f2
---+-------
0  | hello
0  | hello
0  | hello
(3 rows)
```

ARRAY_SIZEを指定する

ARRAY_SIZEは、サイズを指定しないで、テーブルに配列のすべての要素を挿入する場合に指定します。

```
01 GROUP-ITEM.
05 IN-F1 PIC S9(9) OCCURS 4.
MOVE 1 TO IN-F1(1)
MOVE 2 TO IN-F1(2)
MOVE 3 TO IN-F1(3)
MOVE 4 TO IN-F1(4)
...
EXEC SQL FOR ARRAY_SIZE INSERT INTO target (f1) VALUES (:IN-F1) END-EXEC
```

注意

複数の配列のホスト変数入力値が与えられた場合、挿入する行の数は配列のサイズの最小値と同じになります。以下に例を示します。

```
01 GROUP-ITEM.
05 IN-F1 PIC S9(9) OCCURS 4.
05 IN-F3 PIC X(10) OCCURS 3.
MOVE 1 TO IN-F1(1)
MOVE 2 TO IN-F1(2)
MOVE 3 TO IN-F1(3)
MOVE 4 TO IN-F1(4)
MOVE "one" TO IN-F3(1)
MOVE "two" TO IN-F3(2)
MOVE "three" TO IN-F3(3)
```

```
...
EXEC SQL FOR ARRAY_SIZE INSERT INTO target (f1,f3) VALUES (:IN-F1,:IN-F3) END-EXEC
```

上記の例では、配列の要素数は“4”と“3”です。しかし、ARRAY_SIZEには、最小の値である“3”が設定されるため、targetテーブルに3行のデータが挿入されます。したがって、テーブルの内容は以下のようになります。

```
f1 | f3
---+-------
1  | one
2  | two
3  | three
(3 rows)
```

問い合わせ(SELECT文)の結果を指定する

挿入値に、問い合わせ(SELECT文)の結果を指定します。

```
EXEC SQL FOR 4 INSERT INTO target(f1) SELECT age FROM source WHERE name LIKE 'foo' END-EXEC
```

上記の例で、問い合わせの結果、1行が返却されたと仮定すると、targetテーブルには、同じ行が4回挿入されます。

注意

FOR句に“2”以上を指定した場合、問い合わせの結果が2行以上の場合、INSERT文はエラーとなります。

FOR句に“1”を指定した場合、SELECT文で返却されるすべての行がテーブルに挿入されます。

```
EXEC SQL FOR 1 INSERT INTO target(f1) SELECT age FROM source END-EXEC
```

この場合、FOR句に指定された“1”は、返却されたすべての行を指します。

RETURNING句を使用する

一括INSERTでは、通常のINSERT文と同様にRETURNING句を使用できます。以下に例を示します。

```
01 GROUP-ITEM.
05 IN-F1 PIC S9(9) OCCURS 4.
05 OUT-F1 PIC S9(9) OCCURS 4.
MOVE 1 TO IN-F1(1)
MOVE 2 TO IN-F1(2)
MOVE 3 TO IN-F1(3)
MOVE 4 TO IN-F1(4)
...
EXEC SQL FOR 3 INSERT INTO target (f1) VALUES (:IN-F1) RETURNING f1 INTO :OUT-F1 END-EXEC
```

上記のINSERT文を実行後、配列out_f1には、“1”、“2”および“3”の3つの要素が格納されます。

7.4.4 DECLARE STATEMENT

DECLARE STATEMENTについては、“PostgreSQL Documentation”の“DECLARE STATEMENT”を参照してください。

7.4.5 データベース多重化運用時のアプリケーション作成

データベース多重化運用時のアプリケーション作成において考慮する事項を説明します。

参照

- データベース多重化運用についての詳細は、“クラスタ運用ガイド(データベース多重化編)”を参照してください。

- クラスタソフトウェアと連携したフェイルオーバ運用での、アプリケーション作成における考慮事項については、“クラスタ運用ガイド(PRIMECLUSTER編)”の“アプリケーションの作成”を参照してください。

7.4.5.1 アプリケーションの接続先切り替えが発生した場合のエラーと対処

データベース多重化運用時に、アプリケーションの接続先の切り替えが発生した場合、明示的にコネクションを切断し、コネクションの再接続またはアプリケーションを再実行してください。

以下に切り替え発生時のエラーと対処を示します。

状態		エラー情報(注)	対処
サーバダウン または Fujitsu Enterprise Postgresシステムダウン	アクセス中にダウンしたとき	57P01 57P02 YE000 26000 40001	切替え完了後、コネクションの再接続、またはアプリケーションを再実行してください。
	ダウン中にアクセスしたとき	08001	
スタンバイサーバへの切替え	アクセス中に切替えたとき	57P01 57P02 YE000 26000 40001	
	切替え中にアクセスしたとき	08001	

注: SQLSTATEの返却値となります。

第8章 SQL リファレンス

Fujitsu Enterprise Postgresが拡張したSQL文の機能を説明します。

8.1 トリガ定義の拡張機能

トリガ定義の拡張機能について説明します。

8.1.1 CREATE TRIGGER

PostgreSQLが提供するCREATE TRIGGERに加え、DOオプションを指定して新しいトリガを定義できます。

記述形式

```
CREATE [ OR REPLACE ] [ CONSTRAINT ] TRIGGER name { BEFORE | AFTER | INSTEAD OF } { event [ OR ... ] }
    ON table_name
    [ FROM referenced_table_name ]
    [ NOT DEFERRABLE | [ DEFERRABLE ] [ INITIALLY IMMEDIATE | INITIALLY DEFERRED ] ]
    [ REFERENCING { { OLD | NEW } TABLE [ AS ] transition_relation_name } [ ... ] ]
    [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( condition ) ]
    { EXECUTE { FUNCTION | PROCEDURE } function_name ( arguments )
        | DO [ LANGUAGE lang_name ] code }
```

説明

CREATE TRIGGERについては、“PostgreSQL Documentation”の“Server Programming”の“Triggers”を参照してください。ここでは、DOオプションについて説明します。

DOオプションを使用して作成されたトリガは、指定したテーブルまたはビューと関連付けられ、特定のイベントが発生した時に、指定されたDO(無名コードブロック)の手続き言語で書かれたコードブロックを実行します。

パラメータ

lang_name

関数を実装している言語の名前です。CREATE TRIGGERとしてplpgsqlをサポートしています。

code

特定のイベントが発生した時に、指定した手続き言語で書かれたコードブロックを実行します。無名コードブロックは、関数のように事前に定義する必要がありません。作成方法は、各手続き言語のトリガプロシージャの作成方法に従います。

注意

- DOオプションを指定したトリガは、EXECUTE PROCEDUREを指定したトリガにREPLACEすることはできません。
- EXECUTE PROCEDUREを指定したトリガは、DOオプションを指定したトリガにREPLACEすることはできません。

使用例

accountsテーブルの行が更新される直前にDOで指定した手続き言語で書かれたコードブロックを実行します。

（LANGUAGEがplpgsqlの場合の例）

```
CREATE TRIGGER check_update
    BEFORE UPDATE ON accounts
    FOR EACH ROW
    DO $$BEGIN RETURN NEW; END;$$ ;
```

参考

DOオプションを指定してトリガが作成された場合、内部的に『“スキーマ名”.”on表名”_”トリガ名”_TRIGPROC(通番)』という関数が定義されます。

8.1.2 pgAdminでの定義方法

トリガ定義の拡張機能は、pgAdminでも利用可能です。

参照

pgAdminでの定義方法については、“pgAdminヘルプ”を参照してください。

第9章 Oracleデータベースとの互換性

Oracleデータベースとの互換機能を利用するための環境設定と、提供する機能について説明します。

9.1 概要

Oracleデータベースとの互換機能を提供します。この機能を利用することにより、既存のアプリケーションを改修するコストを削減し、Fujitsu Enterprise Postgresへの移行が容易になります。

以下の互換機能を提供します。

表9.1 Oracleデータベース互換機能の一覧

分類		Oracleデータベース互換機能	
		項目	概要
SQL	問合せ	外部結合演算子(+)	外部結合のための演算子
		DUAL表	システムが用意している表
	関数	DECODE	値を比較し変換
		SUBSTR	文字列の一部取り出し
		NVL	NULL値の変換
パッケージ		DBMS_ALERT	アラートの送信
		DBMS_ASSERT	入力値に対するアサーションの実施
		DBMS_OUTPUT	メッセージの送信
		DBMS_PIPE	セッション間通信の実行
		DBMS_RANDOM	乱数の生成
		DBMS_UTILITY	様々な関数の追加
		UTL_FILE	ファイルの操作
		DBMS_SQL	動的SQLの実行

参照

このほか、Oracleデータベースとの互換機能については、下記ファイルを参照してください。

L
- **Linuxの場合**
  <Fujitsu Enterprise Postgresのインストール先ディレクトリ>/share/doc/extension/README.asciidoc

W
- Windows(R)の場合
  <Fujitsu Enterprise Postgresのインストール先ディレクトリ>¥doc¥extension¥README.asciidoc

9.2 Oracleデータベース互換機能利用時の注意事項

Oracleデータベース互換機能利用時の注意事項を説明します。

9.2.1 search_pathの注意事項

Oracleデータベース互換機能で定義されるオブジェクトは、oracleスキーマに定義されます。そのため、本機能を利用する際には、postgresql.confの、“search_path”パラメータに、“oracle”を追加する必要があります。

```
search_path = '"$user", public, oracle'
```

参考

- search_pathは、スキーマ検索パスの優先順位を指定する機能です。
- search_pathについては、“PostgreSQL Documentation”の“Server Administration”の“Statement Behavior”を参照してください。

9.2.2 SUBSTRの注意事項

SUBSTRは、Fujitsu Enterprise PostgresとOracleデータベースにおいて異なる外部仕様で実装されています。

このため、SUBSTRを利用する場合は、どちらの仕様を優先するかを定義する必要があります。標準設定ではFujitsu Enterprise Postgresの仕様を優先して実行します。

Oracleデータベース互換のSUBSTRを利用する場合は、postgresql.confの、“search_path”パラメータに、“oracle”および“pg_catalog”を設定してください。この時、“pg_catalog”より前に“oracle”を設定する必要があります。

```
search_path = '"$user", public, oracle, pg_catalog'
```

9.2.3 アプリケーション開発用のインタフェースとの連携時の注意事項

アプリケーション開発用のインタフェースでは、“表9.1 Oracleデータベース互換機能の一覧”のSQLを利用できます。

なお、Oracleデータベース互換のSUBSTRを利用する場合、SearchPathパラメータとして、“oracle”および“pg_catalog”の前に、publicとSQL文中のスキーマ名の両方を指定してください。

9.3 問合せ

以下の、「問合せ」をサポートしています。

- 外部結合演算子(+)
- DUAL表

9.3.1 外部結合演算子(+)

WHERE句の条件式において、表結合として付帯したい表の列に外部結合演算子である(+)を付加することで結合表(OUTER JOIN)と同じ、外部結合を実現します。

記述形式

SELECT文

```
SELECT … [WHERE [NOT] joinCond ...] …
SELECT … [WHERE srchCond ]... ] …
```

結合条件

```
{ colSpec(+) = colSpec | colSpec = colSpec(+) }
```

注意

ここでは、SELECT文のWHERE句のみを抜粋しています。SELECT文全体の記述形式については、“PostgreSQL Documentation”の“Reference”の“SQL Commands”を参照してください。

一般規則

WHERE句

- WHERE句は、導出される表に対して探索条件(*srchCond*)、または結合条件(*joinCond*)を指定します。
- 探索条件は、評価の結果としてBOOLEAN型を返す任意の式です。この条件を満たさない行はすべて出力から取り除かれます。すべての変数に実際の行の値を代入して、式が真を返す場合、その行は条件を満たすとみなされます。
- 結合条件は、外部結合演算子を指定した比較条件です。結合条件を満たすすべての行と、結合条件を満たす行を除いた、一方の表のすべての行を返却します。
- 探索条件よりも結合条件の優先度が高くなります。したがって、結合条件で返却されたすべての行に対して探索条件が有効となります。
- 外部結合演算子を使用した問合せには、次の規則と制限事項があります。そのため、外部結合演算子よりも、FROM句の結合表(OUTER JOIN)を使用することをお薦めします。
  - 外部結合演算子は、WHERE句にのみ指定できます。
  - 外部結合演算子は、実表またはビューの列に対してのみ指定できます。
  - 複数の結合条件によって外部結合を行いたい場合、すべての結合条件に対して外部結合演算子を指定する必要があります。
  - 結合条件に定数を組み合わせる場合、対応する列指定に外部結合演算子を指定してください。指定していない場合は探索条件と評価されます。
  - 表T1の列に外部結合演算子を指定して表T2と結合し、表T1と表T3を探索条件を使用して結合した場合、表T1の外部結合の結果行は返却されません。
  - 結合条件の左右の列指定(*colSpec*)に、同一テーブル上の列を指定することはできません。
  - 外部結合演算子を列指定以外の式に指定することはできませんが、式を構成する列に対しては指定することができます。

外部結合演算子は結合表(OUTER JOIN)の機能と比較して以下の機能制限があります。外部結合演算子で使用できない機能を使用したい場合は、結合表(OUTER JOIN)を使用してください。

表9.2 外部結合演算子の機能範囲

結合表(OUTER JOIN)で使用できる機能	外部結合演算子
2つの表の外部結合	○
3つ以上の表の外部結合	○(注)
同一問合せ内での結合表との併用	×
結合条件へのOR論理演算子の使用	×
結合条件へのIN述語の使用	×
結合条件への副問合せの使用	×

○:使用できます
×:使用できません

注) 外部結合演算子における外部結合では、他の1つの表に対してのみ外部結合結果を返すことができます。そのため、表T1と表T2、および表T2と表T3の組み合わせで外部結合を行いたい場合、表T2に対して外部結合演算子を同時に指定することはできません。

例

【表の構成】

t1

col1	col2	col3
1001	AAAAA	1000
1002	BBBBB	2000
1003	CCCCC	3000

t2

col1	col2
1001	aaaaa
1002	bbbbb
1004	ddddd

例1) 次の例では、表t1に存在しないレコードを含めた表t2のすべてのレコードを返却します。

```
SELECT *
  FROM t1, t2
  WHERE t1.col1(+) = t2.col1;
 col1 |    col2    | col3 | col1 |    col2
------+------------+------+------+------------
 1001 | AAAAA      | 1000 | 1001 | aaaaa
 1002 | BBBBB      | 2000 | 1002 | bbbbb
      |            |      | 1004 | ddddd
(3 rows)
```

これは、次に示すFROM句の結合表(OUTER JOIN)の構文と同じ問合せです。

```
SELECT *
  FROM t1 RIGHT OUTER JOIN t2
       ON t1.col1 = t2.col1;
```

例2) 次の例では、表t1に存在しないレコードを含めた表t2のレコードから、探索条件でt1.col3が2000以上であるレコードに絞り込みをして返却します。結合条件で優先的に絞り込んだ後、探索条件によって絞り込まれるため、返却されるレコードは1レコードのみとなります。

```
SELECT *
  FROM t1, t2
  WHERE t1.col1(+) = t2.col1
    AND t1.col3 >= 2000;
 col1 |    col2    | col3 | col1 |    col2
------+------------+------+------+------------
 1002 | BBBBB      | 2000 | 1002 | bbbbb
(1 row)
```

これは、次に示すFROM句の結合表(OUTER JOIN)の構文と同じ問合せです。

```
SELECT *
  FROM t1 RIGHT OUTER JOIN t2
       ON t1.col1 = t2.col1
  WHERE t1.col3 >= 2000;
```

9.3.2 DUAL表

DUAL表はシステムで用意された仮想表です。関数や演算式などの結果式を求めるテストなどのように実表にアクセスする必要性がないSQLを実行したい場合に使用します。

例

次の例では、システムの現在の日付を取得します。

```
SELECT  CURRENT_DATE  "date" FROM DUAL;
    date
------------
 2013-05-14
(1 row)
```

9.4 SQL関数のリファレンス

以下の、「SQL関数」をサポートしています。

- DECODE
- SUBSTR
- NVL

9.4.1 DECODE

機能

値を比較し別の値に変換します。

記述形式

DECODE(*expr*, *srch*, *result* [, *srch*, *result*]... [, *default*])

一般規則

- DECODEは、変換対象値式(*expr*)と各検索値(*srch*)の値を1つずつ比較し、変換対象値式と検索値とが一致する場合は対応する結果値(*result*)を返却します。変換対象値式と検索値のすべてが一致しないとき、省略値(*default*)が指定されている場合は省略値を返却し、省略値が指定されていない場合はNULL値を返却します。
- 検索値に同じ値が指定されている場合は、最初に出現した検索値に対応する結果値を返却します。
- 結果値と省略値において、使用できるデータ型は以下のとおりです。
  - CHAR
  - VARCHAR
  - NCHAR
  - NCHAR VARYING
  - TEXT
  - INTEGER
  - BIGINT
  - NUMERIC
  - DATE
  - TIME WITHOUT TIME ZONE
  - TIMESTAMP WITHOUT TIME ZONE

－ TIMESTAMP WITH TIME ZONE

- 変換対象値式と各検索値のデータ型はすべて同じデータ型を指定してください。ただし、検索値に定数を指定した場合、変換対象値式に対して変換可能なデータ型であれば、同じデータ型以外でも指定可能です。検索値に定数を指定した場合の指定可能なデータ型は、“A.3 暗黙の型変換”の“表A.1 定数を含む暗黙の型変換可能なデータ型の組合せ”を参照してください。
- 結果値と省略値がすべて定数の場合、結果値と省略値は以下のデータ型になります。

  － すべて文字列定数の場合、すべて文字列型になります。

  － 数定数が1つ以上ある場合、すべて数値型になります。

  － 日時/時刻型にキャストした定数が1つ以上ある場合、すべて日時/時刻型になります。
- 結果値と省略値に、定数と非定数が混在している場合、定数は非定数のデータ型に型変換します。変換可能なデータ型は、“A.3 暗黙の型変換”の“表A.1 定数を含む暗黙の型変換可能なデータ型の組合せ”を参照してください。
- 結果値と省略値のデータ型はすべて同じデータ型を指定してください。ただし、結果値と他の結果値または省略値のデータ型が変換可能なデータ型であれば、同じデータ型でなくても指定することができます。変換可能なデータ型は、以下のとおりです。

表9.3 DECODEにおける変換可能なデータ型の組合せ(サマリ)

		他の結果値または省略値		
		数値型	文字列型	日付/時刻型
任意の結果値	数値型	○	×	×
	文字列型	×	○	×
	日付/時刻型	×	×	△(注)

○:型変換可能
△:一部型変換可能
×:型変換不可能

注) 日付/時刻型に関して型変換が可能なデータ型について説明します。

表9.4 DECODEにおける結果値と省略値の変換可能なデータ型(日付/時刻型)

		他の結果値または省略値			
		DATE	TIME WITHOUT TIME ZONE	TIMESTAMP WITHOUT TIME ZONE	TIMESTAMP WITH TIME ZONE
任意の結果値	DATE	○	×	○	○
	TIME WITHOUT TIME ZONE	×	○	×	×
	TIMESTAMP WITHOUT TIME ZONE	○	×	○	○
	TIMESTAMP WITH TIME ZONE	○	×	○	○

○:型変換可能
×:型変換不可能

- 戻り値は、結果値や省略値のうち最大の長さや精度を持つデータ型になります。

例

次の例では、表t1のcol3の値を比較し別の値に変換します。col3の値が検索値1と一致する場合、結果値として「one」を返却します。col3の値が検索値1、2、3のいずれとも一致しない場合、省略値である「other number」を返却します。

```
SELECT col1,
       DECODE(col3, 1000, 'one',
                    2000, 'two',
                    3000, 'three',
                    'other number') "num-word"
       FROM t1;
col1  | num-word
------+----------
 1001 | one
 1002 | two
 1003 | three
(3 rows)
```

9.4.2 SUBSTR

機能

文字列の一部を抜き出します。

記述形式

SUBSTR(*str*, *startPos* [, *len*])

一般規則

- SUBSTRは、文字値式(*str*)の開始位置(*startPos*)の文字から文字列長(*len*)分の文字列を抜き出して返却します。
- 開始位置が正の場合、文字値式の先頭からが開始位置となります。
- 開始位置が0の場合、開始位置に1を指定したことと同じになります。
- 開始位置が負の場合、文字値式の終端からが開始位置となります。
- 文字列長を指定しない場合は、文字値式の終わりまでのすべての文字を返却します。文字列長が1より小さい場合、NULL値を返却します。
- 開始位置と文字列長のデータ型は、SMALLINT型またはINTEGER型を指定してください。定数を指定した場合の指定可能なデータ型は、"A.3 暗黙の型変換"の"表A.1 定数を含む暗黙の型変換可能なデータ型の組合せ"を参照してください。
- 戻り値のデータ型は、TEXT型です。

注意

- SUBSTRには、上記の仕様と同等の動作をする関数と、SUBSTRINGと同等の動作をする関数との2つが存在します。上記の仕様と同等の動作にするためには、search_pathの修正が必要です。
- search_pathはpostgresql.confで設定することを推奨します。この場合、インスタンス単位で有効になります。postgresql.confの設定方法については、"9.2.2 SUBSTRの注意事項"を参照してください。
- search_pathの設定は、ユーザー単位やデータベース単位でも設定することが可能です。設定例について以下に示します。

  ― ユーザー単位の設定例

  SQLコマンドを実行することで設定可能です。ユーザー名は例としてuser1にしています。

  ```
  ALTER USER user1 SET search_path = "$user", public, oracle, pg_catalog;
  ```

  ― データベース単位の設定例

  SQLコマンドを実行することで設定可能です。データベース名は例としてdb1にしています。

  ```
  ALTER DATABASE db1 SET search_path = "$user", public, oracle, pg_catalog;
  ```

  変更の際、「oracle」は、「pg_catalog」よりも前に指定する必要があります。

- 変更が未実施の場合、SUBSTRはSUBSTRINGと同等となります。

参照

ALTER USER、ALTER DATABASEの詳細については、“PostgreSQL Documentation”の“Reference” の“SQL Commands”を参照してください。

参考

SUBSTRINGの一般規則は以下になります。

- 開始位置が正、0、負に関わらず、文字値式の先頭からが開始位置になります。
- 文字列長を指定しない場合、文字値式の終わりまでのすべての文字を返却します。
- 返却する文字式が0以下、または指定した文字列長が1より小さい場合、空文字列を返却します。

参照

SUBSTRINGの詳細については、“PostgreSQL Documentation”の“The SQL Language”の“String Functions and Operators”を参照してください。

例

次の例では、「ABCDEFG」の一部の文字列を抜き出しています。

```
SELECT SUBSTR('ABCDEFG',3,4) "Substring" FROM DUAL;

 Substring
-----------
 CDEF
(1 row)

SELECT SUBSTR('ABCDEFG',-5,4) "Substring" FROM DUAL;

 Substring
-----------

(1 row)
```

9.4.3 NVL

機能

NULL値を変換します。

記述形式

NVL(*expr1*, *expr2*)

一般規則

- NVLは、NULL値を変換します。式1(*expr1*)がNULL値である場合、式2(*expr2*)を返却します。式1がNULL値でない場合、式1を返却します。

- 式1と式2は同じデータ型を指定してください。ただし、式2に定数を指定した場合、式1に対して変換可能なデータ型であれば、同じデータ型以外でも指定可能です。この時、式2は式1のデータ型に合わせて変換されるため、式1がNULL値である場合に返却される式2の値も、式1のデータ型に変換された値となります。
- 定数についての変換可能なデータ型は、“A.3 暗黙の型変換”の“表A.1 定数を含む暗黙の型変換可能なデータ型の組合せ”を参照してください。

例

次の例では、表t1のcol1の値がNULL値の場合には「IS NULL」を表示します。

```
SELECT col2, NVL(col1,'IS NULL') "nvl" FROM t1;
 col2 |   nvl
------+---------
 aaa  | IS NULL
(1 row)
```

9.5 パッケージのリファレンス

「パッケージ」とは、スキーマを使用して複数のファンクションを1つの機能としてまとめたものであり、PL/pgSQLから呼び出すことで利用できます。

以下の、「パッケージ」をサポートしています。

- DBMS_ALERT
- DBMS_ASSERTION
- DBMS_OUTPUT
- DBMS_PIPE
- DBMS_RANDOM
- DBMS_UTILITUY
- UTL_FILE
- DBMS_SQL

PL/pgSQLから各機能を呼び出すには、PERFORM文やSELECT文を使用し、パッケージ名で機能名を修飾してください。呼び出し形式の詳細については、各パッケージの機能ごとの説明を参照してください。

以降では、サポートしているパッケージのうち、DBMS_SQLについて解説します。他のパッケージについては、インストール先に格納されているREADMEを参照してください。

参照

DBMS_SQL以外のパッケージについては、下記ファイルを参照してください。

L

- **Linuxの場合**
  <Fujitsu Enterprise Postgresのインストール先ディレクトリ>/share/doc/extension/README.asciidoc

W

- Windows(R)の場合
  <Fujitsu Enterprise Postgresのインストール先ディレクトリ>¥doc¥extension¥README.asciidoc

9.5.1 DBMS_SQL

概要

PL/pgSQLから動的SQLを実行することができます。

機能

機能	説明
BIND_VARIABLE	SQL文内のホスト変数に値を設定します。
CLOSE_CURSOR	カーソルをクローズします。
COLUMN_VALUE	FETCH_ROWSを実行して取り出した選択リストの列の値を取得します。
DEFINE_COLUMN	値を取り出す列と格納先を定義します。
EXECUTE	SQL文を実行します。
FETCH_ROWS	指定したカーソルの次の行に位置付け、行から値を取り出します。
OPEN_CURSOR	新規カーソルをオープンします。
PARSE	SQL文を解析します。

注意

・ DBMS_SQLでは、利用者が動的SQLを実行する上でデータ型を意識する必要があり、使えるデータ型が限られます。以下のデータ型をサポートします。

- INTEGER
- DECIMAL
- NUMERIC
- REAL
- DOUBLE PRECISION
- CHAR(注1)
- VARCHAR(注1)
- NCHAR(注1)
- NCHAR VARYING(注1)
- TEXT
- DATE
- TIMESTAMP WITHOUT TIME ZONE
- TIMESTAMP WITH TIME ZONE
- INTERVAL(注2)
- SMALLINT
- BIGINT

注1)
CHAR型、VARCHAR型、NCHAR型、NCHAR VARYING型のホスト変数は、文字列関数の引数や戻り値と合わせるためにTEXT型として扱います。文字列関数の詳細については、“PostgreSQL Documentation”の“The SQL Language”の“String Functions and Operators”を参照してください。
Oracleデータベース互換機能のNVLおよびDECODEの引数に指定する場合は、引数間のデータ型が同じになるように、ホスト変数に対してCASTによりデータ型を変換してください。

例

探索条件にNVLを指定し、NVLの引数としてNCHAR型のホスト変数を期待する場合。

col1:NCHAR型の列
h1:NCHAR型のホスト変数

```
PERFORM DBMS_SQL.PARSE(cursor, 'SELECT col2 FROM t3 WHERE NVL(col1,CAST(:h1 AS NCHAR(3))) = N''あいう
''', 1);
PERFORM DBMS_SQL.BIND_VARIABLE(cursor, ':h1', N'あいう');
```

注2)
選択リストに指定したINTERVAL型の値をCOLUMN_VALUEで取得する場合、選択リストの時間隔修飾子と同じか時間隔修飾子を指定しないなど範囲の広いINTERVAL型の変数を指定してください。狭い範囲の時間隔修飾子の変数を指定すると、その時間隔修飾子の範囲の値を取得しますが、範囲外の値が切り捨てられたというエラーは発生しません。

例

選択リストにINTERVAL値を返す値式を設定し、その結果をCOLUMN_VALUEで受け取る場合。なお、例に記載したSQL文の演算結果はINTERVAL DAY TO SECONDの範囲の値を返します。

[悪い例]

変数(v_interval)がINTERVAL DAY TO HOURのため、MINUTE以降の値が切り捨てられます。

```
   v_interval     INTERVAL DAY TO HOUR;
・・・
   PERFORM DBMS_SQL.PARSE(cursor, 'SELECT CURRENT_TIMESTAMP - ''2010-01-01'' FROM DUAL', 1);
・・・
   SELECT value INTO v_interval FROM DBMS_SQL.COLUMN_VALUE(cursor, 1, v_interval);
   結果：1324 days 09:00:00
```

[良い例]

変数(v_interval)をINTERVALとすることで、値を正しく受け取ります。

```
   v_interval     INTERVAL;
・・・
   PERFORM DBMS_SQL.PARSE(cursor, 'SELECT CURRENT_TIMESTAMP - ''2010-01-01'' FROM DUAL', 1);
・・・
   SELECT value INTO v_interval FROM DBMS_SQL.COLUMN_VALUE(cursor, 1, v_interval);
   結果：1324 days 09:04:37.530623
```

記述形式

```
{ BIND_VARIABLE(cursor, varName, val [, len ])
| CLOSE_CURSOR(cursor)
| COLUMN_VALUE(cursor, colPos, varName)
| DEFINE_COLUMN(cursor, colPos, varName [, len ])
| EXECUTE(cursor)
| FETCH_ROWS(cursor)
| OPEN_CURSOR([parm1 ])
| PARSE(cursor, sqlStmt, parm1 [, parm2, parm3, parm4 ])
}
```

9.5.1.1 機能説明

DBMS_SQLの各機能について説明します。

BIND_VARIABLE

- BIND_VARIABLEは、SQL文内のホスト変数に値を設定します。
- カーソル番号(*cursor*)は処理対象のカーソル番号です。
- ホスト変数名(*varName*)はSQL文内のホスト変数の名前を文字列で指定します。

- 値式(*val*)はホスト変数に設定する値です。ホスト変数は、値式のデータ型になります。そして、SQL文中の指定場所に応じて暗黙的に型変換されます。暗黙の型変換については、“A.3 暗黙の型変換”を参照してください。
- 文字列長(*len*)は値式が文字列型の場合の文字数です。文字列長の指定がない場合は、文字列全体の長さとなります。
- SQL文のホスト変数は、ホスト変数を識別するために先頭にコロンを付ける必要があります。BIND_VARIABLEで指定するホスト変数名はコロンを付加してもしなくても構いません。以下にSQL文で指定するホスト変数名とBIND_VARIABLEで指定するホスト変数名の例を示します。

```
PERFORM DBMS_SQL.PARSE(cursor, 'SELECT emp_name FROM emp WHERE sal > :x', 1);
```

この例では、BIND_VARIABLEは次のようになります。

```
PERFORM DBMS_SQL.BIND_VARIABLE(cursor, ':x', 3500);
```

または、

```
PERFORM DBMS_SQL.BIND_VARIABLE(cursor, 'x', 3500);
```

- ホスト変数名の長さは、30バイト以下(コロンを除く)である必要があります。
- 設定する値のデータ型が文字列の場合、第4引数として列値の有効サイズを指定します。

例

設定する値のデータ型が文字列以外の場合

```
PERFORM DBMS_SQL.BIND_VARIABLE(cursor, ':NO', 1);
```

設定する値のデータ型が文字列の場合

```
PERFORM DBMS_SQL.BIND_VARIABLE(cursor, ':NAME', h_memid, 5);
```

CLOSE_CURSOR

- CLOSE_CURSORは、カーソルをクローズします。
- カーソル番号(*cursor*)は処理対象のカーソル番号です。
- 戻り値はNULL値です。

例

```
cursor := DBMS_SQL.CLOSE_CURSOR(cursor);
```

COLUMN_VALUE

- COLUMN_VALUEは、FETCH_ROWSを実行して取り出した選択リストの列の値を取得します。
- カーソル番号(*cursor*)は処理対象のカーソル番号です。
- 列位置(*colPos*)はSELECT文の選択リストの列の位置です。最初の列の位置は1です。
- 変数名(*varName*)は、格納先の変数名を指定します。
- 取り出した情報は、SELECT文でvalue, column_error, actual_lengthの列の値として取得します。

- valueは列位置で指定した列の値を返却します。変数名のデータ型は列のデータ型に合わせてください。PARSEに指定したSELECT文の選択リストの列がDBMS_SQLに対応していないデータ型の場合、CASTを使って対応するデータ型に型変換してください。
- column_errorはNUMERIC型です。列の値が、valueに正しく設定できなかった場合に、0以外の値を返却します。
  22001：取り出した文字列が切り捨てられている
  22002：取り出した値の内容がNULL値
- actual_lengthはINTEGER型です。取り出した値が文字列型の場合は、文字数を返却します。切り捨てられている場合、切り捨てられる前の文字数を返却します。文字列型でない場合は、バイト数を返却します。

例

列の値、エラーコード、列値の実際の長さを取得する場合

```
SELECT value, column_error, actual_length INTO v_memid, v_col_err, v_act_len FROM
DBMS_SQL.COLUMN_VALUE(cursor, 1, v_memid);
```

列の値だけを取得する場合

```
SELECT value INTO v_memid FROM DBMS_SQL.COLUMN_VALUE(cursor, 1, v_memid);
```

DEFINE_COLUMN

- DEFINE_COLUMNは、値を取り出す列と格納先を定義します。
- カーソル番号(*cursor*)は処理対象のカーソル番号です。
- 列位置(*colPos*)はSELECT文の選択リストの列の位置です。最初の列の位置は1です。
- 変数名(*varName*)は、格納先を指定します。格納先のデータ型は、値を取り出す列のデータ型に合わせてください。PARSEに指定したSELECT文の選択リストの列がDBMS_SQLに対応していないデータ型の場合、CASTを使って対応するデータ型に型変換してください。
- 文字列長(*len*)は文字列型の列に対する列値の最大の文字数です。
- 列値のデータ型が文字列の場合、第4引数として列値の有効サイズを指定します。

例

列値のデータ型が文字列以外の場合

```
PERFORM DBMS_SQL.DEFINE_COLUMN(cursor, 1, v_memid);
```

列値のデータ型が文字列の場合

```
PERFORM DBMS_SQL.DEFINE_COLUMN(cursor, 1, v_memid, 10);
```

EXECUTE

- EXECUTEは、SQL文を実行します。
- カーソル番号(*cursor*)は処理対象のカーソル番号です。
- 戻り値は、INTEGER型で、INSERT文、UPDATE文、DELETE文の場合のみ有効で、処理した行数です。それ以外の場合は無効です。

例

```
ret := DBMS_SQL.EXECUTE(cursor);
```

FETCH_ROWS

- FETCH_ROWSは、次の行に位置付け、行から値を取り出します。
- カーソル番号(*cursor*)は処理対象のカーソル番号です。
- 戻り値は、INTEGER型で、取り出した行数です。すべて取り出した場合は0を返却します。
- 取り出した情報はCOLUMN_VALUEで取得します。

例

```
LOOP
    IF DBMS_SQL.FETCH_ROWS(cursor) = 0 THEN
        EXIT;
    END IF;
.
.
.
END LOOP;
```

OPEN_CURSOR

- OPEN_CURSORは、新規のカーソルをオープンします。
- パラメータ1(*parm1*)はOracleデータベースとの互換のためのパラメータであり、Fujitsu Enterprise Postgresでは目的を持たないパラメータです。INTEGER型が指定可能で、指定値は無視されます。指定値に意味はありませんが、指定する場合は1を指定してください。なお、Oracleデータベースから移行する場合、指定値を変更する必要はありません。
- 不要になったカーソルはCLOSE_CURSORを実行してクローズしてください。
- 戻り値は、INTEGER型で、カーソル番号です。

例

```
cursor := DBMS_SQL.OPEN_CURSOR();
```

PARSE

- PARSEは、動的SQL文の解析を行います。
- カーソル番号(*cursor*)は処理対象のカーソル番号です。
- SQL文(*sqlStmt*)は解析するSQL文です。
- パラメータ1(*parm1*)、パラメータ2(*parm2*)、パラメータ3(*parm3*)およびパラメータ4(*parm4*)は、Oracleデータベースとの互換のためのパラメータであり、Fujitsu Enterprise Postgresでは目的を持たないパラメータです。指定値は無視されます。指定値に意味はありませんが、指定する場合は以下の値を指定してください。
  - パラメータ1はINTEGER型で、1を指定してください。
  - パラメータ2およびパラメータ3はTEXT型で、指定する場合はNULLを指定してください。
  - パラメータ4はBOOLEAN型で、指定する場合はTRUEを指定してください。

なお、Oracleデータベースから移行する場合、パラメータ2、パラメータ3およびパラメータ4は、指定値を変更する必要はありません。

- SQL文のホスト変数は、先頭にコロンを付けます。
- DDL文はPARSEを発行した時点で実行されます。DDL文のEXECUTEは不要です。
- オープンしているカーソルに対して再度PARSEを呼び出した場合は、カーソル内のデータ領域の内容をリセットし、新たにSQL文の解析を行います。

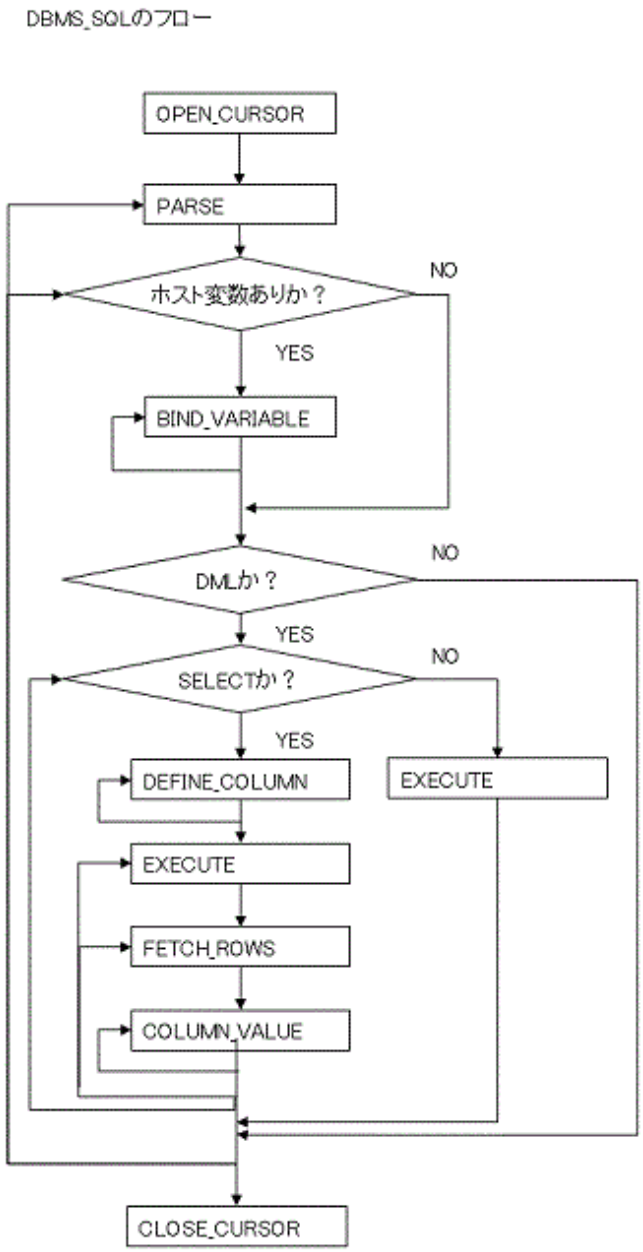
例

```
PERFORM DBMS_SQL.PARSE(cursor, 'SELECT memid, memnm FROM member WHERE memid = :NO', 1);
```

9.5.1.2 使用例

DBMS_SQLの各機能のフローと使用例を示します。

DBMS_SQLのフロー



使用例

```
CREATE FUNCTION smp_00()
RETURNS INTEGER
AS $$
DECLARE
    str_sql     VARCHAR(255);
    cursor      INTEGER;
    h_smpid     INTEGER;
    v_smpid     INTEGER;
    v_smpnm     VARCHAR(20);
    v_smpage    INTEGER;
    errcd       INTEGER;
    length      INTEGER;
```

```
    ret         INTEGER;
BEGIN
    str_sql     := 'SELECT smpid, smpnm, smpage FROM smp_tbl WHERE smpid < :H_SMPID ORDER BY smpid';
    h_smpid     := 3;
    v_smpid     := 0;
    v_smpnm     := '';
    v_smpage    := 0;

    cursor := DBMS_SQL.OPEN_CURSOR();

    PERFORM DBMS_SQL.PARSE(cursor, str_sql, 1);

    PERFORM DBMS_SQL.BIND_VARIABLE(cursor, ':H_SMPID', h_smpid);

    PERFORM DBMS_SQL.DEFINE_COLUMN(cursor, 1, v_smpid);
    PERFORM DBMS_SQL.DEFINE_COLUMN(cursor, 2, v_smpnm, 10);
    PERFORM DBMS_SQL.DEFINE_COLUMN(cursor, 3, v_smpage);

    ret := DBMS_SQL.EXECUTE(cursor);
    loop
        if DBMS_SQL.FETCH_ROWS(cursor) = 0 then
            EXIT;
        end if;

       SELECT value,column_error,actual_length INTO v_smpid,errcd,length FROM DBMS_SQL.COLUMN_VALUE(cursor,
1, v_smpid);
        RAISE NOTICE '-------------------------------------------------------';
        RAISE NOTICE '-------------------------------------------------------';
        RAISE NOTICE 'smpid       = %', v_smpid;
        RAISE NOTICE 'errcd       = %', errcd;
        RAISE NOTICE 'length      = %', length;

       SELECT value,column_error,actual_length INTO v_smpnm,errcd,length FROM DBMS_SQL.COLUMN_VALUE(cursor,
2, v_smpnm);
        RAISE NOTICE '-------------------------------------------------------';
        RAISE NOTICE 'smpnm       = %', v_smpnm;
        RAISE NOTICE 'errcd       = %', errcd;
        RAISE NOTICE 'length      = %', length;

      select value,column_error,actual_length INTO v_smpage,errcd,length FROM DBMS_SQL.COLUMN_VALUE(cursor,
3, v_smpage);
        RAISE NOTICE '-------------------------------------------------------';
        RAISE NOTICE 'smpage      = %', v_smpage;
        RAISE NOTICE 'errcd       = %', errcd;
        RAISE NOTICE 'length      = %', length;
        RAISE NOTICE '';
    end loop;

    cursor := DBMS_SQL.CLOSE_CURSOR(cursor);
    RETURN 0;
END;
$$ LANGUAGE plpgsql;
```

第10章　アプリケーションの接続先切り替え機能

アプリケーションの接続先切り替え機能とは、冗長化して構成された複数のサーバに対して、接続対象のサーバがどのサーバであるかを意識せずに接続することができるようにする機能です。

アプリケーションの接続情報に、接続サーバとしてプライマリサーバとスタンバイサーバを指定して使用します。プライマリサーバよりもスタンバイサーバを優先させて接続先サーバとすることもできます。

アプリケーション接続先の切り替えが発生した場合は、明示的にコネクションを切断し、コネクションの再接続またはアプリケーションを再実行してください。切り替えの確認方法については、各クライアントインタフェースの“アプリケーションの接続先切り替えが発生した場合のエラーと対処”を参照してください。

10.1 アプリケーションの接続先切り替え機能の接続情報

アプリケーションの接続先切り替え機能を利用する場合には、データベース接続時に、以下の情報を設定してください。

IPアドレスまたはホスト名

データベース多重化システムを構成するIPアドレスまたはホスト名を指定します。

ポート番号

各データベースサーバがアプリケーションの接続をlistenしているポート番号です。
各クライアントインタフェースでは、複数のポート番号を指定することができますが、例えば以下の形式でポート番号を1つだけ指定した場合、クライアントインタフェースよってポート番号の指定値が異なります。

host1,host2:port2

JDBCドライバまたは.NET Data Provider

ポート番号を1つだけ指定すると、host1:27500(省略値)とhost2:port2が指定されたとみなされます。ポート番号は、すべて省略するか、サーバの個数分指定してください

その他のドライバ

ポート番号を1つだけ指定すると、すべてのポートで同じポート番号が指定されたとみなされます。

ターゲットサーバ

指定した接続先サーバの情報の中から、アプリケーションが接続するサーバの選択順を指定します。ターゲットサーバに指定する値は、それぞれ以下の意味です。省略した場合には、“any”になります。

プライマリサーバ

指定した“IPアドレスまたはホスト名”の中から、プライマリサーバを選択して接続します。更新を伴うアプリケーションや、REINDEX や VACUUM といった管理作業のように、プライマリサーバでのみ実行可能な作業を行う場合に指定します。

スタンバイサーバ

指定した“IPアドレスまたはホスト名”の中から、スタンバイサーバを優先的に選択して接続します。スタンバイサーバでは更新を行うことはできません。接続先がスタンバイサーバではない場合、指定された＜接続先サーバタイプ＞が見つからないというメッセージが出力されます。

プライマリサーバの優先

指定した“IPアドレスまたはホスト名”の中から、プライマリサーバを優先的に選択して接続します。プライマリサーバが存在しない場合には、スタンバイサーバに接続します。

スタンバイサーバの優先

指定した“IPアドレスまたはホスト名”の中から、スタンバイサーバを優先的に選択して接続します。スタンバイサーバが存在しない場合には、プライマリサーバに接続します。

任意

この指定方法は、データベース多重化システムでは推奨しません。指定した“IPアドレスまたはホスト名”の中から、指定された順番に接続先を選択していきますが、最初に接続に成功したサーバがスタンバイサーバの場合には、常に書き込み操作が失敗するためです。

各ドライバごとのサーバの選択順の設定値は、以下のとおりです。

サーバの選択順	JDBCドライバ	.NET Data Provider	他のドライバ
プライマリサーバ	“primary”(注1)	“read-write”(注1) “primary”(注2)	“read-write”(注1) “primary”(注2)
スタンバイサーバ	“secondary”(注2)	“standby” “read-only”(注2)	“standby” “read-only”(注2)
プライマリサーバの優先	“preferPrimary”(注3)	“prefer-primary”	－
スタンバイサーバの優先	“preferSecondary”(注2)	“prefer-standby”	“prefer-standby”(注4)
任意	“any”	“any”	“any”

注1: デフォルトのトランザクションモードが読み取り専用のプライマリサーバは選択されません。

注2: デフォルトのトランザクションモードが読み取り専用のプライマリサーバも選択されます。

注3: デフォルトのトランザクションモードが読み書き可能なプライマリサーバが優先されます。

注4: Fujitsu Enterprise Postgres 13以前のバージョンからの互換扱いとして“prefer-standby”の代わりに“prefer-read”も指定可能ですが、推奨できません。

SSLのサーバ証明書のCommon Name

多重化システムの各サーバで、各サーバごとに同じサーバ証明書を作成し、SSL認証を行う場合は、サーバ証明書のCommon Nameを本パラメータに指定して接続してください。これによって、多重化システムを構成する複数のサーバ名を意識することなく、CommonNameを使用したSSL認証を行うことができます。

10.2 アプリケーションの接続先切り替え機能を利用する

アプリケーションの接続先切り替え機能での接続先サーバの設定方法を説明します。

各クライアントインタフェースの接続情報として記載するパラメータは、アプリケーションの接続先切り替え機能に特化した部分のみ説明しています。それ以外のパラメータについては、各クライアントインタフェースの“セットアップ”および“データベースへの接続”を参照してください。

10.2.1 JDBCドライバを利用する場合

DriverManagerクラスの接続文字列またはデータソースに、以下の情報を設定します。

表10.1 設定する情報

引数	説明
host1 host2	IPアドレスまたはホスト名を指定します。 IPアドレスまたはホスト名は省略できます。省略した場合には、localhostになります。
port1 port2	接続先のポート番号を指定します。 ポート番号は省略できます。省略した場合には、27500になります。
database_name	データベース名を指定します。
targetServerType	アプリケーションが接続するサーバの選択順を指定します。 詳細は“ターゲットサーバ”を参照してください。

引数	説明
sslmode	通信を暗号化する場合に指定してください。デフォルトは無効に設定されています。 sslmodeの設定値は以下のとおりです。 disable:非SSLで接続します。 require:必ずSSLで接続します。 verify-ca:SSLで接続し、信頼できるCAから発行された証明書を使用します。(注) verify-full:SSLで接続し、信頼できるCAから発行された証明書を使用してサーバのホスト名が証明書と一致するかを検証します。(注)
sslservercertcn	本パラメータはSSL認証(sslmode=verify-full)を行う場合のみ有効となります。 サーバ証明書のCN(Common Name)を指定します。省略した場合は、nullとなり、hostに指定されたホスト名を使用して、サーバ証明書のCN(Common Name)を認証します。

注: “verify-ca”または“verify-full”を指定する場合、CA証明書ファイルを接続文字sslrootcertで指定できます。

Driver Managerを使用する場合

DriverManagerクラスのAPIに、以下のURIを指定します。

```
jdbc:postgresql://[host1][:port1],[host2][:port2]/database_name[?targetServerType={primary | secondary | preferPrimary | preferSecondary | any}][&sslmode=verify-full&sslrootcert=CA証明書ファイル
&sslservercertcn=<対象とするサーバ証明書のCN(Common Name)>]
```

- ターゲットサーバを省略した場合は、anyとなります。
- IPV6を使用する場合、ホストは[host]の形式で指定してください。

  [指定例]

```
jdbc:postgresql://[2001:Db8::1234]:27500,192.168.1.1:27500/database_name
```

データソースを使用する場合

データソースのプロパティに、以下の形式で指定します。

```
source.setServerName("[host1][:port1],[host2][:port2]");
source.setTargetServerType("primary");
source.setSslmode("verify-full");
source.setSslrootcert("CA証明書ファイル");
source.setSslservercertcn("<対象とするサーバ証明書のCN(Common Name)>");
```

- IPアドレスおよびホスト名を省略した場合は、localhostとなります。
- ポート番号を省略した場合は、portNumberプロパティに指定された値が使用されます。また、portNumberプロパティが省略されている場合は、27500となります。
- ターゲットサーバを省略した場合は、anyとなります。
- IPV6を使用する場合、ホストは[host]の形式で指定してください。

  [指定例]

```
source.setServerName("[2001:Db8::1234]:27500,192.168.1.1:27500");
```

注意

接続パラメータloginTimeoutを利用する場合には、指定したすべてのホストへの接続の試みにかかった時間に対して適用されます。

10.2.2 ODBCドライバを利用する場合

接続文字列またはデータソースに、以下の情報を設定します。

表10.2 設定する情報

パラメータ	説明
Servername	カンマ区切りでIPアドレス1およびIPアドレス2、またはホスト名を指定します。 ODBCの規約に基づいて、カンマ区切りを含む文字列全体を{}で囲むことを推奨します。 指定形式:{host1,host2}
Port	カンマ区切りで接続先のポート番号を指定します。 ODBCの規約に基づいて、カンマ区切りを含む文字列全体を{}で囲むことを推奨します。 指定形式: {port1,port2} Servernameのn番目に指定したIPアドレスまたはホストに対応するポート番号は、Portのn番目に指定してください。 ポート番号は省略できます。省略した場合には、27500になります。 Servernameをn個指定し、Portをm個指定した場合は、エラーとなります。ただし、m=nまたはmが1の場合は、エラーとなりません。また、Portを1つだけ指定した場合は、Servernameに指定したすべてのIPアドレスまたはホスト名に対して、同じポート番号が適用されます。
target_session_attrs	アプリケーションが接続するサーバの選択順を指定します。 詳細は“ターゲットサーバ”を参照してください。
SSLMode	通信を暗号化する場合に指定してください。デフォルトは無効に設定されています。 SSLModeの設定値は以下のとおりです。 disable:非SSLで接続します。 allow:非SSLで接続し、失敗したらSSLで接続します。 prefer:SSLで接続し、失敗したら非SSLで接続します。 require:必ずSSLで接続します。 verify-ca:SSLで接続し、信頼できるCAから発行された証明書を使用します。(注) verify-full:SSLで接続し、信頼できるCAから発行された証明書を使用してサーバのホスト名が証明書と一致するかを検証します。(注)
SSLServerCertCN	本パラメータはSSL認証(SSLMode=verify-full)を行う場合のみ、有効となります。 サーバ証明書のCN(Common Name)を指定します。省略した場合は、nullとなり、Servernameに指定されたホスト名を使用してサーバ証明書のCN(Common Name)を認証します。

注:“verify-ca”または“verify-full”を指定する場合、CA証明書ファイルをOSのシステム環境変数PGSSLROOTCERTで以下のように指定してください。

例)
変数名: PGSSLROOTCERT
変数値: CA証明書ファイル

接続文字列を指定する場合

以下の接続文字列を指定します。

```
...;Servername={host1,host2};Port={port1,port2};[ target_session_attrs={any | read-write | read-only | primary | standby | prefer-standby}];[ SSLMode=verify-full;SSLServerCertCN=<対象とするサーバ証明書のCN(Common Name)>]...
```

－ IPV6を使用する場合、ホストはhostの形式で指定してください。

[指定例]

```
Servername={2001:Db8::1234,192.168.1.1};Port={27500,27500};
```

データソースを使用する場合

以下の形式で指定します。

```
Servername={host1,host2}
Port={port1,port2}
target_session_attrs={any | read-write | read-only | primary | standby | prefer-standby}
SSLMode=verify-full
SSLServerCertCN=<対象とするサーバ証明書のCN(Common Name)>
```

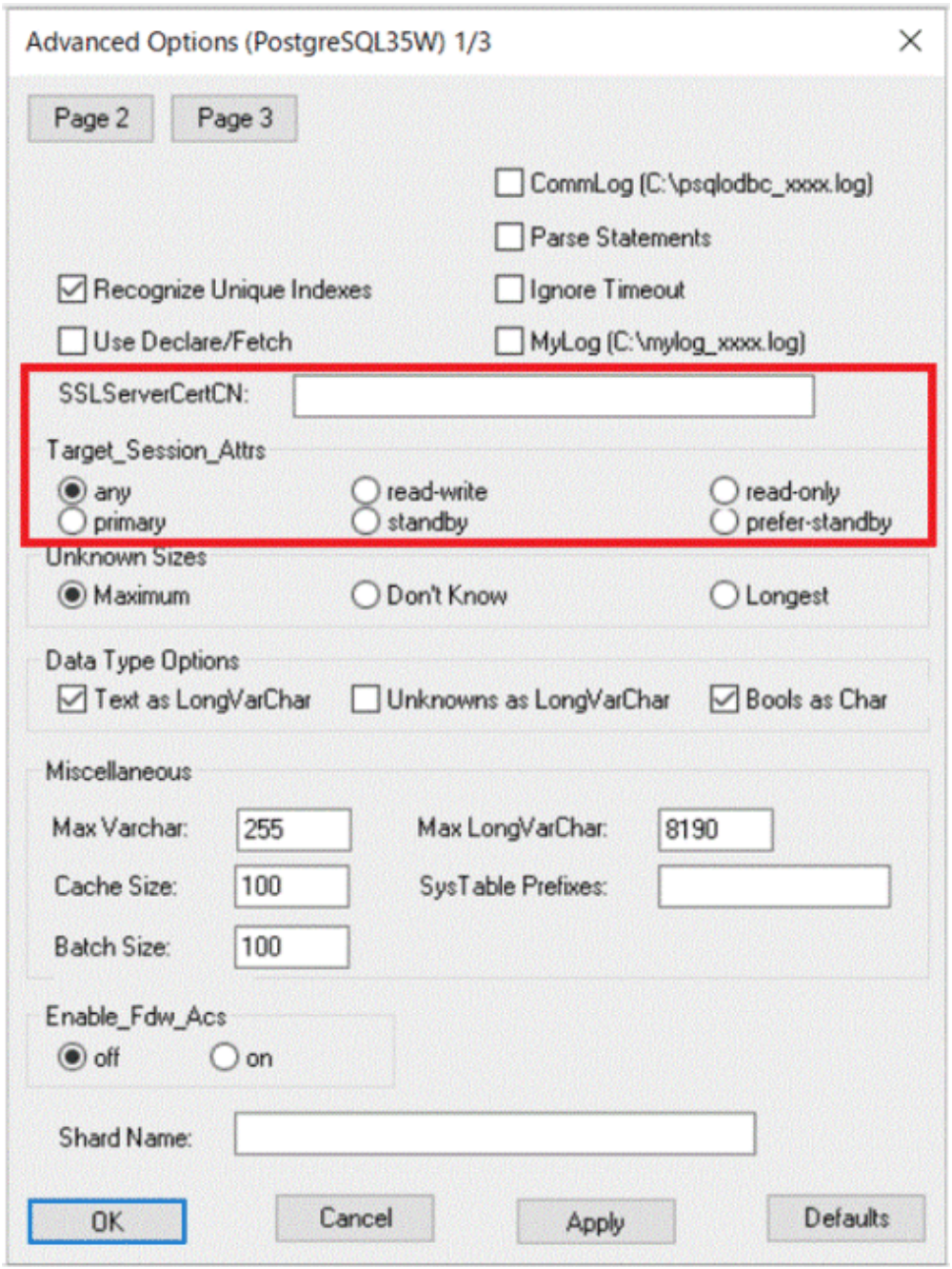
－ IPV6を使用する場合、ホストはhostの形式で指定してください。

[指定例]

```
Servername={2001:Db8::1234,192.168.1.1}
```

ODBCデータソースアドミニストレータからのデータソース登録

ODBCデータソースアドミニストレータでは、以下のように指定します。



注意

接続パラメータlogin_timeoutを利用する場合には、指定した各ホストへの接続に対してlogin_timeoutが適用されます。多重化されているデータベースサーバの両方がダウンしている場合、接続がタイムアウトするまでには、login_timeoutの2倍の時間がかかります。

10.2.3 .NET Data Providerを利用する場合

NpgsqlConnectionの接続文字列またはデータソースに、以下の情報を設定します。

表10.3 設定する情報

引数	説明
host1 host2	IPアドレスまたはホスト名を指定します。
port1 port2	接続先のポート番号を指定します。
TargetSessionAttributes	アプリケーションが接続するサーバの選択順を指定します。 詳細は“ターゲットサーバ”を参照してください。

接続文字列を指定する場合

以下の接続文字列を指定します。

```
host1[:port1],host2[:port2];[Target Session Attributes={any | primary | standby | prefer-primary | prefer-standby | read-write | read-only}];
```

- ポート番号を省略した場合は、接続文字列のPortキーの指定値が使用されます。Portキーについては、“4.3.3 接続文字列”を参照してください。
- ターゲットサーバを省略した場合は、anyとなります。
- IPV6を使用する場合、ホストは[host]の形式で指定してください。

  [指定例]

  ```
  host=[2001:Db8::1234]:27500,192.168.1.1:27500;
  ```

NpgsqlConnectionStringBuilderのプロパティに指定する場合

データソースのHostプロパティに、以下の形式で指定します。

```
host1[:port1],host2[:port2]
```

- ポート番号を省略した場合は、Portプロパティに指定された値が使用されます。また、Portプロパティが省略されている場合は、27500となります。

データソースのTargetSessionAttributesプロパティに、以下の形式で指定します。

```
any | primary | standby | prefer-primary | prefer-standby | read-write | read-only
```

- ターゲットサーバを省略した場合は、anyとなります。

注意

接続パラメータTimeoutを利用する場合には、指定した各ホストへの接続に対してTimeoutが適用されます。多重化されているデータベースサーバの両方がダウンしている場合、接続がタイムアウトするまでには、Timeoutの2倍の時間がかかります。

10.2.4 接続サービスファイルを利用する場合

接続パラメータに、以下の情報を設定します。

表10.4 設定する情報

パラメータ	説明
host	カンマ区切りでホスト名を指定します。
hostaddr	カンマ区切りでIPアドレス1およびIPアドレス2を指定します。

パラメータ	説明
port	カンマ区切りで接続先のポート番号を指定します。 hostまたはhostaddrのn番目に指定したサーバのポート番号は、portのn番目に指定してください。 ポート番号は省略できます。省略した場合には、27500になります。 hostまたはhostaddrをn個指定し、portをm個指定した場合は、エラーとなります。ただし、m=nまたはmが1の場合は、エラーとなりません。また、portを1つだけ指定した場合は、hostまたはhostaddrに指定したすべてのIPアドレスまたはホスト名に対して、同じポート番号が適用されます。
target_session_attrs	アプリケーションが接続するサーバの選択順を指定します。 詳細は“ターゲットサーバ”を参照してください。
sslmode	通信を暗号化する場合に指定してください。デフォルトは無効に設定されています。 sslmodeの設定値は以下のとおりです。 disable：非SSLで接続します。 allow：非SSLで接続し、失敗したらSSLで接続します。 prefer：SSLで接続し、失敗したら非SSLで接続します。 require：必ずSSLで接続します。 verify-ca：SSLで接続し、信頼できるCAから発行された証明書を使用します。(注) verify-full：SSLで接続し、信頼できるCAから発行された証明書を使用してサーバのホスト名が証明書と一致するかを検証します。(注)
sslservercertcn	本パラメータはSSL認証(sslmode=verify-full)を行う場合のみ、有効となります。 サーバ証明書のCN(Common Name)を指定します。省略した場合は、nullとなり、hostに指定されたホスト名を使用してサーバ証明書のCN(Common Name)を認証します。

注：“verify-ca”または“verify-full”を指定する場合、CA証明書ファイルをOSのシステム環境変数PGSSLROOTCERT(接続パラメータsslrootcert)で以下のように指定してください。

例)
変数名: PGSSLROOTCERT
変数値: CA証明書ファイル

注意

接続パラメータconnect_timeoutを利用する場合には、指定した各ホストへの接続に対してconnect_timeoutが適用されます。多重化されているデータベースサーバの両方がダウンしている場合、接続がタイムアウトするまでには、connect_timeoutの2倍の時間がかかります。

ポイント

C言語用ライブラリ、埋め込みSQL、psqlコマンド(接続先を指定するその他のクライアントコマンドも含みます)を利用する場合には、接続サービスファイルを用いて接続先を指定することを推奨します。

接続サービスファイルには、接続先情報やコネクションに対して設定する各種のチューニング情報を1セットとして名前(サービス名)を定義します。データベース接続時には、接続サービスファイルに定義されたサービス名を用いることで、接続情報の変更によるアプリケーションの修正が不要になります。

10.2.5 C言語用ライブラリ(libpq)を利用する場合

接続サービスファイルの利用を推奨します。“10.2.4 接続サービスファイルを利用する場合”を参照してください。

接続サービスファイルを利用しない場合は、データベース接続制御関数(PQconnectdbParams、PQconnectdbなど)または環境変数に、以下の情報を設定します。

表10.5 設定する情報

パラメータ(環境変数名)	説明
host(PGHOST)	カンマ区切りでホスト名を指定します。
hostaddr(PGHOSTADDR)	カンマ区切りでIPアドレス1およびIPアドレス2を指定します。
port(PGPORT)	カンマ区切りで接続先のポート番号を指定します。 hostまたはhostaddrのn番目に指定したサーバのポート番号は、portのn番目に指定してください。 ポート番号は省略できます。省略した場合には、27500になります。 hostまたはhostaddrをn個指定し、portをm個指定した場合は、エラーとなります。ただし、m=nまたはmが1の場合は、エラーとなりません。また、portを1つだけ指定した場合は、hostまたはhostaddrに指定したすべてのIPアドレスまたはホスト名に対して、同じポート番号が適用されます。
target_session_attrs(PGTARGETSESSIONATTRS)	アプリケーションが接続するサーバの選択順を指定します。 詳細は“ターゲットサーバ”を参照してください。
sslmode(PGSSLMODE)	通信を暗号化する場合に指定してください。デフォルトは無効に設定されています。 sslmodeの設定値は以下のとおりです。 disable：非SSLで接続します。 allow：非SSLで接続し、失敗したらSSLで接続します。 prefer：SSLで接続し、失敗したら非SSLで接続します。 require：必ずSSLで接続します。 verify-ca：SSLで接続し、信頼できるCAから発行された証明書を使用します。(注) verify-full：SSLで接続し、信頼できるCAから発行された証明書を使用してサーバのホスト名が証明書と一致するかを検証します。(注)
sslservercertcn(PGXSSLSERVERCERTCN)	本パラメータはSSL認証(sslmode=verify-full)を行う場合のみ、有効となります。 サーバ証明書のCN(Common Name)を指定します。省略した場合は、nullとなり、hostに指定されたホスト名を使用してサーバ証明書のCN(Common Name)を認証します。

注：“verify-ca”または“verify-full”を指定する場合、CA証明書ファイルをOSのシステム環境変数PGSSLROOTCERT(接続パラメータsslrootcert)で以下のように指定してください。

例)
変数名: PGSSLROOTCERT
変数値: CA証明書ファイル

URIを使用する場合

```
postgresql://host1[:port1],host2[:port2][,...]/database_name
[?target_session_attrs={read-write | read-only | primary | standby | prefer-standby | any }]
```

ー IPV6を使用する場合、ホストは[host]の形式で指定してください。

[指定例]

```
postgresql://postgres@[2001:Db8::1234]:27500,192.168.1.1:27500/database_name
```

key-valueを使用する場合

```
host=host1[,host2] port=port1[,port2] user=user1 password=pwd1 dbname=mydb [target_session_attrs={read-write | read-only | primary | standby | prefer-standby | any }]
```

ー IPV6を使用する場合、ホストはhostの形式で指定してください。

[指定例]

```
host=2001:Db8::1234,192.168.1.1 port=27500,27500
```

注意

接続パラメータconnect_timeoutを利用する場合には、指定した各ホストへの接続に対してconnect_timeoutが適用されます。多重化されているデータベースサーバの両方がダウンしている場合、接続がタイムアウトするまでには、connect_timeoutの2倍の時間がかかります。

参考

パスワードファイル(.pgpass) を使用する場合は、各サーバに合致するエントリを記述してください。

- 例1

```
host1:port1:dbname:user:password
host2:port2:dbname:user:password
```

- 例2

```
*:port:dbname:user:password
```

10.2.6 埋め込みSQLを利用する場合

接続サービスファイルの利用を推奨します。“10.2.4 接続サービスファイルを利用する場合”を参照してください。

ポイント

接続サービスファイルを利用するには、以下のいずれかの方法があります。

- 文字列リテラルまたはホスト変数を使用して、以下のように記述する方法

  tcp:postgresql://?service=my_service

- 環境変数PGSERVICEにサービス名を設定し、かつCONNECT TO DEFAULTを用いる方法

接続サービスファイルを利用しない場合は、以下のSQL文のtargetに、リテラルまたは変数を用いて接続先サーバの情報を指定します。

```
EXEC SQL CONNECT TO target [AS connection-name] [USER user-name];
```

指定方法

```
dbname@host1,host2[:[port1][,port2]]
tcp:postgresql://host1,host2[:[port1][,port2]] [/dbname] [?target_session_attrs={read-write | read-only |
```

```
primary | standby | prefer-standby | any }][&sslmode=verify-full&sslservercertcn=<対象とするサーバ証明書のCN(Common Name)>]
```

ー 上記の形式をリテラルや変数を使わずに直接指定することはできません。

表10.6 設定する情報

引数	説明
host1 host2	IPアドレスまたはホスト名を指定します。IPV6形式のIPアドレスは、指定できません。
port1 port2	カンマ区切りで接続先のポート番号を指定します。 ポート番号は省略できます。省略した場合には、27500になります。
dbname	データベース名を指定します。
target_session_attrs	アプリケーションが接続するサーバの選択順を指定します。 詳細は“ターゲットサーバ”を参照してください。
sslmode	通信を暗号化する場合に指定してください。デフォルトは無効に設定されています。 sslmodeの設定値は以下のとおりです。 disable:非SSLで接続します。 allow:非SSLで接続し、失敗したらSSLで接続します。 prefer:SSLで接続し、失敗したら非SSLで接続します。 require:必ずSSLで接続します。 verify-ca:SSLで接続し、信頼できるCAから発行された証明書を使用します。(注) verify-full:SSLで接続し、信頼できるCAから発行された証明書を使用してサーバのホスト名が証明書と一致するかを検証します。(注)
sslservercertcn	本パラメータはSSL認証(sslmode=verify-full)を行う場合のみ、有効となります。 サーバ証明書のCN(Common Name)を指定します。省略した場合は、nullとなり、hostに指定されたホスト名を使用してサーバ証明書のCN(Common Name)を認証します。

注:“verify-ca”または“verify-full”を指定する場合、CA証明書ファイルをOSのシステム環境変数PGSSLROOTCERT(接続パラメータsslrootcert)で以下のように指定してください。

例)
変数名: PGSSLROOTCERT
変数値: CA証明書ファイル

ポイント

環境変数での設定することもできます。環境変数については、“10.2.5 C言語用ライブラリ(libpq)を利用する場合”を参照してください。

注意

接続パラメータconnect_timeoutを利用する場合には、指定した各ホストへの接続に対してconnect_timeoutが適用されます。多重化されているデータベースサーバの両方がダウンしている場合、接続がタイムアウトするまでには、connect_timeoutの2倍の時間がかかります。

10.2.7 psqlコマンドを利用する場合

接続サービスファイルの利用を推奨します。“10.2.4 接続サービスファイルを利用する場合”を参照してください。

接続サービスファイルを利用しない場合は、psqlコマンドのオプション/環境変数に、以下の情報を指定します。

表10.7 設定する情報

オプション(環境変数)	説明
-h/--host(PGHOST/ PGHOSTADDR)	カンマ区切りでIPアドレス1およびIPアドレス2、またはホスト名を指定します。 環境変数PGHOSTまたはPGHOSTADDR に指定することも可能です。
-p/--port(PGPORT)	カンマ区切りで接続先のポート番号を指定します。 環境変数PGPORT に指定することも可能です。 -hオプションのn番目に指定したIPアドレスに対応するポート番号は、-pオプションのn番目に指定してください。 ポート番号は省略できます。省略した場合には、27500となります。 -hオプションをn個指定し、-pオプションをm個指定した場合は、エラーとなります。ただし、m=nまたはmが1の場合は、エラーとなりません。また、-pオプションを1つだけ指定した場合は、-hオプションに指定したすべてのIPアドレスまたはホスト名に対して、同じポート番号が適用されます。
(PGTARGETSESSIONATTRS)	アプリケーションが接続するサーバの選択順を指定します。 詳細は“ターゲットサーバ”を参照してください。
(PGSSLMODE)	通信を暗号化する場合に指定してください。デフォルトは無効に設定されています。 PGSSLMODEの設定値は以下のとおりです。 disable：非SSLで接続します。 allow：非SSLで接続し、失敗したらSSLで接続します。 prefer：SSLで接続し、失敗したら非SSLで接続します。 require：必ずSSLで接続します。 verify-ca：SSLで接続し、信頼できるCAから発行された証明書を使用します。(注) verify-full：SSLで接続し、信頼できるCAから発行された証明書を使用してサーバのホスト名が証明書と一致するかを検証します。(注)
(PGXSSLSERVERCERTCN)	本環境変数はSSL認証（PGSSLMODE=verify-full）を行う場合のみ、有効となります。 サーバ証明書のCN(Common Name)を指定します。省略した場合は、nullとなり、hostに指定されたホスト名を使用してサーバ証明書のCN(Common Name)を認証します。

注：“verify-ca”または“verify-full”を指定する場合、CA証明書ファイルをOSのシステム環境変数PGSSLROOTCERT(接続パラメータsslrootcert)で以下のように指定してください。

例)
変数名: PGSSLROOTCERT
変数値: CA証明書ファイル

注意

接続パラメータconnect_timeoutを利用する場合には、指定した各ホストへの接続に対してconnect_timeoutが適用されます。多重化されているデータベースサーバの両方がダウンしている場合、接続がタイムアウトするまでには、connect_timeoutの2倍の時間がかかります。

参考

接続先を指定するその他のクライアントコマンドも、psqlコマンドと同様の方法で接続先サーバの情報を指定してください。

第11章　Vertical Clustered Index(VCI)を利用した検索

VCIを利用した検索について説明します。

注意

本機能は、Advanced Editionのみで使用できます。

11.1 動作条件

参照する列すべてを指定してVCIインデックスを定義することで、VCIを利用した高速な集計が可能となります。

VCIを利用した検索が可能になる条件を以下に説明します。

なお、通常のインデックスと同じく、VCIを利用するかはコスト計算により判断されます。このため、VCIが利用可能な場合でも、他の実行計画の方がコストが低い場合は、その計画が選択されます。

VCIが選択されるSQL文

一般的なSELECT文のほかに、以下のSQL文でもVCIを利用することができます。ただし、“VCIが選択されない条件”に該当する場合は、VCIを利用しません。

- SELECT INTO ～
- CREATE TABLE AS SELECT ～
- CREATE MATERIALIZED VIEW ～ AS SELECT ～
- CREATE VIEW ～ AS SELECT ～
- COPY (SELECT ～) TO ～

VCIが選択されない条件

以下の条件の場合には、VCIを利用しません。

- 親問い合わせの列を参照する副問い合わせを指定している
- ロック処理句（FOR UPDATEなど）を指定している
- スクロール可能、またはWITH HOLDを指定したカーソルを使用している
- トランザクションの隔離レベルがSERIALIZABLEである
- “VCIが利用されない関数と演算子”に示した関数、または演算子を指定している
- ユーザ定義関数を指定している

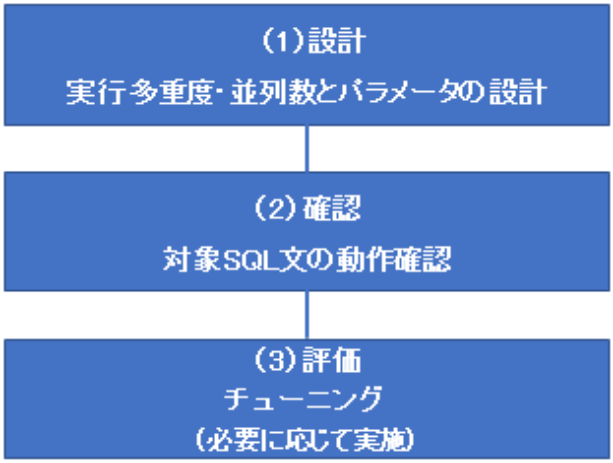
表11.1 VCIが利用されない関数と演算子

分類		関数/演算子
算術関数と演算子	乱数関数	random、setseed
文字列関数と演算子	文字列関数	引数の書式を指定したformat、regexp_matches、regexp_split_to_array、regexp_split_to_table
日付/時刻関数と演算子	日付/時刻関数	age(timestamp)、current_date、current_time、current_timestamp、localtime 、localtimestamp、statement_timestamp、transaction_timestamp
	遅延実行関数	pg_sleep、pg_sleep_for、pg_sleep_until
列挙型サポート関数		すべての関数と演算子

分類		関数/演算子
幾何関数と演算子		すべての関数と演算子
ネットワークアドレス関数と演算子		すべての関数と演算子
テキスト検索関数と演算子		すべての関数と演算子
XML関数		すべての関数
JSON関数と演算子		すべての関数と演算子
シーケンス操作関数		すべての関数
配列関数と演算子		すべての関数と演算子
範囲関数と演算子		すべての関数と演算子
集約関数	汎用集約関数	array_agg、json_agg、json_object_agg、string_agg、xmlagg
	統計処理用の集約関数	corr、covar_pop、covar_samp、regr_avgx、regr_avgy、regr_count、regr_intercept、regr_r2、regr_slope、regr_sxx、regr_sxy、regr_syy
	順序集合集約関数	すべての関数
	仮定集合集約関数	すべての関数
ウィンドウ関数		すべての関数
副問い合わせ式		左辺に行コンストラクタを指定した副問い合わせ式
行と配列の比較		行コンストラクタ、複合型の比較
集合を返す関数		すべての関数
システム情報関数		すべての関数
システム管理関数		すべての関数
トリガ関数		すべての関数
セション情報関数		current_role、current_user

11.2 使用方法

VCIの使用方法について、以下のフローに従って、説明します。



11.2.1 設計

VCIを使用するためには、事前に以下の設計を行います。

- SQL文の実行多重度と並列数の設計
- パラメータの設定

SQL文の実行多重度と並列数の設計

集計処理を行うVCIを利用した検索に割り当て可能なCPUコア数から、最大同時実行SQL文数と並列数を決定します。VCIを利用した検索を行うSQL文をどれだけの多重度で実行するのか、どれだけの並列数でVCIを利用した検索を行うのかを事前に設計してください。

例えば、割り当て可能なCPU数が32コアの場合、最大同時実行SQL文が8、並列数は4、などのように設計します。

注意

L

VCIを利用した検索では、SQL文ごとに動的共有メモリとして一時ファイルを“/dev/shm”、またはvci.smc_directoryパラメータに指定したディレクトリに作成します。そのため、“導入ガイド(サーバ編)”の“VCIで使用するメモリの見積り式”の“検索時にクエリごとに消費されるメモリ量”を参照し、本メモリ容量が本ディレクトリで十分確保できることも併せて、SQL文の実行多重度と並列数を見積もってください。万が一、検索実行時点で本ディレクトリの容量が不足している場合には、メモリ不足でSQL文がエラーとなります。

W

VCIを利用した検索では、SQL文ごとに動的共有メモリとして一時ファイルをデータ格納ディレクトリのサブディレクトリ、またはvci.smc_directoryパラメータに指定したディレクトリに作成します。そのため、“導入ガイド(サーバ編)”の“VCIで使用するメモリの見積り式”の“検索時にクエリごとに消費されるメモリ量”を参照し、本メモリ容量が本ディレクトリで十分確保できることも併せて、SQL文の実行多重度と並列数を見積もってください。万が一、検索実行時点で本ディレクトリの容量が不足している場合には、メモリ不足でSQL文がエラーとなります。

パラメータの設定

インスタンスを作成した直後のパラメータの設定では、VCIの並列検索機能は利用できません。

そのため、上記の“SQL文の実行多重度と並列数の設計”で設計した値を元に、以下のパラメータを設定します。

パラメータ名	用途	デフォルト	値の指標
vci.max_parallel_degree	1つのSQL文で利用するVCIの並列処理プロセス(バックグラウンドプロセス)の上限数を設定します。	0	並列数を指定します。
vci.smc_directory	VCIを利用した検索時に動的共有メモリとして一時ファイルを作成するディレクトリ名を指定します。	L /dev/shm W データ格納ディレクトリのサブディレクトリ(データ格納ディレクトリ¥¥base¥¥pgsql_tmp)	検索時にクエリごとに消費されるメモリ量を満たす空き容量が確保できるディレクトリを指定します。
max_worker_processes	システムがサポートするバックグラウンドプロセスの最大数を指定します。	8	VCIを利用した検索を行う最大同時実行SQL文数 × vci.max_parallel_degree の値を加算します。

参照

パラメータの詳細および設定については、“運用ガイド”の“パラメータ”を参照してください。

11.2.2 確認

対象のSQL文を“EXPLAIN ANALYZE”で実行し、以下の点を確認します。

- VCIが利用されているか
  VCIが利用されている場合、計画ノードに“Custom Scan (VCI…)”が表示されます。
- 並列数
  “Allocated Workers”にSQL文実行時の並列数が表示されます。設計した並列数で動作していることを確認します。
- レスポンス
  “Execution time”で表示された実行時間が想定通りかを確認します。

以下に、EXPLAIN ANALYZEの出力結果例を示します。

```
EXPLAIN ANALYZE SELECT COUNT(*) FROM test WHERE x > 10000;
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
-------
Custom Scan (VCI Aggregate)  (cost=19403.15..19403.16 rows=1 width=0) (actual time=58.505..58.506 rows=1
loops=1)
      Allocated Workers: 4
   ->  Custom Scan (VCI Scan) using test_x_idx on test  (cost=0.00..16925.00 rows=991261 width=0) (never
executed)
        Filter: (x > 10000)
Planning time: 0.151 ms
Execution time: 86.910 ms
(6 rows)
```

注意

VCIを利用した実行計画で出力されるコスト値は不正確な値が出力されます。VCIは、SQL文実行時の最良の実行計画の全体または一部を、VCIを利用した実行計画に置き換えることで動作します。置き換え対象の実行計画のコストが一定値(vci.cost_thresholdパラメータ)よりも低い場合は置き換えませんが、コストの再計算自体は行っていません。このため、元の実行計画のコスト値がそのまま出力されています。

11.2.3 評価

“11.2.2 確認”の結果から、以下の場合に、チューニングを行います。

VCIが利用されない場合

- “11.1 動作条件”を満たしていることを確認してください。
- “vci.enable”パラメータにonが指定されていることを確認してください。
- 大量のデータを挿入した直後など、統計情報が古くなった場合にVCIが適切に使用されない場合があります。このような場合は、VACUUM ANALYZE文、またはANALYZE文を実行してください。
- VCI検索用のメモリが不足する場合はVCIが利用されません。こうした事象は、VCIを定義したテーブルに対する長時間継続するトランザクションが存在する場合に、発生しやすくなります。vci.log_queryパラメータを“on”に設定し、“could not use VCI: local ROS size (%zu) exceeds limit (%zu)”、または“out of memory during local ROS generation”の警告が出力されるかを確認してください。出力される場合は、“vci.max_local_ros”パラメータの値を大きくしてください。

想定通りのレスポンスでない場合

以下のチューニングにより、レスポンスが改善される場合があります。

- “vci.max_parallel_degree”パラメータが未設定、または0が指定されている場合は、“11.2.1 設計”を基に適切な値を設定してください。

ー CPU使用率に余裕がある場合は、“vci.max_parallel_degree” の値を大きくして、再確認してください。また、“max_worker_processes”を並列検索を行う最大同時実行SQL文数×“vci.max_parallel_degree”よりも小さい値で指定している場合は、“max_worker_processes”の値を大きくして、再確認してください。

11.3 使用上の注意

VCIを利用する場合の注意事項を説明します。

- VCIの利用有無にかかわらず、結果の内容は変わりません。ただし、“ORDER BY”が指定されていない場合は、レコードの返却順については変わる可能性があります。
- リソースの消費を制限するため、特定の時間帯や特定の業務(SQLアプリケーション)においてのみ本機能を利用する場合は、postgresql.confのパラメータ編集やSET文により“vci.enable”パラメータの有効化／無効化を行ってください。
- オプティマイザヒント(pg_hint_plan)にVCIは指定できません。指定した場合、そのヒント句は無視されます。
- オプティマイザヒント(pg_hint_plan)にVCI以外のプランを指定した場合でも、VCIが利用される場合があります。このため、ヒント句で問い合わせ計画を指定する場合は、SET文で“vci.enable”パラメータにoffを指定してください。
- ストリーミングレプリケーションのスタンバイサーバでVCIを利用した検索を行うと、以下のメッセージが出力されることがあります。

```
"LOG: recovery has paused"
"HINT: Execute pg_wal_replay_resume() to continue."
```

本メッセージは、VCIを利用した検索によりVCIに対するWALの適用が一時的に待機するために出力されます。

- VCIを利用した検索を実行した場合でも、収集済み統計情報ビューであるpg_stat_all_indexesおよびpg_stat_user_indexesのidx_scan、idx_tup_read、idx_tup_fetch列の情報は更新されません。
- 現在は、パラレル集約の実行計画を、VCIを利用した実行計画に置き換えることはできません。このため、パーティションテーブルのある列にVCIを作成し、その列に対して集約(sum()など)すると、以下のいずれかのプランを選択します。対象のテーブルの状況に合わせて、設定パラメータを変えて使い分けてください。

  ー VCIスキャン以外のスキャン方法を使用したパラレル集約のプラン

  max_parallel_workers_per_gatherが1以上の場合に選択されます。

```
explain select sum(value) from test;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------------
 Finalize Aggregate  (cost=99906.30..99906.31 rows=1 width=8)
   ->  Gather  (cost=99906.08..99906.29 rows=2 width=8)
         Workers Planned: 2
         ->  Partial Aggregate  (cost=98906.08..98906.09 rows=1 width=8)
               ->  Parallel Append  (cost=0.00..94739.83 rows=1666500 width=4)
                     ->  Parallel Seq Scan on test_1  (cost=0.00..43203.67 rows=833250 width=4)
                     ->  Parallel Seq Scan on test_2  (cost=0.00..43203.67 rows=833250 width=4)
```

  このプランは、集約対象となる件数(検索条件にヒットする件数)が非常に多い場合には高速です。なぜなら、スキャンの性能ではなく集約を並列化するメリットが重要だからです。例えば、各パラレルワーカーはシーケンシャルスキャンしながら、スキャンしたレコードのほとんどを集約していきます。

  ー VCIスキャンの結果を1多重で集約するプラン

  max_parallel_workers_per_gatherを0に設定し、パラレル集約の実行計画を作成しないようにすることで選択されます。

```
explain select sum(value) from test;
                                        QUERY PLAN
-------------------------------------------------------------------------------------------------
 Aggregate  (cost=145571.00..145571.01 rows=1 width=8)
   ->  Append  (cost=0.00..135572.00 rows=3999600 width=4)
         ->  Custom Scan (VCI Scan) using test_1_id_value_idx on test_1  (cost=0.00..57787.00
rows=1999800 width=4)
```

```
               Allocated Workers: 2
         ->  Custom Scan (VCI Scan) using test_2_id_value_idx on test_2  (cost=0.00..57787.00
rows=1999800 width=4)
               Allocated Workers: 2
```

このプランは、集約対象となる件数がそれほど多くない場合や、レコードサイズに比べて集約対象の列のサイズが小さい場合には高速です。なぜなら、スキャン性能がより重要であるため、各パーティションのVCIスキャンの結果をより高速に集約できるからです。

- アクセス対象のパーティションが1つの場合には、本来ならば下記のようなVCI集約のプランを使用できます。下記は、パーティション除去によって、1つのパーティションのみをスキャンする場合の例です。

```
explain select sum(value) from test where id < 1000001;
                                      QUERY PLAN
-------------------------------------------------------------------------------------------------------
 Custom Scan (VCI Aggregate)  (cost=62786.50..62786.51 rows=1 width=8)
   Allocated Workers: 2
   ->  Custom Scan (VCI Scan) using test_1_id_value_idx on test_1  (cost=0.00..57787.00 rows=1999800
width=4)
         Filter: (id < 1000001)
```

しかし、現在のプランナは、テーブルがパーティショニングされていると、パラレル集約のプランを作成するため、VCI集約を選択しようとしません。そのため、このケースにおいてはmax_parallel_workers_per_gatherを0に設定して、強制的に、プランナがVCI集約を選択するようにしてください。

付録A　アプリケーション開発における注意事項

Fujitsu Enterprise Postgresでアプリケーションを開発する際の注意事項を説明します。

A.1　関数および演算子の注意事項

関数および演算子を使用した場合の注意事項を説明します。

A.1.1　関数および演算子の一般規則

関数および演算子を使用する場合の一般規則について説明します。アプリケーションの開発で関数および演算子を使用する場合は、一般規則に従って開発を行ってください。

一般規則

- 関数に指定する引数の数は、定められた数を指定してください。
- 関数に指定するデータ型は、定められたデータ型を指定してください。定められたデータ型と異なるデータ型の場合は、CASTを使用して明示的にデータ型の変換を行ってください。
- 演算子に指定するデータ型は、比較可能なデータ型を指定してください。比較できないデータ型の場合は、CASTを使用して明示的にデータ型の変換を行ってください。

参照

Fujitsu Enterprise Postgresで使用可能な関数および演算子については、“PostgreSQL Documentation”の“The SQL Language”の“Functions and Operators”を参照してください。

A.1.2　関数および演算子を使用したアプリケーション開発時の異常

関数および演算子を使用したアプリケーション開発時に発生したトラブルの事例、およびその対処方法について説明します。

SQL実行時に"関数 *** は存在しません"のエラーが発生する**

関数の一般規則が守られていないSQL文を実行した場合、以下のエラーが発生することがあります。

```
ERROR:  関数 ***** は存在しません
```

補足：***** にはエラーの発生した関数名とその引数のデータ型が表示されます。

エラーの発生した原因は以下のいずれかです。

- 指定した関数が存在しません。
- 指定した関数の引数の数、または引数のデータ型に誤りがあります。

【対処方法】

以下の点について確認を行い、誤りを修正してください。

- 指定した関数名、引数の数、または引数のデータ型に誤りがないか確認し、誤りがあれば修正してください。
- メッセージに表示された関数の引数のデータ型を確認し、意図していないデータ型が表示されている場合は、CASTなどを使用してデータ型の変換を行ってください。

SQL実行時に"演算子が存在しません"のエラーが発生する

演算子に比較できないデータ型を指定したSQL文を実行した場合、以下のエラーが発生することがあります。

```
ERROR:  演算子が存在しません: *****
```

補足：***** にはエラーの発生した演算子、および指定した値のデータ型が表示されます。

【対処方法】

演算子の左右に指定した式のデータ型が比較可能であるか確認してください。必要であればCASTなどで明示的にデータ型の変換を行い、左右のデータ型が比較可能となるように修正してください。

A.2 一時テーブルを使用する場合の注意事項

標準SQLでは、あらかじめ一時テーブルを定義しておくことで、アプリケーションがデータベースに接続した際に、自動的に空の一時テーブルを作成します。しかし、Fujitsu Enterprise Postgresでは、アプリケーションがデータベースに接続した際に、明示的にCREATE TABLE文を使用して一時テーブルを作成する必要があります。

同一セッションで同じ一時テーブルの作成と削除を繰り返すと、システムテーブルの膨張や使用メモリ量の増加につながることがあります。これを防ぐために、一度作成した一時テーブルを再利用するようにCREATE TABLE文を記述します。

たとえば、繰り返し実行する各トランザクションで、一時テーブルを作成・削除するような場合には、次のようにCREATE TABLE文を記述します。

- トランザクション開始時に一時テーブルが存在しないときだけ作成するように“IF NOT EXISTS”を指定します。
- トランザクション終了時にすべての行を削除するように“ON COMMIT DELETE ROWS”を指定します。

参照

CREATE TABLE文の詳細は、“PostgreSQL Documentation”の“Reference”の“SQL Commands”を参照してください。

一時テーブルを使用したSQLの例を以下に示します。

悪い例(一時テーブルを作成・削除する例)

```
BEGIN;
CREATE TEMPORARY TABLE mytable(col1 CHAR(4), col2 INTEGER) ON COMMIT DROP;
 ・・・  mytableに対する処理

COMMIT;
```

良い例(一時テーブルを再利用する例)

```
BEGIN;
CREATE TEMPORARY TABLE IF NOT EXISTS mytable(col1 CHAR(4), col2 INTEGER) ON COMMIT DELETE ROWS;
・・・  mytableに対する処理

COMMIT;
```

A.3 暗黙の型変換

暗黙の型変換とは、目的に合わせた型を明示的に指定しなくても、Fujitsu Enterprise Postgresにより自動的に行われる型変換です。

型変換可能な組合せは、変換元の値式が定数と定数でない場合で異なります。

定数でない場合は同じデータ型種類の範囲のみで型変換が可能となります。

定数の場合、文字列定数は変換先のデータ型に合わせた型変換が可能です。数定数は、内部的に特定の数値型に変換されます。そして内部的に変換された数定数が変換先のデータ型に合わせて数値型の範囲で型変換が可能となります。ビット文字列定数はビット列データ型のみ指定可能です。以下に定数の型変換の範囲を記載します。

表A.1 定数を含む暗黙の型変換可能なデータ型の組合せ

変換先		変換元		
		文字列定数(注1)	数定数(注2)	ビット文字列定数
数値型	SMALLINT	○	×	×
	INTEGER	○	○(注3)	×
	BIGINT	○	○(注4)	×
	DECIMAL	○	○(注5)	×
	NUMERIC	○	○(注5)	×
	REAL	○	×	×
	DOUBLE PRECISION	○	×	×
	SMALLSERIAL	○	×	×
	SERIAL	○	○(注3)	×
	BIGSERIAL	○	○(注4)	×
通貨型	MONEY	○	×	×
文字列型	CHAR	○	×	×
	VARCHAR	○	×	×
	NCHAR	○	×	×
	NCHAR VARYING	○	×	×
	TEXT	○	×	×
バイナリ列型	BYTEA	○	×	×
日付/時刻型	TIMESTAMP WITHOUT TIME ZONE	○	×	×
	TIMESTAMP WITH TIME ZONE	○	×	×
	DATE	○	×	×
	TIME WITHOUT TIME ZONE	○	×	×
	TIME WITH TIME ZONE	○	×	×
	INTERVAL	○	×	×
論理値型	BOOLEAN	○	×	×
幾何データ型	POINT	○	×	×
	LSEG	○	×	×
	BOX	○	×	×
	PATH	○	×	×
	POLYGON	○	×	×
	CIRCLE	○	×	×
ネットワークアドレス型	CIDR	○	×	×
	INET	○	×	×
	MACADDR	○	×	×
	MACADDR8	○	×	×
ビット列データ型	BIT	○	×	○

変換先		変換元		
		文字列定数(注1)	数定数(注2)	ビット文字列定数
	BIT VARYING	○	×	○
テキスト検索に関する型	TSVECTOR	○	×	×
	TSQUERY	○	×	×
UUID型	UUID	○	×	×
XML型	XML	○	×	×
JSON型	JSON	○	×	×

○:型変換可能
×:型変換不可能

注1) 変換先のデータ型へ型変換可能な形の文字列のみ指定可能(たとえば、変換先が数値型ならば'1'など)

注2) 数定数の表記については、最初に変換される特定の数値型を○で表しています

注3) INTEGER型で表現できる整数を指定可能

注4) INTEGER型で表現できず、BIGINT型で表現できる整数を指定可能

注5) INTEGER型およびBIGINT型で表現できず、NUMERIC型で表現できる整数、もしくは小数点または指数記号(e)を含んだ数定数を指定可能

暗黙の型変換は、データの比較および格納などにおいて利用可能です。

目的によって変換規則が異なりますので、目的ごとに説明します。

A.3.1 関数の引数

関数の引数に指定した値式は、関数の引数のデータ型へ型変換されます。

参照

関数の引数に指定できるデータ型は、"PostgreSQL Documentation"の"The SQL Language"の"Functions and Operators"を参照してください。

A.3.2 演算子

比較演算子、BETWEEN、IN

比較演算子、BETWEEN、INで、比較可能なデータ型の組合せを以下に示します。

表A.2 比較可能なデータ型の組合せ

左辺	右辺		
	数値型	文字列型	日付/時刻型
数値型	○	×	×
文字列型	×	○	×
日付/時刻型	×	×	○

○:比較可能
×:比較不可能

長さが異なる文字列どうしの比較は、短い文字列に空白を補い、長さを同じにします。

精度が異なる数値を比較する場合、精度の大きい方に型変換をします。

集合演算やCASEも同様の規則です。

その他の演算子

演算子に指定した値式は、演算子に指定可能なデータ型へ型変換されます。

参照

演算子に指定可能なデータ型は、“PostgreSQL Documentation”の“The SQL Language”の“Functions and Operators”を参照してください。

A.3.3 値の格納

INSERT文のVALUES句やUPDATE文のSET句に指定した値式は、格納する列のデータ型へ型変換されます。

A.4 インデックス使用時の注意事項

以下のインデックスを使用する場合の注意事項を説明します。

- SP-GiST インデックス

A.4.1 SP-GiST インデックス

SP-GiST インデックスを定義したテーブルに対して2多重以上で更新した場合に、アプリケーションが無応答になる場合があります。また、事象が発生した場合には、Check Pointer プロセスを始めとするすべてのシステム処理も無応答となります。これらの理由により、SP-GiST インデックスの使用は推奨されません。

A.5 定義名にマルチバイトを用いる場合の注意事項

データベースサーバがWindowsの場合には、データベース名およびユーザー名には、マルチバイト文字を用いないでください。

データベースサーバがWindows以外の場合であっても、注意すべき点や接続できないクライアントがあるため、データベース名とユーザー名には、マルチバイト文字を用いないことを推奨します。

以下に、その注意点と制約を示します。

1) クライアントの符号化方式の設定

CREATE時にクライアントの符号化方式を必ず設定してください。

参照

クライアントの符号化方式の設定方法は、“PostgreSQL Documentation”の“Server Administration”の“Character Set Support”を参照してください。

2) 接続に用いる名前の符号化方式

接続時に用いる名前の符号化方式は、これらの名前をCREATEしたときに接続していたデータベースの符号化方式と同じにしてください。

これは以下の理由によるものです。

- Fujitsu Enterprise Postgresの名前の記憶方式

  システムカタログには、CREATE時に接続していたデータベースの符号化方式で符号化した名前を保存します。

- 接続時のコード変換方針

  接続時には、クライアントから送信された名前をコード変換せずに、システムカタログ内の名前とのマッチングを行います。

したがって、例えば、定義時に接続していたデータベースの符号化方式がEUC_JP であり、かつ、UTF-8 で符号化されたデータベース名を指定して接続すると、データベースが存在しないものと見なされます。

3) 接続方法の制約

以下の前提で、各クライアント種別における接続方法の制約を下表に示します。

- 1)および2)の条件を満たすこと。
- データベース名とユーザー名の符号化方式は同一であること。

クライアント種別	クライアントOS	
	Windows(R)	Linux
JDBCドライバ	接続できません	接続できません
ODBCドライバ	接続できません	接続方法の制約はありません
.NET Data Provider	定義に用いた符号化方式がUTF-8以外では接続できません	－
C言語による埋め込みSQL	接続サービスファイル(pg_service.conf)以外では接続できません	接続方法の制約はありません
psqlコマンド	接続サービスファイル(pg_service.conf)以外では接続できません	接続方法の制約はありません

A.6 共有ライブラリを利用するアプリケーションのビルド・実行方法

Fujitsu Enterprise Postgresでアプリケーションを開発する際の、共有ライブラリの利用における以下の補足事項を説明します。

- アプリケーションのDT_RUNPATHの設定
- 間接的に利用するライブラリのアプリケーションへの直接リンク

A.6.1 アプリケーションのDT_RUNPATHの設定

アプリケーションが利用するライブラリの探索について

アプリケーションがlibpqやecpgといったFujitsu Enterprise Postgresの共有ライブラリを利用する場合、アプリケーション実行時にそれらのライブラリをロードする必要があります。

ライブラリのロードに当たり、ライブラリをマシン中から探索します。

ライブラリの探索では、探索する先のパスを指定できます。

パスの指定のために、アプリケーションやライブラリに記録されるDT_RUNPATH属性や、環境変数LD_LIBRARY_PATHが利用できます。

一般的に、DT_RUNPATH属性の利用が推奨されます。環境変数LD_LIBRARY_PATHは、該当アプリケーション以外の実行にも影響を及ぼしてしまう可能性があるためです。

以下では、DT_RUNPATHを利用する場合について説明します。

DT_RUNPATHの設定

アプリケーションのDT_RUNPATH属性に、利用するライブラリを格納する運用環境でのパスを設定するようにビルドしてください。

例えば、libpqやecpgを利用する場合、「<運用環境のFujitsu Enterprise Postgres クライアント機能のインストールディレクトリ>/lib」を設定してください。(注1)

DT_RUNPATH属性の設定方法は、ご利用のコンパイラ、またはリンカのドキュメントを参照してください。

設定するパスの種類や運用方法としては、以下の3種類があります。各運用に適した物を選択し、設定・運用してください。

(1)絶対パスを指定する方法

「<ライブラリを格納するディレクトリの絶対パス>/lib」を設定します。

(2)相対パスを指定する方法(注1)

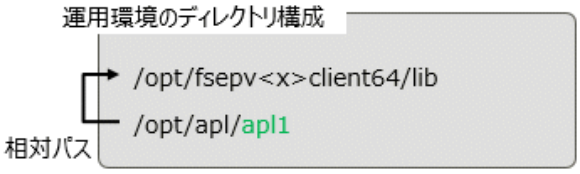
「<ライブラリを格納するディレクトリのアプリケーションからの相対パス>/lib」を設定します。

(3)任意のパスを指定、かつ、シンボリックリンクを利用する方法

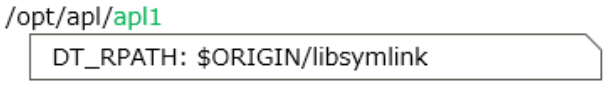
「(任意のパス)」を設定します。また、任意のパスの位置に、ライブラリを格納するディレクトリに対するシンボリックリンクを作成します。

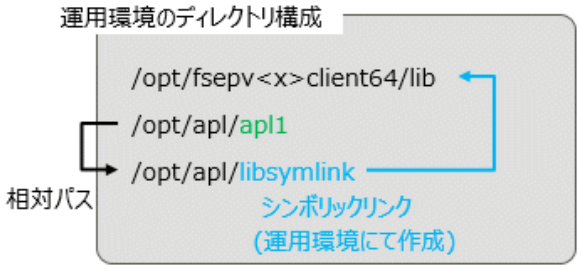
(1)絶対パスを指定

(2)相対パスを指定



(3)任意のパスを指定+シンボリックリンクを利用





注1:

例えば、gccでアプリケーションをビルドしている場合、gccのオプションとして"-Wl,-rpath,<運用環境のFujitsu Enterprise Postgresのインストールディレクトリ/lib>,--enable-new-dtags"を付与することで、DT_RUNPATHが設定できます。

参照

DT_RUNPATHの設定については、例えば、リンカldのrpathやenable-new-dtagsオプションを参照してください。

また、アプリケーションからの相対位置の指定については、ld.soの$ORIGINを参照してください。

DT_RUNPATHが設定できない場合

上記のいずれの設定もできない場合、アプリケーションの実行に際し、環境変数LD_LIBRARY_PATHに、ライブラリを格納するディレクトリへのパスを設定することで、アプリケーションが必要とする共有ライブラリを見つけることができます。

ただし、LD_LIBRARY_PATHを設定した状態で、該当のアプリケーション以外のコマンドやプログラムを実行した場合、それらが実行時エラーになる、または期待しない動作となる可能性があることに注意してください。

参照

共有ライブラリの探索についての詳細は、ld.soのmanページの説明を確認してください。

A.6.2 間接的に利用するライブラリのアプリケーションへの直接リンク

注意

本内容は煩雑である上、多くのアプリケーションにおいて本内容に該当する可能性は低いと考えられます。

アプリケーション作成時のテスト等において、アプリケーションが予期せず実行できない場合(下記で説明するように、libF3のロードに失敗するか、他ライブラリと競合が起こり実行時エラーとなる場合)にのみ、参考にすることをお勧めします。

Fujitsu Enterprise Postgresが同梱するライブラリを利用するアプリケーションについて、例えば、以下のように、libF1,libF2,libF3が他のアプリケーションやライブラリと以下のような依存関係を持つ場合、アプリケーションビルド時に、libF3を直接リンクするようビルドオプションに指定してください。

直接リンクしない場合、アプリケーション実行時にLD_LIBRARY_PATH にlibF3 が格納されるパスを指定してください。

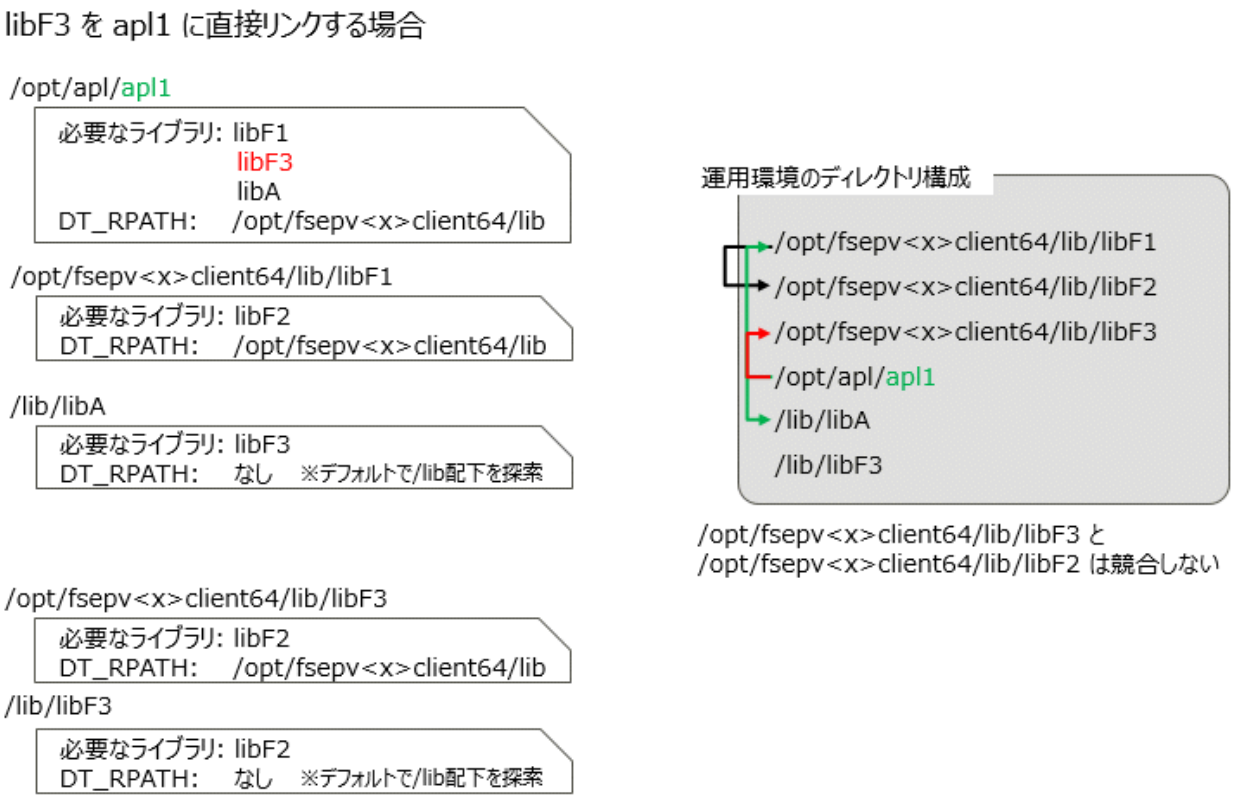
上記以外の場合、アプリケーション実行時にエラーとなる可能性があります。

例

以下を前提とした例で説明します。

- アプリケーションとライブラリの間に、以下のような依存関係がある。
  - apl1->libF1->libF2 (apl1がlibF1を必要とし、libF1がlibF2を必要とする)
  - apl1->libA->libF3->libF2 (apl1がlibAを必要とし、libAがlibF3を必要とし、libF3がlibF2を必要とする)
- libF1、libF2、libF3はFujitsu Enterprise Postgresが同梱するライブラリである。

libF3 を apl1 に直接リンクしない場合



一般的にライブラリのロードにおいて、あるlibAがlibBを必要とし、libBがlibCを必要とする場合、libBの探索にはlibAに設定されたDT_RUNPATHの値のみが利用され、libCの探索にはlibBに設定された DT_RUNPATH の値のみが利用されます。

上記の「libF3をapl1に直接リンクしない場合」の図において言えば、libF1の探索では、apl1に設定されたDT_RUNPATHの値が利用されます。libF2 の探索では、libF1 に設定されたDT_RUNPATHの値が利用されます。それぞれのDT_RUNPATHには、「Fujitsu Enterprise Postgresのインストールディレクトリ/lib」が設定されているため、libF1やlibF2を探索した際、Fujitsu Enterprise Postgres が同梱するlibF1、libF2 を発見し、それらをロードできます。

しかし、libF3 の探索では、Fujitsu Enterprise Postgres が同梱するlibF3 をロードできません。libF3の探索では、libAに設定されたDT_RUNPATHの値を利用しますが、libAにはDT_RUNPATHの値が設定されていないためです。

そのため、libF3を見つけることができずlibF3 のロードに失敗するか、または、マシン中に存在する、期待しないlibF3を発見し、それをロードしても、Fujitsu Enterprise Postgres が同梱するlibF2 との間で競合が起こり、実行時エラーとなる可能性があります。(ここでの競合とは、libF3で参照するシンボルが、libF2で定義されていないことです。)

上記の「libF3 をapl1に直接リンクする場合」の図で示したように、アプリケーションにlibF3を直接リンクすることで、アプリケーションのDT_RUNPATH を利用し、Fujitsu Enterprise Postgresが同梱するlibF3 をロードすることができます。または、LD_LIBRARY_PATH を設定することで、Fujitsu Enterprise Postgres が同梱するlibF3 をロードすることができます。

付録B　Oracleデータベースとの機能差異や記述差異に伴う移行手順

“第9章 Oracleデータベースとの互換性”に記載している範囲において、Fujitsu Enterprise PostgresとOracleデータベースについて以下の観点でFujitsu Enterprise Postgresへの移行手順を説明します。

- 機能差異
- 記述差異

移行対象のOracleデータベースのバージョンは、7～10.2gを想定しています。

B.1　外部結合演算子(外部結合を行う)

機能

WHERE句の条件式において、表結合として付帯したい表の列に外部結合演算子である(+)を付加することで結合表(OUTER JOIN)と同じ、外部結合を実現します。

B.1.1　^=比較演算子を使用して比較したい

Oracleデータベース

```
SELECT *
  FROM t1, t2
  WHERE t1.col1(+) ^= t2.col1;
```

注: col1はCHAR(4)型とします。

Fujitsu Enterprise Postgres

```
SELECT *
  FROM t1, t2
  WHERE t1.col1(+) != t2.col1;
```

注: col1はCHAR(4)型とします。

機能差異

Oracleデータベース

^=比較演算子を指定できます。

Fujitsu Enterprise Postgres

^=比較演算子を指定できません。

移行手順

以下の手順で移行してください。

1. “^=”キーワードで検索し、使用箇所を特定します。
2. 左辺または右辺に“(+)”キーワードを確認します。
3. “^=”を“ !=”に修正します。

B.2 DECODE(値を比較し対応する結果を返す)

機能

DECODEは、変換対象値式と各検索値の値を1つずつ比較し、変換対象値式と検索値とが一致する場合は対応する結果値を返却します。

B.2.1 文字列型の数値データと数字を比較したい

Oracleデータベース

```
SELECT DECODE( col1,
               1000, 'ITEM-A',
               2000, 'ITEM-B',
               'ITEM-C')
  FROM t1;
```

注: col1はCHAR(4)型とします。

Fujitsu Enterprise Postgres

```
SELECT DECODE( CAST(col1 AS INTEGER),
               1000, 'ITEM-A',
               2000, 'ITEM-B',
               'ITEM-C')
  FROM t1;
```

注: col1はCHAR(4)型とします。

機能差異

Oracleデータベース

変換対象値式と各検索値が、それぞれ文字列値と数値となる場合は、文字列値が比較対象となる数値のデータ型に変換されて比較します。

Fujitsu Enterprise Postgres

変換対象値式が文字列値である場合、各検索値を数値で指定できません。

移行手順

変換対象値式に指定できるデータ型は不定であるため、変換対象値式の値(例のcol1)をCASTを使用して明示的に数値(例ではINTEGER型)に変換してください。

B.2.2 50個以上の条件式の中から比較結果を求める

Oracleデータベース

```
SELECT DECODE(col1,
              1,'A',
              2,'B',
              ...
              78,'BZ',
              NULL,'UNKNOWN',
              'OTHER')
   FROM t1;
```

注: col1はINTEGER型とします。

Fujitsu Enterprise Postgres

```
SELECT CASE
          WHEN col1 = 1 THEN 'A'
          WHEN col1 = 2 THEN 'B'
          ...
          WHEN col1 = 78 THEN 'BZ'
          WHEN col1 IS NULL THEN 'UNKNOWN'
          ELSE 'OTHER'
       END
   FROM t1;
```

注: col1はINTEGER型とします。

機能差異

Oracleデータベース

最大127項目(引数合計が255以内まで)の検索値を指定できます。

Fujitsu Enterprise Postgres

最大49項目(引数合計が100以内まで)の検索値までしか指定できません。

移行手順

以下の手順でCASE式に移行してください。

1. DECODEの変換対象値式(例の第1引数のcol1)と探索値(例の第2引数の1)を、CASE式の探索条件に指定します。DECODEの結果値(例の第3引数の'A')を、CASE式のTHENに指定します(例のWHEN col1 = 1 THEN 'A')。なお、探索値がNULL値の場合は、CASE式の探索条件を'IS NULL'で指定します。
2. DECODEの省略値(例の最後の引数の'OTHER')が指定されている場合は、CASE式のELSEに省略値を指定します(例のELSE 'OTHER')。

B.2.3 データ型が統一されていない結果値から比較結果を求める

Oracleデータベース

```
SELECT DECODE( col1,
               '1000', 'A',
               '2000', 1,
               'OTHER')
FROM t1;
```

注: col1はCHAR(4)型とします。

Fujitsu Enterprise Postgres

```
SELECT DECODE( col1,
              '1000', 'A',
              '2000', '1',
              'OTHER')
FROM t1;
```

注: col1はCHAR(4)型とします。

機能差異

Oracleデータベース

すべての結果値のデータ型が最初に指定された結果値のデータ型に変換されます。

Fujitsu Enterprise Postgres

エラーとなります。

移行手順

以下の手順で移行してください。

1. 最初に指定した結果値の定数のデータ型を確認します。
2. 各結果値に指定した定数を、1.の定数のデータ型に変更します。

B.3 SUBSTR(指定した文字列の長さを切り出す)

機能

SUBSTRは、文字値式の開始位置の文字から文字列長分の文字列を抜き出して返却します。

SUBSTRを使用する場合は、“9.2.2 SUBSTRの注意事項”についても参照してください。

B.3.1 関数の引数に指定できるデータ型と異なるデータ型の値式を指定したい

Oracleデータベース

```
SELECT SUBSTR( col1,
               1,
               col2)
  FROM DUAL;
```

注: col1、col2はCHAR型とします。

Fujitsu Enterprise Postgres

```
CREATE CAST (CHAR AS INTEGER) WITH INOUT AS IMPLICIT;

SELECT SUBSTR( col1,
               1,
               col2)
  FROM DUAL;
# SELECT文に変更ありません;
```

注: col1、col2はCHAR型とします。

機能差異

Oracleデータベース

関数の引数に指定できるデータ型に型変換できる場合は暗黙の型変換が行われます。

Fujitsu Enterprise Postgres

データ型種類が異なる場合や、桁落ちが発生する場合は暗黙の型変換は行われません。

移行手順

文字列長のデータ型が明確であるため、文字列長に指定したCHAR型の値(例のcol2)がINTEGER型へ、暗黙の型変換されるように、先だって一回だけ以下のCREATE CASTを実行します。

```
CREATE CAST (CHAR AS INTEGER) WITH INOUT AS IMPLICIT;
```

B.3.2 日時型の値から指定した長さ分の文字列を抜き出したい

Oracleデータベース

```
SELECT SUBSTR( CURRENT_TIMESTAMP,
               1,
               8)
  FROM DUAL;
```

Fujitsu Enterprise Postgres

```
SELECT SUBSTR( TO_CHAR(CURRENT_TIMESTAMP,
               'DD-MON-YY HH.MI.SS.US PM')
                1,
                8)
  FROM DUAL;
```

機能差異

Oracleデータベース

文字値式にCURRENT_TIMESTAMPなどの日時型の値を指定できます。

Fujitsu Enterprise Postgres

文字値式にCURRENT_TIMESTAMPなどの日時型の値を指定できません。

移行手順

SUBSTRの文字値式に、TO_CHARを指定して、TO_CHARの1引数に日時型(例のCURRENT_TIMESTAMP)を指定し、第2引数に、移行前のSUBSTRの結果と同じになるように書式テンプレートパターン(例の'DD-MON-YY HH.MI.SS.US PM')を指定します。

TO_CHARの指定形式:TO_CHAR(第1引数, 第2引数)

参考

Fujitsu Enterprise PostgresのTO_CHARに指定可能な書式テンプレートパターンについては、“PostgreSQL Documentation”の“Data Type Formatting Functions”を参照してください。

B.3.3 文字列値とNULL値を連結したい

Oracleデータベース

```
SELECT SUBSTR( col1 || col2,
               2,
               5)
  FROM t1;
```

注: col1およびcol2は文字列型とし、かつcol2はNULL値を返す可能性のある列とします。

Fujitsu Enterprise Postgres

```
SELECT SUBSTR( col1 || NVL(col2, '')
               2,
               5)
  FROM t1;
```

注: col1およびcol2は文字列型とし、かつcol2はNULL値を返す可能性のある列とします。

機能差異

Oracleデータベース

NULL値は空文字列になって、文字列が連結されます。

Fujitsu Enterprise Postgres

NULL値は空文字列にはならず、文字列を連結した結果がNULL値になります。

移行手順

以下の手順で移行してください。

1. “||”というキーワードで検索し、使用箇所を特定します。
2. NULL値と結合するか確認します。NULL値と結合する可能性がある場合は、3.を行います。
3. NVL(値式,'')に修正します。

B.4 NVL(NULL値を置き換える)

機能

NVLは、NULL値を変換します。

B.4.1 データ型が統一されていない引数から結果を求める

Oracleデータベース

```
SELECT NVL( col1,
            col2)
  FROM t1;
```

注: col1はVARCHAR(100)型、col2はCHAR(100)型とします。

Fujitsu Enterprise Postgres

```
SELECT NVL( col1,
            CAST(col2 AS VARCHAR(100)))
  FROM t1;
```

注: col1はVARCHAR(100)型、col2はCHAR(100)型とします。

機能差異

Oracleデータベース

異なるデータ型の値式を指定できます。復帰値は第1引数が文字列値の場合はVARCHAR2、数値の場合は表現範囲の大きい方の数値型で返却します。

Fujitsu Enterprise Postgres

異なるデータ型の値式を指定できません。

移行手順

式1、および式2に指定できるデータ型が不定であるため、以下の手順で移行してください。

1. 式1、および式2に指定したデータ型を確認します。
2. 結果として受け取りたい方のデータ型で、もう一方の引数をCASTで明示的に型変換します。

B.4.2 ある日の日数を加算するなどの日時と数値の演算をしたい

Oracleデータベース

```
SELECT NVL( col1 + 10, CURRENT_DATE)
  FROM t1;
```

注: col1はTIMESTAMP WITHOUT TIME ZONE型、またはTIMESTAMP WITH TIME ZONE型とします。

Fujitsu Enterprise Postgres

```
SELECT NVL( CAST(col1 AS DATE) + 10, CURRENT_DATE)
  FROM t1;
```

注: col1はTIMESTAMP WITHOUT TIME ZONE型、またはTIMESTAMP WITH TIME ZONE型とします。

機能差異

Oracleデータベース

TIMESTAMP WITHOUT TIME ZONE型、およびTIMESTAMP WITH TIME ZONE型と数値の演算(加減算)をすることができます。演算結果は、DATE型になります。

Fujitsu Enterprise Postgres

TIMESTAMP WITHOUT TIME ZONE型、およびTIMESTAMP WITH TIME ZONE型と数値の演算(加減算)をすることができません。DATE型と数値の演算(加減算)をすることができます。

移行手順

以下の手順で移行してください。

1. “+”、および“-”というキーワードで演算(加減算)を行っている箇所を検索し、TIMESTAMP WITHOUT TIME ZONE型、およびTIMESTAMP WITH TIME ZONE型と数値の演算であるか確認します。
2. TIMESTAMP WITHOUT TIME ZONE型、およびTIMESTAMP WITH TIME ZONE型と数値の演算である場合は、CASTを使用して、明示的にTIMESTAMP WITHOUT TIME ZONE型、およびTIMESTAMP WITH TIME ZONE型をDATE型に変換します。

B.4.3 一定の期間経過した後の日付などINTERVAL型の計算結果を利用したい

Oracleデータベース

```
SELECT NVL( CURRENT_DATE + (col1 * 1.5), col2)
  FROM t1;
```

注: col1とcol2はINTERVAL YEAR TO MONTH型とします。

Fujitsu Enterprise Postgres

```
SELECT NVL( CURRENT_DATE +
        CAST(col1 * 1.5 AS
          INTERVAL YEAR TO MONTH), col2)
  FROM t1;
```

注: col1とcol2はINTERVAL YEAR TO MONTH型とします。

機能差異

Oracleデータベース

INTERVAL YEAR TO MONTH型の乗余算結果は、INTERVAL YEAR TO MONTH型となり端数(日数)は切り捨てられます。

Fujitsu Enterprise Postgres

INTERVAL YEAR TO MONTH型の乗余算結果は、INTERVAL型となり端数(日数)は切り捨てられません。

移行手順

以下の手順で移行してください。

1. “*”、および“/”というキーワードで乗余算を行っている箇所を検索し、指定されている値がINTERVAL YEAR TO MONTH型であるか確認します。
2. INTERVAL YEAR TO MONTH型である場合は、CASTを使用して演算結果を明示的にINTERVAL YEAR TO MONTH型に型変換します。

B.5 DBMS_OUTPUT(メッセージを出力する)

機能

DBMS_OUTPUTは、PL/pgSQLからpsqlなどのクライアントにメッセージを送信します。

B.5.1 処理の進行状況などのメッセージを画面に出力したい

Oracleデータベース

```
set serveroutput on;・・・(1)

DECLARE
  v_col1        CHAR(20);
  v_col2        INTEGER;
  CURSOR c1 IS
    SELECT col1, col2 FROM t1;
BEGIN
  DBMS_OUTPUT.PUT_LINE('-- BATCH_001 Start --');
  OPEN c1;
  DBMS_OUTPUT.PUT_LINE('-- LOOP Start --');
  LOOP
    FETCH c1 INTO v_col1, v_col2;
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT('.');
  END LOOP;
  DBMS_OUTPUT.NEW_LINE;・・・(2)
  DBMS_OUTPUT.PUT_LINE('-- LOOP End --');
  CLOSE c1;

  DBMS_OUTPUT.PUT_LINE('-- BATCH_001 End --');


EXCEPTION
  WHEN OTHERS THEN
     DBMS_OUTPUT.PUT_LINE('-- SQL Error --');
     DBMS_OUTPUT.PUT_LINE('ERROR : ' || SQLERRM );
END;
/
```

Fujitsu Enterprise Postgres

```
DO $$
DECLARE
    v_col1      CHAR(20);
    v_col2      INTEGER;
    c1 CURSOR FOR
        SELECT col1, col2 FROM t1;
BEGIN
```

```
    PERFORM DBMS_OUTPUT.SERVEROUTPUT(TRUE);・・・(1)
    PERFORM DBMS_OUTPUT.ENABLE(NULL);・・・(1)

    PERFORM DBMS_OUTPUT.PUT_LINE('-- BATCH_001 Start --');

    OPEN c1;
    PERFORM DBMS_OUTPUT.PUT_LINE('-- LOOP Start --');
    LOOP
        FETCH c1 INTO v_col1, v_col2;
        EXIT WHEN FOUND = false;
        PERFORM DBMS_OUTPUT.PUT('.');
    END LOOP;
    PERFORM DBMS_OUTPUT.NEW_LINE(); ・・・(2)

    PERFORM DBMS_OUTPUT.PUT_LINE('-- LOOP End --');
    CLOSE c1;

    PERFORM DBMS_OUTPUT.PUT_LINE('-- BATCH_001 End --');

EXCEPTION
    WHEN OTHERS THEN
        PERFORM DBMS_OUTPUT.PUT_LINE('-- SQL Error --');
        PERFORM DBMS_OUTPUT.PUT_LINE('ERROR : ' || SQLERRM );
END;
$$
;
```

(1) SERVEROUTPUT/ENABLE

記述差異

Oracleデータベース

SET文を使用してSERVEROUTPUT ONを指定します。

Fujitsu Enterprise Postgres

DBMS_SQL.SERVEROUTPUT(TRUE)を指定します。

移行手順

以下の手順で移行してください。

1. ストアドプロシジャのPL/SQLブロックより前にSET SERVEROUTPUT文が指定されているか確認します。
2. SET SERVEROUTPUT文が指定されている場合は、PL/pgSQLのBEGINの直後にDBMS_SQL.SERVEROUTPUTを指定します。画面に出力するためにONが指定されている場合はTRUEを指定します。OFFが指定されている場合はFALSEを指定します。
3. SET SERVEROUTPUTがONである場合のみ、追加でDBMS_SQL.ENABLEを指定します。引数に指定する値は以下の通りです。
   - SET SERVEROUTPUT文にSIZE指定がある場合は、指定されたサイズを引数に指定します。
   - SET SERVEROUTPUT文にSIZE指定がない場合は、Oracle10.1g以前の場合は2000、Oracle10.2g以降の場合はNULL値を指定します。

補足) ストアドプロシジャのPL/SQLブロックにDBMS_SQL.ENABLEが指定されている場合は、その引数の値に従ってください。

(2) NEW_LINE

記述差異

Oracleデータベース

“パッケージ名.機能名”の引数がない場合、括弧を省略することができます。

Fujitsu Enterprise Postgres

“パッケージ名.機能名”の引数がない場合でも、括弧を省略できません。

移行手順

以下の手順で移行してください。

1. “DBMS_OUTPUT.NEW_LINE”というキーワードでストアドプロシジャ内を検索します。
2. “パッケージ名.機能名”の後に括弧がない場合は、括弧を追記してください。

B.5.2 別のプロシージャ(PL/SQL)ブロックで実行した結果を受け取りたい(GET_LINESの場合)

Oracleデータベース

```
set serveroutput off;

DECLARE
    v_num        INTEGER;
BEGIN

    DBMS_OUTPUT.DISABLE;・・・(3)
    DBMS_OUTPUT.ENABLE(20000);・・・(4)
    DBMS_OUTPUT.PUT_LINE('-- ITEM CHECK --');


    SELECT count(*) INTO v_num FROM t1;

    IF v_num = 0 THEN
        DBMS_OUTPUT.PUT_LINE('-- NO ITEM --');

    ELSE
        DBMS_OUTPUT.PUT_LINE('-- IN ITEM(' || v_num || ') --');
    END IF;
END;
/

set serveroutput on;

DECLARE
    v_buffs      DBMSOUTPUT_LINESARRAY;・・・(5)
    v_num        INTEGER := 10;
BEGIN

    DBMS_OUTPUT.GET_LINES(v_buffs, v_num);・・・(5)

    FOR i IN 1..v_num LOOP
        DBMS_OUTPUT.PUT_LINE('LOG : ' || v_buffs(i));・・・(5)
    END LOOP;
END;
/
```

Fujitsu Enterprise Postgres

```
DO $$
DECLARE
    v_num        INTEGER;
```

```
BEGIN
    PERFORM DBMS_OUTPUT.SERVEROUTPUT(FALSE);
    PERFORM DBMS_OUTPUT.DISABLE();・・・(3)
    PERFORM DBMS_OUTPUT.ENABLE(20000);・・・(4)
PERFORM DBMS_OUTPUT.PUT_LINE('-- ITEM CHECK --');

    SELECT count(*) INTO v_num FROM t1;

    IF v_num = 0 THEN
        PERFORM DBMS_OUTPUT.PUT_LINE('-- NO ITEM --');
    ELSE
        PERFORM DBMS_OUTPUT.PUT_LINE('-- IN ITEM(' || v_num || ') --');
    END IF;
END;
$$
;

DO $$
DECLARE
    v_buffs      VARCHAR[];・・・(5)
    v_num        INTEGER := 10;
BEGIN
    PERFORM DBMS_OUTPUT.SERVEROUTPUT(TRUE);
    SELECT lines, numlines INTO v_buffs, v_num FROM DBMS_OUTPUT.GET_LINES(v_num);・・・(5)

    FOR i IN 1..v_num LOOP
        PERFORM DBMS_OUTPUT.PUT_LINE('LOG : ' || v_buffs[i]);・・・(5)
    END LOOP;
END;
$$
;
```

(3) DISABLE

DBMS_OUTPUTパッケージのNEW_LINEと同じです。記述差異および、記述差異に伴う移行手順については、NEW_LINEを参照してください。

(4) ENABLE

DBMS_OUTPUTパッケージのNEW_LINEと同じです。記述差異および、記述差異に伴う移行手順については、NEW_LINEを参照してください。

(5) GET_LINES

Oracleデータベースにおける記述形式

DBMS_OUTPUT.GET_LINES(第1引数, 第2引数)

記述差異

Oracleデータベース

取得する値は、引数に指定した変数で受け取ります。

Fujitsu Enterprise Postgres

取得する値は、DBMS_OUTPUT.GET_LINESの検索結果なので、SELECT文のINTO句に指定した変数で受け取ります。

移行手順

以下の手順で移行してください。

1. “DBMS_OUTPUT.GET_LINES”というキーワードでストアドプロシジャ内を検索し、呼び出し箇所を特定します。

2. DBMS_OUTPUT.GET_LINESの第1引数に指定している変数(例のv_buffs)のデータ型(例のDBMSOUTPUT_LINESARRAY)をVARCHAR型の配列(例のVARCHAR[])に変更します。

3. DBMS_OUTPUT.GET_LINESの呼び出し箇所を、SELECT INTO文に置き換えます。

   - 選択リストに、lines, numlinesと記載します。
   - INTO句に、DBMS_OUTPUT.GET_LINESに設定した第1引数(例のv_buffs)と第2引数を引数(例のv_num)と同じ順番で記載します。
   - FROM句にDBMS_OUTPUT.GET_LINESを記載します。引数は修正前の第2引数(例のv_num)のみ指定します。

4. 第1引数(例のv_buffs)を参照をしている箇所を洗い出し、PL/pgSQLの配列の参照形式(例のv_buffs[i])に変更します。

B.5.3 別のプロシージャ(PL/SQL)ブロックで実行した結果を受け取りたい(GET_LINEの場合)

Oracleデータベース

```
set serveroutput on;

DECLARE
    v_buff1       VARCHAR2(100);
    v_buff2       VARCHAR2(1000);
    v_num        INTEGER;
BEGIN

    v_buff2 := '';
    LOOP
        DBMS_OUTPUT.GET_LINE(v_buff1, v_num);・・・(6)
        EXIT WHEN v_num = 1;
        v_buff2 := v_buff2 || v_buff1;
    END LOOP;

    DBMS_OUTPUT.PUT_LINE(v_buff2);
END;
/
```

注: 値を取得する処理のみ記載しています。

Fujitsu Enterprise Postgres

```
DO $$
DECLARE
    v_buff1       VARCHAR(100);
    v_buff2       VARCHAR(1000);
    v_num        INTEGER;
BEGIN
    PERFORM DBMS_OUTPUT.SERVEROUTPUT(TRUE);
    v_buff2 := '';
    LOOP
        SELECT line, status INTO v_buff1, v_num FROM DBMS_OUTPUT.GET_LINE();・・・(6)
        EXIT WHEN v_num = 1;
        v_buff2 := v_buff2 || v_buff1;
    END LOOP;

    PERFORM DBMS_OUTPUT.PUT_LINE(v_buff2);
END;
$$
;
```

注: 値を取得する処理のみ記載しています。

(6) GET_LINE

Oracleデータベースにおける記述形式

DBMS_OUTPUT.GET_LINE(第1引数, 第2引数)

記述差異

Oracleデータベース

取得する値は、引数に指定した変数で受け取ります。

Fujitsu Enterprise Postgres

取得する値は、DBMS_OUTPUT.GET_LINEの検索結果なので、SELECT文のINTO句に指定した変数で受け取ります。

移行手順

以下の手順で移行してください。

1. “DBMS_OUTPUT.GET_LINE”というキーワードでストアドプロシジャ内を検索し、呼び出し箇所を特定します。
2. DBMS_OUTPUT.GET_LINEの呼び出し箇所を、SELECT INTO文に置き換えます。
   - 選択リストに、line, statusと記載します。
   - INTO句に、DBMS_OUTPUT.GET_LINEに設定した第1引数(例のv_buff1)と第2引数(例のv_num)を引数と同じ順番で記載します。
   - FROM句にDBMS_OUTPUT.GET_LINEを記載します。引数を指定しませんが、括弧は指定します。

B.6 UTL_FILE(ファイル操作を行う)

機能

UTL_FILEは、PL/pgSQLからテキストファイルの読込みと書込みを行います。

B.6.1 テキストファイルの読込みと書込みを行うディレクトリを登録したい

Oracleデータベース

```
[Oracle9i以前]
初期化パラメータで以下を設定
   UTL_FILE_DIR='/home/fsep' ・・・(1)

[Oracle9.2i以降]
   CREATE DIRECTORY文で設定
   CREATE DIRECTORY DIR AS '/home/fsep';・・・(1)
```

Fujitsu Enterprise Postgres

```
INSERT INTO UTL_FILE.UTL_FILE_DIR(dir)
       VALUES('/home/fsep');・・・(1)
```

(1) UTL_FILE_DIR/CREATE DIRECTORY

機能差異

Oracleデータベース

操作対象のディレクトリは、CREATE DIRECTORY文、または初期化パラメータのUTL_FILE_DIRを使用して設定します。

Fujitsu Enterprise Postgres

操作対象のディレクトリは、CREATE DIRECTORY文、または初期化パラメータのUTL_FILE_DIRで設定できません。

移行手順

INSERT文を使用してUTL_FILE.UTL_FILE_DIRテーブルに対象ディレクトリ情報を設定します。なお、この移行作業は、PL/pgSQL関数の実行に先立って、1回実施してください。

ー 初期化パラメータのUTL_FILE_DIRを使用している場合

1. 初期化パラメータのUTL_FILE_DIRの値(例の'/home/fsep')を確認します。
2. 1.で確認したディレクトリ名を、INSERT文で指定して実行します。
   - INTO句にはUTL_FILE.UTL_FILE_DIR(dir)を指定します。
   - VALUES句には対象のディレクトリ名を文字列定数(例の'/home/fsep')で指定します。
   - 複数のディレクトリが指定されている場合は、ディレクトリ単位に複数回INSERT文を実行します。

ー CREATE DIRECTORY文を使用している場合

1. CREATE DIRECTORY文で登録したディレクトリ名(例の'/home/fsep')を確認します。確認は、DBA権限を持つユーザーでSQL*Plusにログインし、show ALL_DIRECTORIES;を実行します。
2. 1.で確認したディレクトリ名を、INSERT文で指定して実行します。INSERT文の指定方法は初期化パラメータのUTL_FILE_DIRを使用している場合と同じです。

B.6.2 ファイルの情報を確認したい

Oracleデータベース

```
CREATE PROCEDURE read_file(fname VARCHAR2) AS


    v_file      UTL_FILE.FILE_TYPE;
    v_exists    BOOLEAN;
    v_length    NUMBER;
    v_bsize     INTEGER;
    v_rbuff     VARCHAR2(1024);
BEGIN


    UTL_FILE.FGETATTR('DIR', fname, v_exists, v_length, v_bsize);・・・(2)


    IF v_exists <> true THEN
        DBMS_OUTPUT.PUT_LINE('-- FILE NOT FOUND --');
        RETURN;
    END IF;

    DBMS_OUTPUT.PUT_LINE('-- FILE DATA --');

    v_file := UTL_FILE.FOPEN('DIR', fname, 'r', 1024);・・・(3)
    FOR i IN 1..3 LOOP
        UTL_FILE.GET_LINE(v_file, v_rbuff, 1024);・・・(4)
        DBMS_OUTPUT.PUT_LINE(v_rbuff);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('... more');
    DBMS_OUTPUT.PUT_LINE('-- READ END --');

    UTL_FILE.FCLOSE(v_file);・・・(5)
    RETURN;

EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('-- FILE END --');
```

```
        UTL_FILE.FCLOSE(v_file);
        RETURN;
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('-- SQL Error --');

        DBMS_OUTPUT.PUT_LINE('ERROR : ' || SQLERRM );
        UTL_FILE.FCLOSE_ALL;・・・(6)
        RETURN;
END;
/

set serveroutput on

call read_file('file01.txt');
```

Fujitsu Enterprise Postgres

```
CREATE FUNCTION read_file(fname VARCHAR) RETURNS void AS $$
DECLARE
    v_file      UTL_FILE.FILE_TYPE;
    v_exists    BOOLEAN;
    v_length    NUMERIC;
    v_bsize     INTEGER;
    v_rbuff     VARCHAR(1024);
BEGIN
    PERFORM DBMS_OUTPUT.SERVEROUTPUT(TRUE);

    SELECT fexists, file_length, blocksize
       INTO v_exists, v_length, v_bsize
       FROM UTL_FILE.FGETATTR('/home/fsep', fname);・・・(2)
    IF v_exists <> true THEN
        PERFORM DBMS_OUTPUT.PUT_LINE('-- FILE NOT FOUND --');
        RETURN;
    END IF;

    PERFORM DBMS_OUTPUT.PUT_LINE('-- FILE DATA --');
    v_file := UTL_FILE.FOPEN('/home/fsep', fname, 'w', 1024);・・・(3)
    FOR i IN 1..3 LOOP
        v_rbuff := UTL_FILE.GET_LINE(v_file, 1024);・・・(4)
        PERFORM DBMS_OUTPUT.PUT_LINE(v_rbuff);
    END LOOP;
    PERFORM DBMS_OUTPUT.PUT_LINE('... more');
    PERFORM DBMS_OUTPUT.PUT_LINE('-- READ END --');

    v_file := UTL_FILE.FCLOSE(v_file);・・・(5)
    RETURN;

EXCEPTION
    WHEN NO_DATA_FOUND THEN
        PERFORM DBMS_OUTPUT.PUT_LINE('-- FILE END --');
        v_file := UTL_FILE.FCLOSE(v_file);
        RETURN;
    WHEN OTHERS THEN
        PERFORM DBMS_OUTPUT.PUT_LINE('-- SQL Error --');
        PERFORM DBMS_OUTPUT.PUT_LINE('ERROR : ' || SQLERRM );
        PERFORM UTL_FILE.FCLOSE_ALL();・・・(6)
        RETURN;
END;
$$
LANGUAGE plpgsql;
```

```
SELECT read_file('file01.txt');
```

(2) FGETATTR

Oracleデータベースにおける記述形式

UTL_FILE.FGETATTR(第1引数, 第2引数, 第3引数, 第4引数, 第5引数)

機能差異

Oracleデータベース

CREATE DIRECTORY文を使用している場合(Oracle9.2i以降)、ディレクトリ名にディレクトリ・オブジェクト名を指定します。

Fujitsu Enterprise Postgres

ディレクトリ名にディレクトリ・オブジェクト名は指定できません。

記述差異

Oracleデータベース

取得する値は、引数に指定した変数で受け取ります。

Fujitsu Enterprise Postgres

取得する値は、UTL_FILE.FGETATTRの検索結果なので、SELECT文のINTO句に指定した変数で受け取ります。

移行手順

以下の手順で移行してください。ディレクトリ・オブジェクト名と実際のディレクトリ名の対応の確認方法は、UTL_FILE_DIR/CREATE DIRECTORYを参照してください。

1. “UTL_FILE.FOPEN”というキーワードでストアドプロシジャ内を検索し、呼び出し箇所を特定します。
2. ディレクトリ・オブジェクト名(例の'DIR')に対応する実際のディレクトリ名(例の'/home/fsep')を確認します。
3. 第1引数のディレクトリ・オブジェクト名(例の'DIR')を、2.で確認した実際のディレクトリ名(例の'/home/fsep')に置き換えます。
4. UTL_FILE.FGETATTRの呼び出し箇所を、SELECT INTO文に置き換えます。
   - 選択リストに、fexists, file_length, blocksizeと記載します。
   - INTO句に、UTL_FILE.FGETATTRに設定していた第3引数～第5引数(例のv_exists, v_length, v_bsize)を引数と同じ順番で記載します。
   - FROM句にUTL_FILE.FGETATTRを記載します。引数は修正前の第1引数の実際のディレクトリ名(例の'/home/fsep')と第2引数(例のfname)のみ指定します。

(3) FOPEN

Oracleにおける記述形式

UTL_FILE.FOPEN(第1引数, 第2引数, 第3引数, 第4引数)

機能差異

Oracleデータベース

CREATE DIRECTORY文を使用している場合(Oracle9.2i以降)、ディレクトリ名にディレクトリ・オブジェクト名を指定します。

Fujitsu Enterprise Postgres

ディレクトリ名にディレクトリ・オブジェクト名は指定できません。

移行手順

以下の手順で移行してください。ディレクトリ・オブジェクト名と実際のディレクトリ名の対応の確認方法は、UTL_FILE_DIR/CREATE DIRECTORYを参照してください。

1. “UTL_FILE.FOPEN”というキーワードでストアドプロシジャ内を検索し、呼び出し箇所を特定します。
2. ディレクトリ・オブジェクト名(例の'DIR')に対応する実際のディレクトリ名(例の'/home/fsep')を確認します。
3. 第1引数のディレクトリ・オブジェクト名(例の'DIR')を、1.で確認した実際のディレクトリ名(例の'/home/fsep')に置き換えます。

(4) GET_LINE

Oracleデータベースにおける記述形式

UTL_FILE.GET_LINE(第1引数, 第2引数, 第3引数)

記述差異

Oracleデータベース

取得する値は、引数に指定した変数で受け取ります。

Fujitsu Enterprise Postgres

取得する値は、UTL_FILE.GET_LINEの復帰値なので、代入文に指定した変数で受け取ります。

移行手順

以下の手順で移行してください。

1. “UTL_FILE.GET_LINE”というキーワードでストアドプロシジャ内を検索し、呼び出し箇所を特定します。
2. UTL_FILE.GET_LINEの呼び出し箇所を、値の代入(:=)に置き換えてください。
   - 左辺にUTL_FILE.GET_LINEに指定していた第2引数(例のv_rbuff)を指定します。
   - 右辺にUTL_FILE.GET_LINEを記載します。引数は修正前の第1引数(例のv_file)と第3引数(例の1024)のみ指定します。

(5) FCLOSE

Oracleデータベースにおける記述形式

UTL_FILE.FCLOSE(第1引数)

記述差異

Oracleデータベース

クローズすると、引数に指定したファイルハンドラがNULL値になります。

Fujitsu Enterprise Postgres

クローズすると、UTL_FILE.FCLOSEの復帰値がNULL値になりますので、代入文に指定したファイルハンドラで受け取ります。

移行手順

以下の手順で移行してください。

1. “UTL_FILE.FCLOSE”というキーワードでストアドプロシジャ内を検索し、呼び出し箇所を特定します。
2. UTL_FILE.FCLOSEの呼び出し箇所を、値の代入(:=)に置き換え、ファイルハンドラ(例のv_file)がNULL値となるようにします。
   - 左辺にUTL_FILE.FCLOSEに指定していた引数(例のv_file)を指定します。
   - 右辺にUTL_FILE.FCLOSEを記載します。引数は修正前と同じ値(例のv_file)を指定します。

(6) FCLOSE_ALL

DBMS_OUTPUTパッケージのNEW_LINEと同じです。記述差異および、記述差異に伴う移行手順については、DBMS_OUTPUTパッケージのNEW_LINEを参照してください。

B.6.3 バックアップなどでファイルをコピーしたい

Oracleデータベース

```
CREATE PROCEDURE copy_file(fromname VARCHAR2, toname VARCHAR2) AS
BEGIN


    UTL_FILE.FCOPY('DIR1', fromname, 'DIR2', toname, 1, NULL);・・・(7)

    RETURN;

EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('-- SQL Error --');

        DBMS_OUTPUT.PUT_LINE('ERROR : ' || SQLERRM );
        RETURN;
END;
/

set serveroutput on

call copy_file('file01.txt','file01_bk.txt');
```

Fujitsu Enterprise Postgres

```
CREATE FUNCTION copy_file(fromname VARCHAR, toname VARCHAR) RETURNS void AS $$
BEGIN
    PERFORM DBMS_OUTPUT.SERVEROUTPUT(TRUE);

    PERFORM UTL_FILE.FCOPY('/home/fsep', fromname, '/home/backup', toname, 1, NULL);・・・(7)
    RETURN;

EXCEPTION
    WHEN OTHERS THEN
        PERFORM DBMS_OUTPUT.PUT_LINE('-- SQL Error --');
        PERFORM DBMS_OUTPUT.PUT_LINE('ERROR : ' || SQLERRM );
        RETURN;
END;
$$
LANGUAGE plpgsql;


SELECT copy_file('file01.txt','file01_bk.txt');
```

(7) FCOPY

Oracleデータベースにおける記述形式

UTL_FILE.FCOPY(第1引数, 第2引数, 第3引数, 第4引数, 第5引数, 第6引数)

機能差異

Oracleデータベース

CREATE DIRECTORY文を使用している場合(Oracle9.2i以降)、ディレクトリ名にディレクトリ・オブジェクト名を指定します。

Fujitsu Enterprise Postgres

ディレクトリ名にディレクトリ・オブジェクト名は指定できません。

移行手順

以下の手順で移行してください。ディレクトリ・オブジェクト名と実際のディレクトリ名の対応の確認方法は、UTL_FILE_DIR/CREATE DIRECTORYを参照してください。

1. “UTL_FILE.FCOPY”というキーワードでストアドプロシジャ内を検索し、呼び出し箇所を特定します。
2. 第1引数、および第3引数のディレクトリ・オブジェクト名(例の'DIR1'と'DIR2')に対応する実際のディレクトリ名(例の'/home/fsep'と'/home/backup')を確認します。
3. ディレクトリ・オブジェクト名(例の'DIR1'と'DIR2')を、1.で確認した実際のディレクトリ名(例の'/home/fsep'と'/home/backup')に置き換えます。

B.6.4 バックアップなどでファイルを移動したい(ファイルの名前を変えたい)

Oracleデータベース

```
CREATE PROCEDURE move_file(fromname VARCHAR2, toname VARCHAR2) AS
BEGIN


    UTL_FILE.FRENAME('DIR1', fromname, 'DIR2', toname, FALSE);・・・(8)
    RETURN;

EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('-- SQL Error --');

        DBMS_OUTPUT.PUT_LINE('ERROR : ' || SQLERRM );
        RETURN;
END;
/

set serveroutput on

call move_file('file01.txt','file02.txt');
```

Fujitsu Enterprise Postgres

```
CREATE FUNCTION move_file(fromname VARCHAR, toname VARCHAR) RETURNS void AS $$
BEGIN
    PERFORM DBMS_OUTPUT.SERVEROUTPUT(TRUE);

    PERFORM UTL_FILE.FRENAME('/home/fsep', fromname, '/home/backup', toname, FALSE);・・・(8)
    RETURN;

EXCEPTION
    WHEN OTHERS THEN
        PERFORM DBMS_OUTPUT.PUT_LINE('-- SQL Error --');
        PERFORM DBMS_OUTPUT.PUT_LINE('ERROR : ' || SQLERRM );
        RETURN;
END;
$$
LANGUAGE plpgsql;


SELECT move_file('file01.txt','file02.txt');
```

(8) FRENAME

UTL_FILEパッケージのFCOPYと同じです。記述差異および、記述差異に伴う移行手順については、UTL_FILEパッケージのFCOPYを参照してください。

B.7 DBMS_SQL(動的SQLを実行する)

機能

DBMS_SQLは、PL/pgSQLから動的SQLを実行することができます。

B.7.1 カーソルを使って検索したい

Oracleデータベース

```
CREATE PROCEDURE search_test(h_where CLOB) AS


    str_sql     CLOB;
    v_cnt       INTEGER;
    v_array     DBMS_SQL.VARCHAR2A;
    v_cur       INTEGER;
    v_smpid     INTEGER;
    v_smpnm     VARCHAR2(20);
    v_addbuff   VARCHAR2(20);
    v_smpage    INTEGER;
    errcd       INTEGER;
    length      INTEGER;
    ret         INTEGER;
BEGIN

    str_sql     := 'SELECT smpid, smpnm FROM smp_tbl WHERE ' || h_where || ' ORDER BY smpid';
    v_smpid     := 0;
    v_smpnm     := '';
    v_smpage    := 0;

    v_cur := DBMS_SQL.OPEN_CURSOR;・・・(1)

    v_cnt :=
      CEIL(DBMS_LOB.GETLENGTH(str_sql)/1000);
    FOR idx IN 1 .. v_cnt LOOP
        v_array(idx) :=
            DBMS_LOB.SUBSTR(str_sql,
                            1000,
                            (idx-1)*1000+1);
    END LOOP;
    DBMS_SQL.PARSE(v_cur, v_array, 1, v_cnt, FALSE, DBMS_SQL.NATIVE);・・・(2)

    DBMS_SQL.DEFINE_COLUMN(v_cur, 1, v_smpid);

    DBMS_SQL.DEFINE_COLUMN(v_cur, 2, v_smpnm, 10);


    ret := DBMS_SQL.EXECUTE(v_cur);
    LOOP
        v_addbuff := '';

        IF DBMS_SQL.FETCH_ROWS(v_cur) = 0 THEN
            EXIT;
        END IF;

        DBMS_OUTPUT.PUT_LINE('------------------------------------------------------');
```

```
        DBMS_SQL.COLUMN_VALUE(v_cur, 1, v_smpid, errcd, length);・・・(3)




        IF errcd = 1405 THEN・・・(3)

          DBMS_OUTPUT.PUT_LINE('smpid       = (NULL)');
        ELSE
          DBMS_OUTPUT.PUT_LINE('smpid       = ' || v_smpid);
        END IF;

        DBMS_SQL.COLUMN_VALUE(v_cur, 2, v_smpnm, errcd, length);

        IF errcd = 1406 THEN
          v_addbuff := '... [len=' || length || ']';
        END IF;
        IF errcd = 1405 THEN
          DBMS_OUTPUT.PUT_LINE('v_smpnm     = (NULL)');
        ELSE
          DBMS_OUTPUT.PUT_LINE('v_smpnm     = ' || v_smpnm || v_addbuff );
        END IF;

DBMS_OUTPUT.PUT_LINE('------------------------------------------------------------');

        DBMS_OUTPUT.NEW_LINE;
    END LOOP;

    DBMS_SQL.CLOSE_CURSOR(v_cur);・・・(4)

    RETURN;
END;
/

Set serveroutput on

call search_test('smpid < 100');
```

Fujitsu Enterprise Postgres

```
CREATE FUNCTION search_test(h_where text) RETURNS void AS $$
DECLARE
    str_sql     text;


    v_cur       INTEGER;
    v_smpid     INTEGER;
    v_smpnm     VARCHAR(20);
    v_addbuff   VARCHAR(20);
    v_smpage    INTEGER;
    errcd       INTEGER;
    length      INTEGER;
    ret         INTEGER;
BEGIN
    PERFORM DBMS_OUTPUT.SERVEROUTPUT(TRUE);
    str_sql     := 'SELECT smpid, smpnm FROM smp_tbl WHERE ' || h_where || ' ORDER BY smpid';
    v_smpid     := 0;
    v_smpnm     := '';
    v_smpage    := 0;

    v_cur := DBMS_SQL.OPEN_CURSOR();・・・(1)

    PERFORM DBMS_SQL.PARSE(v_cur, str_sql, 1);・・・(2)
```

```
    PERFORM DBMS_SQL.DEFINE_COLUMN(v_cur, 1, v_smpid);
    PERFORM DBMS_SQL.DEFINE_COLUMN(v_cur, 2, v_smpnm, 10);

    ret := DBMS_SQL.EXECUTE(v_cur);
    LOOP
        v_addbuff := '';

        IF DBMS_SQL.FETCH_ROWS(v_cur) = 0 THEN
            EXIT;
        END IF;

        PERFORM DBMS_OUTPUT.PUT_LINE('-------------------------------------------------------');
        SELECT value,column_error,actual_length
          INTO v_smpid, errcd, length
          FROM DBMS_SQL.COLUMN_VALUE(v_cur,
                                     1,
                                     v_smpid);・・・(3)
        IF errcd = 22002 THEN・・・(3)
          PERFORM DBMS_OUTPUT.PUT_LINE('smpid       = (NULL)');
        ELSE
          PERFORM DBMS_OUTPUT.PUT_LINE('smpid       = ' || v_smpid);
        END IF;

       SELECT value,column_error,actual_length INTO v_smpnm, errcd, length FROM DBMS_SQL.COLUMN_VALUE(v_cur,
2, v_smpnm);
        IF errcd = 22001 THEN
          v_addbuff := '... [len=' || length || ']';
        END IF;
        IF errcd = 22002 THEN
          PERFORM DBMS_OUTPUT.PUT_LINE('v_smpnm     = (NULL)');
        ELSE
          PERFORM DBMS_OUTPUT.PUT_LINE('v_smpnm     = ' || v_smpnm || v_addbuff );
        END IF;

        PERFORM DBMS_OUTPUT.PUT_LINE('-------------------------------------------------------');
        PERFORM DBMS_OUTPUT.NEW_LINE();
    END LOOP;

    v_cur := DBMS_SQL.CLOSE_CURSOR(v_cur);・・・(4)
    RETURN;
END;
$$
LANGUAGE plpgsql;


SELECT search_test('smpid < 100');
```

(1) OPEN_CURSOR

DBMS_OUTPUTパッケージのNEW_LINEと同じです。記述差異および、記述差異に伴う移行手順については、DBMS_OUTPUTパッケージのNEW_LINEを参照してください。

(2) PARSE

Oracleデータベースにおける記述形式

DBMS_SQL.PARSE(第1引数, 第2引数, 第3引数, 第4引数, 第5引数, 第6引数)

機能差異

Oracleデータベース

SQL文を文字列の表タイプ(VARCHAR2A型、VARCHAR2S型)で指定することができます。これは、第2引数に指定します。

SQL文を処理する方法の指定にDBMS_SQL.NATIVE、DBMS_SQL.V6、DBMS_SQL.V7を指定することができます。

Fujitsu Enterprise Postgres

SQL文を文字列の表タイプでは指定できません。

SQL文を処理する方法の指定にDBMS_SQL.NATIVE、DBMS_SQL.V6、DBMS_SQL.V7を指定することができません。

移行手順

以下の手順で移行してください。

1. “DBMS_SQL.PARSE”というキーワードでストアドプロシジャ内を検索し、呼び出し箇所を特定します。
2. 第2引数(例のv_array)に指定されたSQL文のデータ型を確認します。
   - データ型がDBMS_SQL.VARCHAR2A型、またはDBMS_SQL.VARCHAR2S型である場合は、表タイプによる指定です。3.の手順から移行処理を継続してください。
   - データ型がDBMS_SQL.VARCHAR2A型、またはDBMS_SQL.VARCHAR2S型でない場合は、文字列による指定です。7.の手順から移行処理を継続してください。
3. DBMS_SQL.VARCHAR2A型、およびDBMS_SQL.VARCHAR2S型に分割する前のSQL文(例のstr_sql)を確認します。
4. DBMS_SQL.VARCHAR2A型、およびDBMS_SQL.VARCHAR2S型にSQLを分割している一連の処理(例のFOR idx付近の処理)を削除します。
5. 第2引数を2.で確認した分割する前のSQL文(例のstr_sql)に置き換えます。
6. 第3引数～第5引数(例のv_cnt, FALSE, DBMS_SQL.NATIVE)までを削除します。
7. DBMS_SQL.NATIVE、DBMS_SQL.V6、DBMS_SQL.V7が指定されている場合は、第3引数を数定数１に置き換えます。
   - DBMS_SQL.VARCHAR2A型、またはDBMS_SQL.VARCHAR2S型を使用している場合は第6引数が該当します。
   - DBMS_SQL.VARCHAR2A型、またはDBMS_SQL.VARCHAR2S型を使用していない場合は第3引数が該当します。

(3) COLUMN_VALUE

Oracleデータベースにおける記述形式

DBMS_SQL.COLUMN_VALUE(第1引数, 第2引数, 第3引数, 第4引数, 第5引数)

機能差異

Oracleデータベース

column_errorに対して以下のエラーコードを戻します。

- 1406 : 取得した値が切り捨てられている
- 1405 : 取得した値の内容がNULL値

Fujitsu Enterprise Postgres

column_errorに対して以下のエラーコードを戻します。

- 22001 : 取得した値が切り捨てられている

- 22002 : 取得した値の内容がNULL値

記述差異

Oracleデータベース

取得する値は、引数に指定した変数で受け取ります。

Fujitsu Enterprise Postgres

取得する値は、DBMS_SQL.COLUMN_VALUEの検索結果なので、SELECT文のINTO句に指定した変数で受け取ります。

移行手順

以下の手順で移行してください。

1. “DBMS_SQL.COLUMN_VALUE”というキーワードでストアドプロシジャ内を検索し、呼び出し箇所を特定します。
2. DBMS_SQL.COLUMN_VALUEの呼び出し箇所を、SELECT INTO文に置き換えます。
   - DBMS_SQL.COLUMN_VALUEの第3引数(例のv_smpid)以降に指定している引数の数を確認します(例の場合は、v_smpid, errcd, length の3個)。
   - 選択リストに、先で確認した引数の数に応じてそれぞれvalue、column_error、actual_lengthの順で指定します。(例えば、第3引数のみ指定されている場合はvalueのみを指定します)
   - INTO句に、DBMS_SQL.COLUMN_VALUEに設定していた第3引数～第5引数(例のv_smpid, errcd, length)を同じ順番で指定します。
   - FROM句にDBMS_SQL.COLUMN_VALUEを記載します。引数は修正前の第1引数～第3引数(例のv_cur, 1, v_smpid)までを指定します。
3. 第4引数(例のcolumn_errorの値)を使用している場合は、対象の変数(例のerrcd)を使用している箇所を確認します。
4. 確認した箇所で判定などの処理を行っている場合は、判定する際の値を以下の通り修正します。
   - 1406 ⇒ 22001
   - 1405 ⇒ 22002

(4) CLOSE_CURSOR

Oracleデータベースにおける記述形式

DBMS_SQL.CLOSE_CURSOR(第1引数)

記述差異

Oracleデータベース

クローズすると、引数に指定したカーソルがNULL値になります。

Fujitsu Enterprise Postgres

クローズすると、DBMS_SQL.CLOSE_CURSORの復帰値がNULL値になりますので、代入文に指定したカーソルで受け取ります。

移行手順

以下の手順で移行してください。

1. “DBMS_SQL.CLOSE_CURSOR”というキーワードでストアドプロシジャ内を検索し、呼び出し箇所を特定します。
2. DBMS_SQL.CLOSE_CURSORの呼び出し箇所を、値の代入(:=)に置き換え、カーソルがNULL値となるようにします。
   - 左辺にDBMS_SQL.CLOSE_CURSORに指定していた引数(例のv_cur)を指定します。
   - 右辺にDBMS_SQL.CLOSE_CURSORを記載します。引数は修正前と同じ値(例のv_cur)を指定します。

付録C　Oracleデータベースとの互換機能が利用するテーブル

Oracleデータベースとの互換機能が利用するテーブルについて説明します。

C.1　UTL_FILE.UTL_FILE_DIR

UTL_FILE.UTL_FILE_DIRテーブルはUTL_FILEパッケージが扱うディレクトリを登録します。

名前	型	説明
dir	text	UTL_FILEパッケージが扱うディレクトリ名

付録D　ECOBPG - COBOL言語による埋め込みSQL

COBOL言語による埋め込みSQLを使用するアプリケーション開発について説明します。

D.1　機能と操作における注意事項

埋め込みSQLプログラムは通常のプログラミング言語（ここではCOBOL）で記述されたコードで、特別にマークされたセクション内のSQLコマンドとともに使用されます。このプログラムを構築するには、まずソースコード (*.pco) を埋め込みSQLプリプロセッサに渡します。ソースコードは、プリプロセッサによって通常のCOBOLプログラム (*.cob) に変換され、その後COBOLコンパイラによって処理されます。(コンパイルとリンクの詳細については“D.9 埋め込みSQLプログラムの処理”を参照してください) 変換されたECOBPGアプリケーションは、libpqライブラリにある関数を埋め込みSQLライブラリ (ecpglib) を介して呼び出し、通常のフロントエンド・バックエンドプロトコルを使ってPostgreSQLサーバと通信します。

COBOLコードからSQLコマンドを扱う場合は、埋め込みSQLの方が他の手法よりも有効です。まず、埋め込みSQLはCOBOLプログラムの変数との面倒な双方間の情報移行を処理してくれます。さらに、プログラム内のSQLコードは構築時に正確な構文になっているかどうか検査されます。また、COBOLでの埋め込みSQLは標準SQLで既に定義されており、他の様々なSQLデータベースシステムでサポートされています。PostgreSQLの実装は可能な限りこの標準に準拠するよう設計されています。 また通常の場合、他のSQLデータベース用に作成された埋め込みSQLプログラムを比較的簡単にPostgreSQLへ移植することができます。

先に述べた通り、埋め込みSQLインタフェース用のプログラムは、通常のCOBOLプログラムに、データベース関連処理を行うための特別なコードを加えたものです。この特別なコードは、常に、次のような形式になっています。

```
EXEC SQL ... END-EXEC
```

このSQL文は、構文上でCOBOLの文の置き換えとなります。SQL文によりますが、データ部、または手続き部で記述することができます。実際の実行を伴うSQL文は手続き部で記述する必要があり、ホスト変数の宣言はデータ部で記述する必要がありますが、プレコンパイラでは正しい部、節に記述されているかは判定しません。埋め込みSQL文における大文字小文字の区別の有無は、COBOLコードではなく、通常のSQLコードの規則に従います。

COBOLコードの記法には固定形式または可変形式が使用できます。各行の6カラム目までは行番号領域、7カラム目は標識領域となります。また、埋め込みSQLコードはB領域(12カラム目以降)に位置している必要があります。

ただし、本マニュアル内のサンプルコードは行番号や領域のためのインデントを省略しています。

ECOBPGは、COBOLコードの記法に従って処理およびプログラムの出力を行います。COBOLコードの記法はecobpgコマンドで指定します。ただし、以下の制限があります。

- 固定形式の場合、12カラムから72カラムがB領域になります。73カラム以降の文字はプレコンパイルされたソースでは削除されます。
- 可変形式の場合、12カラムからそのレコードの最後(最大251カラム)までがB領域になります。252カラム以降の文字はプレコンパイルされたソースでは削除されます。

ECOBPGはCOBOLの文を可能な限り受け入れます。しかし、以下の制限があります。

- ホスト変数を宣言するセクションでは、デバッグ行を使用できません。
- ホスト変数を宣言するセクション以外では、デバッグ行を使用できますが、デバッグ行の中にSQLは使用できません。
- ホスト変数を宣言するセクションでは、区切り文字としてコンマやセミコロンを使用できません。代わりにスペースを使用してください。
- EcpgでサポートされているEXEC SQL VAR コマンドは、ECOBPGでは使用できません。代わりに、COBOLのREDEFINE句を使用してください。

以下の節で、すべての埋め込みSQL文について説明します。

D.2　データベース接続の管理

この節では、データベース接続の開始、終了、および切り替え方について解説しています。

D.2.1 データベースサーバへの接続

以下のSQL文を使用して、データベースへ接続します。

```
EXEC SQL CONNECT TO target [AS connection-name] [USER user-name] END-EXEC.
```

*target*は以下の方法で指定されます。

- *dbname*[@*hostname*][:*port*]
- tcp:postgresql://*hostname*[:*port*][/*dbname*][?*options*]
- unix:postgresql://*hostname*[:*port*][/*dbname*][?*options*]
- 上記形式のいずれかを含むSQL規約の文字列定数
- 上記形式のいずれかを含む文字変数への参照
- DEFAULT

接続対象をリテラル(つまり、変数を参照しない形)で指定し、その値を引用符でくくらなかった場合、大文字小文字の区別に関して通常のSQLの規則が適用されます。また、この場合、必要に応じて個々のパラメータを二重引用符で別々にくくることもできます。実際には、おそらく(単一引用符でくくられた)文字列リテラルもしくは変数の参照を使用した方がエラーをより防止することができます。DEFAULT接続対象は、デフォルトデータベース、デフォルトのユーザ名で接続を初期化します。この場合は、ユーザ名と接続名を分けて指定することができません。

ユーザ名を指定するには、別の方法もあります。

- *username*
- *username*/*password*
- *username* IDENTIFIED BY *password*
- *username* USING *password*

これまで同様、*username*と*password*は、SQL識別子、SQL文字列リテラル、文字型変数への参照を取ることができます。

1つのプログラム内で複数の接続を処理する場合には、*connection-name*を使用します。プログラムで1つしか接続を使わない場合は省略してかまいません。最も最近に開かれた接続が現在の接続になり、SQL文を実行しようとする時にデフォルトでこの接続が使用されます(本章の後で説明します)。

以下にCONNECT文について、数例を示します。

```
EXEC SQL CONNECT TO mydb@sql.mydomain.com END-EXEC.
```

```
EXEC SQL CONNECT TO unix:postgresql://sql.mydomain.com/mydb AS myconnection USER john END-EXEC.
```

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 TARGET PIC X(25).
01 USER PIC X(5).
EXEC SQL END DECLARE SECTION END-EXEC.
 ...
MOVE "mydb@sql.mydomain.com" TO TARGET.
MOVE "john" TO USER.
EXEC SQL CONNECT TO :TARGET USER :USER END-EXEC.
```

最後の形式では、文字変数参照として上を参照する変数を使用しています。このため、固定長文字列(つまり、VARYINGを指定しない)変数しか使用できません。後続のスペースは無視されます。後の節で、接頭辞にコロンを持つ場合のSQL文内でのCOBOL変数の使用方法について説明します。

接続対象の書式は標準SQLでは規定されていないことに注意してください。そのため、移植可能なアプリケーションを開発したいのであれば、上の例の最後の方法を参考にして、接続対象文字列をどこかにカプセル化してください。

D.2.2 接続の選択

前項で示したSQL文は現在の接続、つまり、最も最近に開いた接続上で実行されます。複数の接続を管理する必要があるアプリケーションでは、これを処理する2つの方法があります。

1つ目の選択肢は、各SQL文で明示的に接続を選択することです。以下に例を示します。

```
EXEC SQL AT connection-name SELECT ... END-EXEC.
```

アプリケーションが複数の接続を不特定な順番で使用する必要がある場合、この選択肢は特に適しています。

2番目の選択肢は、現在の接続を切り替えるSQL文を実行することです。以下のSQL文です。

```
EXEC SQL SET CONNECTION connection-name END-EXEC.
```

多くのSQL文を同一接続に対して使用する場合、この選択肢は特に便利です。

以下に複数のデータベースコネクションを管理しているプログラムの例を示します。

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
    01 DBNAME PIC X(7).
EXEC SQL END DECLARE SECTION END-EXEC.

    EXEC SQL CONNECT TO testdb1 AS con1 USER testuser END-EXEC.
    EXEC SQL CONNECT TO testdb2 AS con2 USER testuser END-EXEC.
    EXEC SQL CONNECT TO testdb3 AS con3 USER testuser END-EXEC.

*   This query would be executed in the last opened database "testdb3".
    EXEC SQL SELECT current_database() INTO :DBNAME END-EXEC.
    DISPLAY "current=" DBNAME " (should be testdb3)".
*   Using "AT" to run a query in "testdb2"
    EXEC SQL AT con2 SELECT current_database() INTO :DBNAME END-EXEC.
    DISPLAY "current=" DBNAME " (should be testdb2)".
*   Switch the current connection to "testdb1".
    EXEC SQL SET CONNECTION con1 END-EXEC.

    EXEC SQL SELECT current_database() INTO :DBNAME END-EXEC.
    DISPLAY "current=" DBNAME " (should be testdb1)".

    EXEC SQL DISCONNECT ALL END-EXEC.
```

この例は、次のような出力を生成します。

```
current=testdb3 (should be testdb3)
current=testdb2 (should be testdb2)
current=testdb1 (should be testdb1)
```

D.2.3 接続の切断

接続を閉じるには以下のSQL文を使用します。

```
EXEC SQL DISCONNECT [connection] END-EXEC.
```

*connection*は以下の方法で指定されます。

- *connection-name*
- DEFAULT
- CURRENT
- ALL

接続名の指定がなければ、現在の接続が閉じられます。

アプリケーションでは、過去に開いたすべての接続を明示的に閉じることを推奨します。

D.3 SQLコマンドの実行

すべてのSQLコマンドは、埋め込みSQLアプリケーション内で実行できます。以下に例をいくつか示します。

D.3.1 SQL文の実行

テーブルの作成:

```
EXEC SQL CREATE TABLE foo (number integer, ascii char(16)) END-EXEC.
EXEC SQL CREATE UNIQUE INDEX num1 ON foo(number) END-EXEC.
EXEC SQL COMMIT END-EXEC.
```

行の挿入:

```
EXEC SQL INSERT INTO foo (number, ascii) VALUES (9999, 'doodad') END-EXEC.
EXEC SQL COMMIT END-EXEC.
```

行の削除:

```
EXEC SQL DELETE FROM foo WHERE number = 9999 END-EXEC.
EXEC SQL COMMIT END-EXEC.
```

行の更新:

```
EXEC SQL UPDATE foo
    SET ascii = 'foobar'
    WHERE number = 9999 END-EXEC.
EXEC SQL COMMIT END-EXEC.
```

単一の行を返すSELECT文は、同様に EXEC SQL を用いて直接実行することができます。複数行の結果セットを扱うためには、アプリケーションはカーソルを使わなければなりません。“D.3.2 カーソルの使用”を参照してください。(特殊なケースでは、アプリケーションは複数の行をホスト変数の配列に一度に読み込むことができます。“配列”を参照してください)

単一行の検索:

```
EXEC SQL SELECT foo INTO :FooBar FROM table1 WHERE ascii = 'doodad' END-EXEC.
```

同様に、設定パラメータは SHOW コマンドによって取得することができます:

```
EXEC SQL SHOW search_path INTO :var END-EXEC.
```

:somethingという形のトークンはホスト変数です。つまり、COBOLプログラム内の変数を参照するものです。これについては“D.4 ホスト変数の使用”で説明します。

D.3.2 カーソルの使用

複数行の結果セットを受け取るためには、アプリケーションはカーソルを定義し、必要に応じてレコードを一行ずつ取り込む必要があります。カーソルを使った処理は、カーソルの宣言、カーソルのオープン、カーソルからのFETCH、カーソルのクローズという流れになります。

カーソルを用いたSELECT:

```
EXEC SQL DECLARE foo_bar CURSOR FOR
    SELECT number, ascii FROM foo
    ORDER BY ascii END-EXEC.
EXEC SQL OPEN foo_bar END-EXEC.
EXEC SQL FETCH foo_bar INTO :FooBar, :DooDad END-EXEC.
...
EXEC SQL CLOSE foo_bar END-EXEC.
EXEC SQL COMMIT END-EXEC.
```

カーソルの宣言の詳細については“D.11.4 DECLARE”を、FETCHコマンドの詳細については“PostgreSQL Documentation”の“Reference”の“SQL Commands”を参照してください。

注意: ECOBPGのDECLAREコマンド自身は、PostgreSQLバックエンドに送られるSQL文を実行しません。OPENコマンドが実行された段階で、バックエンド内部で(DECLAREコマンドで宣言された)カーソルが開かれます。

D.3.3 トランザクションの管理

デフォルトモードでは、SQL文はEXEC SQL COMMITが発行されることによってのみコミットされます。埋め込みSQLインタフェースでも、ecobpgコマンドの-tコマンドラインオプション、あるいはEXEC SQL SET AUTOCOMMIT TO ON文によって(libpqの振舞いに似た)トランザクションの自動コミットをサポートしています。自動コミットモードでは、問い合わせが明示的なトランザクションブロックの内部にある場合を除き、すべての問い合わせが自動的にコミットされます。自動コミットモードは、EXEC SQL SET AUTOCOMMIT TO OFFを使用して明示的に無効にすることができます。

参照

ecobpgの-tコマンドラインオプションに関しては、“PostgreSQL Documentation”の“PostgreSQL Client Applications”の“ecpg”を参照してください。

以下のトランザクション管理コマンドを使用することができます:

EXEC SQL COMMIT END-EXEC.

実行中のトランザクションのコミット。

EXEC SQL ROLLBACK END-EXEC.

実行中のトランザクションのロールバック。

EXEC SQL SET AUTOCOMMIT TO ON END-EXEC.

自動コミットモードの有効化。

SET AUTOCOMMIT TO OFF END-EXEC.

自動コミットモードの無効化。デフォルト状態。

D.3.4 Prepared Statements

SQL文に渡す値がコンパイル時に決まらない場合、または同じSQL文を何度も実行する場合、プリペアード・ステートメントが便利です。

SQL文は、PREPAREコマンドを使ってプリペアします。まだ決まっていない値については、プレースホルダ "?" を使います:

```
EXEC SQL PREPARE stmt1 FROM "SELECT oid, datname FROM pg_database WHERE oid = ?" END-EXEC.
```

SQL文が一行のみの結果を返却する場合には、アプリケーションはSQL文を PREPARE した後、USING を用いてプレースホルダに実際の値を与えて EXECUTE を実行することができます。

```
EXEC SQL EXECUTE stmt1 INTO :dboid, :dbname USING 1 END-EXEC.
```

SQL文が複数の行を返却する場合には、アプリケーションはプリペアード・ステートメントの宣言に対応したカーソルを利用することができます。入力パラメータを設定するために、カーソルは USING とともに開かれなければなりません:

```
EXEC SQL PREPARE stmt1 FROM "SELECT oid,datname FROM pg_database WHERE oid > ?" END-EXEC.
EXEC SQL DECLARE foo_bar CURSOR FOR stmt1 END-EXEC.
* when end of result set reached, break out of while loop
EXEC SQL WHENEVER NOT FOUND GOTO FETCH-END END-EXEC.
EXEC SQL OPEN foo_bar USING 100 END-EXEC.
...
PERFORM NO LIMIT
    EXEC SQL FETCH NEXT FROM foo_bar INTO :dboid, :dbname END-EXEC
END-PERFORM.

FETCH-END.
EXEC SQL CLOSE foo_bar END-EXEC.
```

プリペアード・ステートメントをこれ以上必要としなくなったら、解放処理をしなければなりません:

```
EXEC SQL DEALLOCATE PREPARE name END-EXEC.
```

PREPARE についての詳細は“D.11.10 PREPARE”を参照してください。また、プレースホルダと入力パラメータの利用についての詳細は “D.5 動的SQL”を参照してください。

D.4 ホスト変数の使用

“D.3 SQLコマンドの実行”では、埋め込みSQLプログラムでどのようにSQL文を実行するのかについて説明しました。このSQL文の中には固定値しか使用しないものや、ユーザが指定する値をSQL文の中に挿入する手段を提供しないもの、問い合わせが返す値をプログラムで処理する手段を提供しないものがありました。この種のSQL文は実際のアプリケーションでは役に立ちません。本節では、ホスト変数という単純な機構を使用した、COBOLプログラムと埋め込みSQL文との間でデータをやり取りする方法を詳細に説明します。埋め込みSQLプログラムでは、SQL文をホスト言語となるCOBOLプログラムコードにおけるゲストguestsとみなします。したがって、COBOLプログラムの変数はホスト変数と呼ばれます。

PostgreSQLバックエンドとECOBPGアプリケーションの間で値をやり取りするその他の方法は、“D.6 記述子領域の使用”で説明されているSQL記述子を使う方法です。

D.4.1 概要

埋め込みSQLにおけるCOBOLプログラムとSQL文との間でのデータのやり取りは特に単純です。値に適切な引用符を付与するといった、様々な複雑な処理を伴う、プログラムにデータを文中に貼り付けさせるという方法はなく、単にSQL文の中に、先頭にコロンを付けたCOBOL変数名を書くだけです。以下に例を示します。

```
EXEC SQL INSERT INTO sometable VALUES (:v1, 'foo', :v2) END-EXEC.
```

このSQL文は、v1とv2という2つのCOBOL変数を参照し、また、通常のSQL文字列リテラルも使用しています。これは、使用できるデータの種類は1つだけという制限がないことを表しています。

SQL文内にCOBOLの変数を挿入するこの様式は、SQL文で値式が想定されている所であればどこでも動作します。

D.4.2 宣言セクション

例えば問い合わせ内のパラメータとして、プログラムからデータベースへデータを渡す、もしくは、データベースからプログラムへデータを渡すためには、このようなデータを含むように意図されたCOBOL変数を、埋め込みSQLプリプロセッサが管理できるように、特殊な印のついたセクションで宣言する必要があります。

このセクションは以下で始まります。

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
```

そして、以下で終わります。

```
EXEC SQL END DECLARE SECTION END-EXEC.
```

この行の間は、以下のような通常のCOBOL変数宣言でなければなりません。

```
01 INTX PIC S9(9) COMP VALUE 4.
01 FOO PIC X(15).
01 BAR PIC X(15).
```

見てわかるとおり、省略可能ですが、変数に初期値を代入することができます。変数のスコープはプログラム内の宣言セクションの位置により決まります。

プログラム内に複数の宣言セクションを持たせることができます。

また、宣言は普通のCOBOL変数としてそのまま出力ファイルに出力されます。ですので、これらを再度宣言する必要はありません。通常、SQLコマンドで使用する予定がない変数はこの特別なセクションの外側で宣言されます。

集団項目の定義もまた、DECLAREセクションの内側で表す必要があります。さもないと、プリプロセッサはその定義が不明であるために、これらの型を扱うことができません。

D.4.3 クエリ実行結果の受け取り

ここまでで、プログラムで生成したデータをSQLコマンドに渡すことができるようになりました。しかし、どのように問い合わせの結果を取り出すのでしょうか？ この目的のために、埋め込みSQLでは、通常のSELECTとFETCHを派生した、特殊なコマンドを提供しています。これらのコマンドは特別なINTO句を持ち、ここで返された値をどのホスト変数に格納すればよいかを指定します。SELECT は単一行を返却するクエリに使用され、FETCH は複数の行を返却するクエリにおいてカーソルとともに使用されます。

以下にサンプルを示します。

```
*
* assume this table:
* CREATE TABLE test (a int, b varchar(50));
*

 EXEC SQL BEGIN DECLARE SECTION END-EXEC.
 01 V1 PIC S9(9).
 01 V2 PIC X(50) VARYING.
 EXEC SQL END DECLARE SECTION END-EXEC.
 ...
 EXEC SQL SELECT a, b INTO :V1, :V2 FROM test END-EXEC.
```

INTO句が選択リストとFROM句の間に現れます。選択リスト内の要素数とINTO直後のリスト（目的リストとも呼ばれます）の要素数は等しくなければなりません。

以下にFETCHコマンドの使用例を示します。

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 V1 PIC S9(9).
01 V2 PIC X(50) VARYING.
EXEC SQL END DECLARE SECTION END-EXEC.

 ...

EXEC SQL DECLARE foo CURSOR FOR SELECT a, b FROM test END-EXEC.

 ...

PERFORM WITH
    ...
    EXEC SQL FETCH NEXT FROM foo INTO :V1, :V2 END-EXEC
    ...
END-PERFORM.
```

ここでは、INTO句が通常のすべての句の後ろに現れています。

D.4.4 データ型の対応

ECOBPGアプリケーションがPostgreSQLバックエンドとCOBOLアプリケーションの間で値をやり取りする際、例えばサーバからクエリの結果を受け取る、または入力パラメータとともにSQL文を実行する場合、それらの値はPostgreSQLのデータ型とホスト言語の変数の型（具体的にはCOBOLのデータ型）の間で変換される必要があります。ECOBPGの重要な点のひとつは、ほとんどの場合においてECOBPGがこれらを自動的に扱うということです。

この点において、2つのデータ型があります: いくつかの単純なPostgreSQLのデータ型、integer や text などは、アプリケーションから直接読んだり書いたりすることができます。その他のPostgreSQLのデータ型、timestamp や date などは、文字列を通してアクセスすることしかできません。特別なライブラリの関数は、ecobpgには存在していません。（pgtypesはecpgには存在しますが、ecobpgには存在しません。）

“表D.1 PostgreSQLデータ型とCOBOL変数型の対応”には、PostgreSQLのどのデータ型がCOBOLのデータ型に対応するかを示されています。与えられたPostgreSQLのデータ型へ値を書き込みまたは読み込みしたい場合には、対応するCOBOLのデータ型の変数を宣言セクションにおいて宣言しなければなりません。

表D.1 PostgreSQLデータ型とCOBOL変数型の対応

PostgreSQL data type	COBOL Host variable type
smallint	PIC S9([1-4]) {BINARY|COMP|COMP-5}
integer	PIC S9([5-9]) {BINARY|COMP|COMP-5}
bigint	PIC S9([10-18]) {BINARY|COMP|COMP-5}
decimal	PIC S9(m)V9(n) PACKED-DECIMAL PIC 9(m)V9(n) DISPLAY (*1) PIC S9(m)V9(n) DISPLAY PIC S9(m)V9(n) DISPLAY SIGN TRAILING [SEPARATE] PIC S9(m)V9(n) DISPLAY SIGN LEADING [SEPARATE]
numeric	decimalと同じ
real	COMP-1
double precision	COMP-2
small serial	PIC S9([1-4]) {BINARY|COMP|COMP-5}
serial	PIC S9([5-9]) {BINARY|COMP|COMP-5}
bigserial	PIC S9([10-18]) {BINARY|COMP|COMP-5}
oid	PIC 9(9) {BINARY|COMP|COMP-5}
character(*n*), varchar(*n*), text	PIC X(*n*), PIC X(*n*) VARYING PIC N(*n*), PIC N(*n*) VARYING
name	PIC X(NAMEDATALEN)
boolean	BOOL(*2)
bytea	BYTEA(n)
other types (例:timestamp)	PIC X(*n*), PIC X(*n*) VARYING PIC N(*n*), PIC N(*n*) VARYING

*1 : USAGE句を指定しない場合、ホスト変数は、DISPLAYとみなされます。
*2 :データ型の定義は、プレコンパイル時に自動的に追加されます。
BOOLの中身はPIC X(1)です。‘1’は真、‘0’は偽を表します。

表から分かるように、整数の桁に複数のパターンを使用できますが、指定した桁数よりも大きな桁数をもつ数をデータベースが返す場合の振る舞いは定義されていません。

VALUE句ではVARYINGを使用することはできません。(other typesでは使用できます。)
REDEFINE句は使用することができますが、プレコンパイルにおいて評価はされません。(COBOLコンパイラが行います。)

文字列の処理

varcharやtextのような文字列のデータ型を扱うためのホスト変数を宣言する方法があります。

ECOBPGによって提供される特殊なデータ型 PIC X(n) VARYING(以下、VARCHAR型と呼びます) を使う方法です。

VARCHAR の定義は、すべての変数が名前の付いた集団項目に変換されます。以下のような宣言は:

```
01 VAR PIC X(180) VARYING.
```

次のように変換されます:

```
01 VAR.
49 LEN PIC S9(4) COMP-5.
49 ARR PIC X(180).
```

--varchar-with-named-memberオプションを使用すると、以下のように変換されます。

```
01 VAR.
49 VAR-LEN PIC S9(4) COMP-5.
49 VAR-ARR PIC X(180).
```

VHARCHAR型のレベル番号には、1から48を使用することができます。VARCHAR型の変数の直後にレベル番号49をもつ変数を使用しないでください。

SQL文への入力用にVARCHAR型のホスト変数を利用する場合、LENにはARRに含まれる文字列の長さを格納する必要があります。

SQL文からの出力用にVARCHAR型のホスト変数を利用する場合、十分な長さのホスト変数を宣言する必要があります。不足している場合、バッファオーバーランの原因となります。

PIC X(n) と VARCHAR ホスト変数は、他のSQLのデータ型の値を文字列表現として保持することもできます。

特殊なデータ型へのアクセス

ECOBPGでは、date型、timestamp型、interval型に対して、特別なサポートはしていません。(ECPGにはpgtypesがありますが、ECOBPGにはありません。)

PIC X(n)やVARCHARホスト変数を利用して文字列表現のデータを入出力に使用することが可能です。データの文字列表現については“PostgreSQL Documentation”の“ Data Types”を参照してください。

bytea

bytea型の扱いは、VARCHARと似ています。bytea型の配列の定義は、すべての変数が名前の付いた集団項目に変換されます。

以下のような宣言は:

```
01 var bytea(100).
```

次のように変換されます:

```
01 var .
49 LEN PIC S9(9) COMP-5 .
49 ARR PIC X(100) .
```

--bytea-with-named-memberオプションを使用すると、以下のように変換されます。

```
01 var .
49 var-LEN PIC S9(9) COMP-5 .
49 var-ARR PIC X(100) .
```

データ項目ARRはバイナリフォーマットデータを保持します。VARCHARとは異なり、データ処理時にロケールや文字の符号化方式の影響は受けません。

その他の使用方法、禁則事項はVARCHARと同じです。

bytea変数は、bytea_outputがhexに設定されている場合にのみ使うことができます。

非プリミティブ型のホスト変数

ホスト変数として、配列、TYPEDEF、集団項目も使うことができます。

配列

COBOLで提供されているOCCURRENCE構文を使用して、配列型の変数を作成、使用することができます。

カーソルを用いずに複数行を返却するクエリ結果を受け取るために使います。配列を使わない場合、複数行からなるクエリの実行結果を処理するには、カーソルとFETCHコマンドを使用する必要があります。しかし、配列のホスト変数を使うと、複数行を一括して受け取ることができます。配列の長さはすべての行を受け入れられるように定義されなければなりません。でなければバッファーオーバーランが発生するでしょう。

以下の例はpg_databaseシステムテーブルをスキャンし、利用可能なデータベースのすべてのOIDとデータベース名を表示します:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 GROUP-ITEM.
  05 DBID PIC S9(9) COMP OCCURS 8.
  05 DBNAME PIC X(16) OCCURS 8.
01 I PIC S9(9) COMP.
EXEC SQL END DECLARE SECTION END-EXEC.
  EXEC SQL CONNECT TO testdb END-EXEC.
* Retrieve multiple rows into arrays at once.
  EXEC SQL SELECT oid,datname INTO :DBID, :DBNAME FROM pg_database END-EXEC.

  PERFORM VARYING I FROM 1 BY 1 UNTIL I > 8
    DISPLAY "oid=" DBID(I) ", dbname=" DBNAME(I)
  END-PERFORM.

  EXEC SQL COMMIT END-EXEC.
  EXEC SQL DISCONNECT ALL END-EXEC.
```

配列のサブスクリプトを特定することにより、配列のメンバを、単純なホスト変数として使用することができます。サブスクリプトを特定するために、COBOLの形式である“(1)”ではなく、Cの形式である“[1]”を使用してください。したし、サブスクリプトは、COBOLの文法に従い、1から開始します。

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 GROUP-ITEM.
   05 DBID PIC S9(9) COMP OCCURS 8.
EXEC SQL END DECLARE SECTION END-EXEC.
   EXEC SQL CONNECT TO testdb END-EXEC.
   EXEC SQL SELECT oid INTO :DBID[1] FROM pg_database WHERE oid=1 END-EXEC.
      DISPLAY "oid=" DBID(1)
   EXEC SQL COMMIT END-EXEC.
   EXEC SQL DISCONNECT ALL END-EXEC.
```

集団項目

従属項目の名前がクエリ結果のカラム名に合致する集団項目は、複数のカラムを一括して受け取るために利用することができます。集団項目は複数のカラムの値を単一のホスト変数で扱うことを可能にします。

以下の例は、pg_databaseシステムテーブルおよびpg_database_size()関数を使って、利用可能なデータベースのOID、名前、サイズを取得します。この例では、メンバー変数の名前がSELECT結果の各カラムに合致する集団項目DBINFO_Tが、複数のホスト変数に格納することなくFETCH文の一行の結果を受け取るために使用されています。

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
   01 DBINFO-T TYPEDEF.
      02 OID PIC S9(9) COMP.
      02 DATNAME PIC X(65).
      02 DBSIZE PIC S9(18) COMP.

   01 DBVAL TYPE DBINFO-T.
EXEC SQL END DECLARE SECTION END-EXEC.
    EXEC SQL DECLARE cur1 CURSOR FOR SELECT oid, datname, pg_database_size(oid) AS size
FROM pg_database END-EXEC.
    EXEC SQL OPEN cur1 END-EXEC.
```

```
* when end of result set reached, break out of loop
    EXEC SQL WHENEVER NOT FOUND GOTO END-FETCH END-EXEC.
    PERFORM NO LIMIT
*       Fetch multiple columns into one structure.
       EXEC SQL FETCH FROM cur1 INTO :DBVAL END-EXEC
*       Print members of the structure.
       DISPLAY "oid=" OID ", datname=" DATNAME ", size=" DBSIZE
    END-PERFORM.
    END-FETCH.
EXEC SQL CLOSE cur1 END-EXEC.
```

集団項目のホスト変数は、多数のカラムを集団項目の従属項目として“吸収”します。追加のカラムは他のホスト変数に割り当てることができます。例えば、上記のプログラムは集団項目に含まれない size 変数を使って以下のように書き換えることができます。

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
    01 DBINFO-T TYPEDEF.
        02 OID PIC S9(9) COMP.
        02 DATNAME PIC X(65).

    01 DBVAL TYPE DBINFO-T.
    01 DBSIZE PIC S9(18) COMP.
EXEC SQL END DECLARE SECTION END-EXEC.

    EXEC SQL DECLARE cur1 CURSOR FOR SELECT oid, datname, pg_database_size(oid) AS size
FROM pg_database END-EXEC.
    EXEC SQL OPEN cur1 END-EXEC.

    * when end of result set reached, break out of loop
    EXEC SQL WHENEVER NOT FOUND GOTO END-FETCH END-EXEC.

    PERFORM NO LIMIT
*       Fetch multiple columns into one structure.
        EXEC SQL FETCH FROM cur1 INTO :DBVAL, :DBSIZE END-EXEC

*       Print members of the structure.
        DISPLAY "oid=" OID ", datname=" DATNAME ", size=" DBSIZE
    END-PERFORM

    FETCH-END.
    EXEC SQL CLOSE cur1 END-EXEC.
```

ホスト変数のSQL文には、ネストされていない集団項目しか使用することはできません。ネストされた集団項目の宣言はOKですが、SQLに対しては集団項目のネストされていない部分を指定しなければなりません。(プリコンパイル時に集団項目に変換されるVARCHAR型は、このルールに従っているとは見なされません。)SQLにおける集団項目における内部の項目を使用するときは、COBOLのＡOF B構文ではなく、ピリオドで区切られたCの構造体のように使用してください。以下に例を示します。

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 NESTED-GROUP.
    02 CHILD1.
        03 A PIC X(10).
        03 B PIC S9(9) COMP.
    02 CHILD2.
        03 A PIC X(10).
        03 B PIC S9(9) COMP.
EXEC SQL END DECLARE SECTION END-EXEC.

* This SQL is valid. CHILD1 has no nested group items.
EXEC SQL SELECT * INTO :NESTED-GROUP.CHILD1 FROM TABLE1 END-EXEC.
```

集団項目の基本項目を特定するために、その基本項目を一意にするために十分な特定を行えば、完全な特定は必要ではありません。詳細は、COBOLの文法資料を参照してください。

TYPEDEF

新しい型と既存の型を対応付けるためには TYPEDEF キーワードを使ってください。

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
    01 MYCHARTYPE TYPEDEF PIC X(40).
    01 SERIAL-T TYPEDEF PIC S9(9) COMP.
EXEC SQL END DECLARE SECTION END-EXEC.
```

また、同様に以下を使うこともできます:

```
EXEC SQL TYPE SERIAL-T IS PIC S9(9) COMP-5. END-EXEC.
```

この宣言は、宣言セクションの一部である必要はありません。

D.4.5 非プリミティブSQLデータ型の扱い方

本節では、非スカラー型およびユーザ定義のSQLデータ型をECOBPGアプリケーションで扱う方法を示します。この内容は、前の項で説明した非プリミティブ型のホスト変数の扱い方とは別のものです。

配列

SQLの配列は、ECOBPGにおいては直接的にはサポートされていません。SQL配列をCOBOLの配列のホスト変数に簡単に対応させることはできません。定められていない振る舞いを引き起こすでしょう。但し、いくつかの回避策があります。

もし、クエリが配列の *要素* に対して個別にアクセスした場合、ECOBPGにおける配列の利用を避けることができます。その際、要素に対応させることができる型のホスト変数を利用しなければなりません。例えば、カラムの型が integer の配列の場合、PIC S9(9) COMP 型のホスト変数を使用することができます。同様に、要素の型が varchar または text の場合、VARCHAR 型のホスト変数を使用することができます。

以下に例を示します。次のようなテーブルを仮定します:

```
CREATE TABLE t3 (
ii integer[]
);
testdb=> SELECT * FROM t3;
ii
-------------
{1,2,3,4,5}
(1 row)
```

以下のプログラム例は、配列の4番目の要素を取得し、それを PIC S9(9) COMP-5 型のホスト変数に保存します:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 II PIC S9(9) COMP.
EXEC SQL END DECLARE SECTION END-EXEC.

EXEC SQL DECLARE cur1 CURSOR FOR SELECT ii[4] FROM t3 END-EXEC.
EXEC SQL OPEN cur1 END-EXEC.

EXEC SQL WHENEVER NOT FOUND GOTO END-FETCH END-EXEC.

PERFORM NO LIMIT
    EXEC SQL FETCH FROM cur1 INTO :II  END-EXEC
    DISPLAY "ii=" II
END-PERFORM.
END-FETCH.
EXEC SQL CLOSE cur1 END-EXEC.
```

複数の配列の要素を、配列型のホスト変数の複数の要素にマッピングするためには、配列型のカラムの各要素とホスト変数配列の各要素は、以下の例のように別々に管理されなければなりません。以下に例を示します。

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 GROUP-ITEM.
    05 II_A PIC S9(9) COMP OCCURS 8.
EXEC SQL END DECLARE SECTION END-EXEC.

EXEC SQL DECLARE cur1 CURSOR FOR SELECT ii[1], ii[2], ii[3], ii[4] FROM t3 END-EXEC.
EXEC SQL OPEN cur1 END-EXEC.

EXEC SQL WHENEVER NOT FOUND GOTO END-FETCH END-EXEC.

PERFORM NO LIMIT
    EXEC SQL FETCH FROM cur1 INTO :II_A[1], :II_A[2], :II_A[3], :II_A[4] END-EXEC
    ...
END-PERFORM.
```

繰り返しになりますが、以下の例は正しく動作しません。なぜなら、配列型のカラムをホストの配列変数に直接対応させることはできないからです。

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 GROUP-ITEM.
    05 II_A PIC S9(9) COMP OCCURS 8.
EXEC SQL END DECLARE SECTION END-EXEC.

EXEC SQL DECLARE cur1 CURSOR FOR SELECT ii FROM t3 END-EXEC.
EXEC SQL OPEN cur1 END-EXEC.

EXEC SQL WHENEVER NOT FOUND GOTO END-FETCH END-EXEC.
PERFORM NO LIMIT
*     WRONG
    EXEC SQL FETCH FROM cur1 INTO :II_A END-EXEC
    ...
END-PERFORM.
```

もうひとつの回避策は、配列をホスト変数のVARCHAR型に文字列表現として保存することです。この表現についてより詳細があります。

参照

この表現の詳細については、“PostgreSQL Documentation”の“Tutorial”の“Arrays”を参照してください。

これは、配列にはホストプログラム内で自然な形ではアクセスできないことを意味しています(文字列表現を解析する追加処理が無ければですが)。

複合型

複合型はECOBPGでは直接はサポートされていませんが、簡単な回避方法が利用可能です。利用可能な回避方法は、先に配列において説明されたものと似ています。つまり、各属性に個別にアクセスするか、外部の文字列表現を使います。

以降の例のため、以下の型とテーブルを仮定します:

```
CREATE TYPE comp_t AS (intval integer, textval varchar(32));
CREATE TABLE t4 (compval comp_t);
INSERT INTO t4 VALUES ( (256, 'PostgreSQL') );
```

もっとも分かりやすい解決法は、各属性に個別にアクセスすることです。以下のプログラムは、comp_t型の各要素を個別に選択することによってサンプルのテーブルからデータを受け取ります:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 INTVAL PIC S9(9) COMP.
01 TEXTVAL PIC X(33) VARYING.
EXEC SQL END DECLARE SECTION END-EXEC.
```

```
* Put each element of the composite type column in the SELECT list.
EXEC SQL DECLARE cur1 CURSOR FOR SELECT (compval).intval, (compval).textval FROM t4 END-EXEC.
EXEC SQL OPEN cur1 END-EXEC.

EXEC SQL WHENEVER NOT FOUND GOTO END-FETCH END-EXEC.

PERFORM NO LIMIT
*    Fetch each element of the composite type column into host variables.
    EXEC SQL FETCH FROM cur1 INTO :INTVAL, :TEXTVAL END-EXEC

    DISPLAY "intval=" INTVAL ", textval=" ARR OF TEXTVAL
END-PERFORM.

END-FETCH.
EXEC SQL CLOSE cur1 END-EXEC.
```

この例を拡張して、FETCHコマンドの値を格納するホスト変数を一つの集団項目にまとめることができます。集団項目の形のホスト変数の詳細については、“集団項目”を参照してください。集団項目に変更するために、この例は以下のように変更することができます。二つのホスト変数intvalとtextvalをcomp_t集団項目の従属項目とし、集団項目をFETCHコマンドで指定します。

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 COMP-T TYPEDEF.
    02 INTVAL PIC S9(9) COMP.
    02 TEXTVAL PIC X(33) VARYING.

01 COMPVAL TYPE COMP-T.
EXEC SQL END DECLARE SECTION END-EXEC.

* Put each element of the composite type column in the SELECT list.
EXEC SQL DECLARE cur1 CURSOR FOR SELECT (compval).intval, (compval).textval FROM t4 END-EXEC.
EXEC SQL OPEN cur1 END-EXEC.
EXEC SQL WHENEVER NOT FOUND GOTO END-FETCH END-EXEC.

PERFORM NO LIMIT
*    Put all values in the SELECT list into one structure.
    EXEC SQL FETCH FROM cur1 INTO :COMPVAL END-EXEC

    DISPLAY "intval=" INTVAL ", textval=" ARR OF TEXTVAL
END-PERFORM.

END-FETCH.
EXEC SQL CLOSE cur1 END-EXEC.
```

集団項目がFETCHコマンドで使われていますが、属性名はSELECT句において各々が指定されています。これは、複合型の値のすべての属性を示す*を用いることで拡張することができます。

```
...
EXEC SQL DECLARE cur1 CURSOR FOR SELECT (compval).* FROM t4 END-EXEC.
EXEC SQL OPEN cur1 END-EXEC.
EXEC SQL WHENEVER NOT FOUND GOTO END-FETCH END-EXEC.

PERFORM NO LIMIT
*   Put all values in the SELECT list into one structure.
    EXEC SQL FETCH FROM cur1 INTO :COMPVAL END-EXEC

    DISPLAY "intval=" INTVAL ", textval=" ARR OF TEXTVAL
END-PERFORM.
```

この方法であれば、ECOBPGが複合型そのものを理解できないとしても、複合型はほぼシームレスに集団項目に対応させることができます。

最後に、VARCHAR 型のホスト変数に外部の文字列表現として複合型の値を格納することもできます。しかし、この方法ではホストプログラムから値のフィールドにアクセスするのは簡単ではありません。

ユーザ定義の基本型

新しいユーザ定義の基本型は、ECOBPGでは直接的にはサポートされていません。外部の文字列表現、VARCHAR 型のホスト変数を使うことができ、この解決法は多くの型について確かに適切かつ十分です。

以下に complex 型を使った例を示します。

参照

complex型の詳細は、“PostgreSQL Documentation”の“Server Programming”の“User-defined Type”を参照してください。

この型の外部文字列表現は (%lf,%lf) で、complex_in() 関数および complex_out() 関数で定義されています。以下の例は、カラム a と b に、complex 型の値 (1,1) および (3,3) を挿入し、その後、それらをテーブルからSELECTします。

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
    01 A PIC X(64) VARYING.
    01 B PIC X(64) VARYING.
EXEC SQL END DECLARE SECTION END-EXEC.

EXEC SQL INSERT INTO test_complex VALUES ('(1,1)', '(3,3)') END-EXEC.

EXEC SQL DECLARE cur1 CURSOR FOR SELECT a, b FROM test_complex END-EXEC.
EXEC SQL OPEN cur1 END-EXEC.

EXEC SQL WHENEVER NOT FOUND GOTO END-FETCH END-EXEC.

PERFORM NO LIMIT
    EXEC SQL FETCH FROM cur1 INTO :A, :B END-EXEC
    DISPLAY "a=" ARR OF A ", b=" ARR OF B
END-PERFORM.

END-FETCH.
EXEC SQL CLOSE cur1 END-EXEC.
```

その他の回避方法は、ユーザ定義型をECOBPGにおいて直接的に使うことを避けることであり、ユーザ定義型とECOBPGが扱えるプリミティブ型を変換する関数またはキャストを作成することです。ただし、型のキャスト、特に暗黙のものは型システムにおいて慎重に導入されなければなりません。

例を示します。

```
CREATE FUNCTION create_complex(r double precision, i double precision) RETURNS complex
LANGUAGE SQL
IMMUTABLE
AS $$ SELECT $1 * complex '(1,0)' + $2 * complex '(0,1)' $$;
```

この定義の後、以下の例は

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 A COMP-2.
01 B COMP-2.
01 C COMP-2.
01 D COMP-2.
EXEC SQL END DECLARE SECTION END-EXEC.

MOVE 1 TO A.
MOVE 2 TO B.
MOVE 3 TO C.
MOVE 4 TO D.
```

```
EXEC SQL INSERT INTO test_complex VALUES (create_complex(:A, :B), create_complex(:C, :D))
END-EXEC.
```

以下と同じ効果をもたらします。

```
EXEC SQL INSERT INTO test_complex VALUES ('(1,2)', '(3,4)') END-EXEC.
```

D.4.6 指示子

上の例ではNULL値を扱いません。実際、取り出し例では、もしデータベースからNULL値が取り出された場合にはエラーが発生します。データベースへNULL値を渡す、または、データベースからNULL値を取り出すためには、第二のホスト変数指定をデータを格納するホスト変数それぞれに追加しなければなりません。第二のホスト変数は*指示子*と呼ばれ、データがNULLかどうかを表すフラグが含まれます。NULLの場合、実際のホスト変数の値は無視されます。以下に、NULL値の取り出しを正しく扱う例を示します。

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 VAL PIC X(50) VARYING.
01 VAL_IND PIC S9(9) COMP-5.
EXEC SQL END DECLARE SECTION END-EXEC.
...
EXEC SQL SELECT b INTO :VAL :VAL_IND FROM test1 END-EXEC.
```

値がNULLでなければ、指示子変数VAL_INDは0となります。値がNULLならば負の値となります。

指示子は他の機能を持ちます。指示子の値が正ならば、値がNULLではありませんが、ホスト変数に格納する際に一部切り詰められたことを示します。

D.5 動的SQL

多くの場合、アプリケーションが実行しなければならないSQL文は、アプリケーションを作成する段階で決まります。しかし、中には、SQL文が実行時に構成されることや外部ソースで提供されることがあります。このような場合、SQL文を直接COBOLソースコードに埋め込むことはできません。しかし、文字列変数として提供される任意のSQL文を呼び出すことができる機能が存在します。

D.5.1 結果セットを伴わないSQL文の実行

任意のSQL文を実行するもっとも簡単な方法は、EXECUTE IMMEDIATE コマンドを使用することです。例を示します:

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 STMT PIC X(30) VARYING.
EXEC SQL END DECLARE SECTION END-EXEC.

MOVE "CREATE TABLE test1 (...);" TO ARR OF STMT.
COMPUTE LEN OF STMT = FUNCTION STORED-CHAR-LENGTH (ARR OF STMT).
EXEC SQL EXECUTE IMMEDIATE :STMT END-EXEC.
```

EXECUTE IMMEDIATE は結果セットを返却しないSQL文に使用することができます(e.g., DDL, INSERT, UPDATE, DELETE)。結果を受け取るSQL文(e.g., SELECT)をこの方法で実行することはできません。次の節で、その実行方法を説明します。

D.5.2 入力パラメータを伴うSQL文の実行

任意のSQL文を実行するより強力な方法は、一度プリペアをし、その後でプリペアード・ステートメントを実行したいところで実行することです。また、SQL文を汎用化した形でプリペアし、パラメータを置き換えることで特定のSQL文を実行させることも可能です。SQL文をプリペアする時、後でパラメータとして置き換えたいところには疑問符を記述してください。以下に例を示します。

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 STMT PIC X(40) VARYING.
EXEC SQL END DECLARE SECTION END-EXEC.
```

```
MOVE "INSERT INTO test1 VALUES(?, ?);" TO ARR OF STMT.
COMPUTE LEN OF STMT = FUNCTION STORED-CHAR-LENGTH (ARR OF STMT).
EXEC SQL PREPARE MYSTMT FROM :STMT END-EXEC.
 ...
EXEC SQL EXECUTE MYSTMT USING 42, 'foobar' END-EXEC.
```

プリペアード・ステートメントが必要なくなった時、割当てを解除しなければなりません。

```
EXEC SQL DEALLOCATE PREPARE name END-EXEC.
```

D.5.3 結果セットを返却するSQL文の実行

単一行を返却するSQL文を実行するには、EXECUTE を使うことができます。結果を保存するには、INTO 句を追加します。

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 STMT PIC X(50) VARYING.
01 V1 PIC S9(9) COMP.
01 V2 PIC S9(9) COMP.
01 V3 PIC X(50) VARYING.
EXEC SQL END DECLARE SECTION END-EXEC.

MOVE "SELECT a, b, c FROM test1 WHERE a > ?" TO ARR OF STMT.
COMPUTE LEN OF STMT = FUNCTION STORED-CHAR-LENGTH (ARR OF STMT).
EXEC SQL PREPARE MYSTMT FROM :STMT END-EXEC.
...
EXEC SQL EXECUTE MYSTMT INTO :V1, :V2, :V3 USING 37 END-EXEC.
```

EXECUTEコマンドはINTO句、USING句、この両方を持つことも、どちらも持たないこともできます。

クエリが2行以上の結果を返すことが想定される場合、以下の例のようにカーソルを使う必要があります。(カーソルの詳細については"D.3.2 カーソルの使用"を参照してください)

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 DBANAME PIC X(128) VARYING.
01 DATNAME PIC X(128) VARYING.
01 STMT PIC X(200) VARYING.
EXEC SQL END DECLARE SECTION END-EXEC.

MOVE "SELECT u.usename as dbaname, d.datname
-              " FROM pg_database d, pg_user u
-              " WHERE d.datdba = u.usesysid"
TO ARR OF STMT.
COMPUTE LEN OF STMT = FUNCTION STORED-CHAR-LENGTH (ARR OF STMT).

EXEC SQL CONNECT TO testdb AS con1 USER testuser END-EXEC.

EXEC SQL PREPARE STMT1 FROM :STMT END-EXEC.

EXEC SQL DECLARE cursor1 CURSOR FOR STMT1 END-EXEC.
EXEC SQL OPEN cursor1 END-EXEC.

EXEC SQL WHENEVER NOT FOUND GOTO FETCH-END END-EXEC.

PERFORM NO LIMIT
    EXEC SQL FETCH cursor1 INTO :DBANAME,:DATNAME END-EXEC
    DISPLAY "dbaname=" ARR OF DBANAME ", datname=" ARR OF DATNAME
END-PERFORM.

FETCH-END.

EXEC SQL CLOSE cursor1 END-EXEC.
```

```
EXEC SQL COMMIT END-EXEC.
EXEC SQL DISCONNECT ALL END-EXEC.
```

D.6 記述子領域の使用

SQL記述子領域はSELECT、FETCH、DESCRIBE文の結果を処理する、より洗練された手法です。SQL記述子領域は1行のデータをメタデータ項目と一緒に1つのデータ集団項目としてグループ化します。特に動的SQL文を実行する場合は結果列の性質が前もってわかりませんので、メタデータが有用です。PostgreSQLは記述子領域を使用するための方法、名前付きSQL記述子領域を提供します。

D.6.1 名前付きSQL記述子領域

名前付きSQL記述子領域は、記述子全体に関する情報を持つヘッダと、基本的に結果行内の1つの列を記述する、1つ以上の項目記述子領域から構成されます。

SQL記述子領域を使用可能にするためには、それを以下のように割り当てなければなりません。

```
EXEC SQL ALLOCATE DESCRIPTOR identifier END-EXEC.
```

この識別子は記述子領域の"変数名"として使用されます。記述子が不要になったら、以下のように解放してください。

```
EXEC SQL DEALLOCATE DESCRIPTOR identifier END-EXEC.
```

記述子領域を使用するには、INTO句内の格納対象として、ホスト変数を列挙するのではなく、記述子領域を指定してください。

```
EXEC SQL FETCH NEXT FROM mycursor INTO SQL DESCRIPTOR mydesc END-EXEC.
```

結果セットが空の場合であっても、記述子領域には問い合わせのメタデータ、つまりフィールド名、が含まれます。

まだ実行されていないプリペアード・クエリでは、結果セットのメタデータを入手するためにDESCRIBEを使用することができます。

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 SQL-STMT PIC X(30) VARYING.
EXEC SQL END DECLARE SECTION END-EXEC.

MOVE "SELECT * FROM table1" TO ARR OF SQL-STMT.
COMPUTE LEN OF SQL-STMT = FUNCTION STORED-CHAR-LENGTH ( ARR OF SQL-STMT ) .
EXEC SQL PREPARE STMT1 FROM :SQL-STMT END-EXEC.
EXEC SQL DESCRIBE STMT1 INTO SQL DESCRIPTOR MYDESC END-EXEC.
```

PostgreSQL 9.0より前では、SQLキーワードは省略可能でした。このためDESCRIPTORおよびSQL DESCRIPTORは名前付きSQL記述子領域を生成しました。これは強制事項になり、SQLキーワードを省略すると、SQLDA記述子領域を生成する構文とみなされますが、ecobpgではSQLDAに対応していないため、エラーになります。

DESCRIBEおよびFETCH文では、INTOおよびUSINGキーワードを同じように使用することができます。これらは結果セットと記述子領域内のメタデータを生成します。

さて、どうやって記述子領域からデータを取り出すのでしょうか。この記述子領域を名前付きフィールドを持つ集団項目とみなすことができます。ヘッダからフィールド値を取り出し、それをホスト変数に格納するには、以下のコマンドを使用します。

```
EXEC SQL GET DESCRIPTOR name :hostvar = field END-EXEC.
```

今のところ、*COUNT*というヘッダフィールドが1つだけ定義されています。これは、記述子領域に存在する項目数を表すものです（つまり、結果内に含まれる列数です）。このホスト変数はPIC S9(9) COMP-5の整数型でなければなりません。項目記述子領域からフィールドを取り出すには、以下のコマンドを使用します。

```
EXEC SQL GET DESCRIPTOR name VALUE num :hostvar = field END-EXEC.
```

*num*はPIC S9(9) COMP-5形式の整数を持つホスト変数を取ることができます。取り得るフィールドは以下の通りです。

CARDINALITY (整数)

結果セット内の行数です。

DATA

実際のデータ項目です（したがってこのフィールドのデータ型は問い合わせに依存します）。

DATETIME_INTERVAL_CODE (整数)

TYPEが9の場合、DATETIME_INTERVAL_CODEは、DATEでは1、TIMEでは2、TIMESTAMPでは3、TIME WITH TIME ZONEでは4、TIMESTAMP WITH TIME ZONEでは5という値を取ります。

DATETIME_INTERVAL_PRECISION (整数)

未実装です。

INDICATOR (整数)

（NULL値や値の切り詰めを示す）指示子です。

KEY_MEMBER (整数)

実装されていません。

LENGTH (整数)

データの文字列の長さです。

NAME (文字列)

列名です。

NULLABLE (整数)

実装されていません。

OCTET_LENGTH (整数)

データの文字表現のバイト長です。

PRECISION (整数)

（numeric型用の）精度です。

RETURNED_LENGTH (整数)

データの文字数です。

RETURNED_OCTET_LENGTH (整数)

データの文字表現のバイト長です。

SCALE (整数)

（numeric型用の）桁です。

TYPE (整数)

列のデータ型の数値コードです。

EXECUTE、DECLAREおよびOPEN文では、INTOおよびUSINGの効果は異なります。また、問い合わせやカーソル用の入力パラメータを提供するために記述子領域は手作業で構築することができます。USING SQL DESCRIPTOR *name*は入力パラメータをパラメータ付きの問い合わせに渡す方法です。名前付きSQL記述子領域を構築するSQL文は以下の通りです。

```
EXEC SQL SET DESCRIPTOR name VALUE num field = :hostvar END-EXEC.
```

PostgreSQLは、1つのFETCH文内の1レコードを複数取り出し、ホスト変数に格納することをサポートします。この場合ホスト変数は配列であると仮定されます。

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 GROUP-ITEM.
    05 IDNUM PIC S9(9) COMP OCCURS 5.
EXEC SQL END DECLARE SECTION END-EXEC.

EXEC SQL FETCH 5 FROM mycursor INTO SQL DESCRIPTOR mydesc END-EXEC.
```

```
EXEC SQL GET DESCRIPTOR mydesc VALUE 1 :IDNUM = DATA END-EXEC.
```

D.7 エラー処理

本節では、埋め込みSQLプログラムにおいて、例外条件や警告をどのように扱うことができるかについて説明します。このために、共に使用できる2つの機能があります。

- WHENEVERコマンドを使用して、警告条件、エラー条件を扱うようにコールバックを設定することができます。
- エラーまたは警告に関する詳細情報はsqlca変数から入手することができます。

D.7.1 コールバックの設定

エラーや警告を受け取る簡単な手法の1つは、特定の条件が発生する度に特定の動作を実行するように設定することです。一般的には以下のようになります。

```
EXEC SQL WHENEVER condition action END-EXEC.
```

*condition*は以下のいずれかを取ることができます。

SQLERROR

SQL文の実行中にエラーが発生する度に、指定した動作が呼び出されます。

SQLWARNING

SQL文の実行中に警告が発生する度に、指定した動作が呼び出されます。

NOT FOUND

SQL文が0行を受け取る、もしくは0行に影響する時、指定した動作が呼び出されます。（この条件はエラーではありませんが、これを特別に扱いたい場合があるかもしれません。）

*action*は以下のいずれかを取ることができます。

CONTINUE

これは、実際のところ、その条件が無視されることを意味します。これがデフォルトです。

GOTO *label*
GO TO *label*

指定したラベルに移動します（COBOLのGOTO文を使用します）。

SQLPRINT

標準エラーにメッセージを出力します。これは、単純なプログラムやプロトタイプ作成時に役に立ちます。メッセージの詳細は設定できません。

STOP

プログラムを終了させるSTOPを呼び出します。

CALL *name* using *args*
DO *name* using *args*

指定した引数で、指定した関数を呼び出します。そのため、引数以外の構文(コンパイラ依存のものを含む)を含めてもかまいません。ただし、以下の制約があります。

- RETURNING句、ON EXCEPTION句、OVER FLOW句は使うことができません。
- コールされた副プログラムでは、WHENEVER文を使うすべてのアクションにおいて、CONTINUEを指定しなければなりません。

標準SQLではCONTINUEとGOTO（とGO TO）のみを提供しています。

簡単なプログラムで使用してみたくなるような例を以下に示します。警告が発生した場合に簡単なメッセージを表示し、エラーが発生した場合にプログラムを中断します。

```
EXEC SQL WHENEVER SQLWARNING SQLPRINT END-EXEC.
EXEC SQL WHENEVER SQLERROR STOP END-EXEC.
```

EXEC SQL WHENEVER文はCOBOLの構文ではなく、SQLプリプロセッサのディレクティブです。設定したエラーもしくは警告動作は、最初のEXEC SQL WHENEVERと条件を発生させたSQL文の間で、同一条件に異なる動作が設定されない限り、ハンドラを設定した箇所より後にある、すべての埋め込みSQL文に適用されます。COBOLプログラムの制御フローは関係しません。したがって、以下の2つのCOBOLプログラムの抜粋はどちらも望み通りの動作を行いません。

```
*
*    WRONG
*
    ...
    IF VERBOSE = 1 THEN
        EXEC SQL WHENEVER SQLWARNING SQLPRINT END-EXEC
    END-IF.
    ...
    EXEC SQL SELECT ... END-EXEC.
...
*
*    WRONG
*
    ...
    CALL SET-ERROR-HANDLER.
*         (and execute "EXEC SQL WHENEVER SQLERROR STOP" in SET-ERROR-HANDLER)
    ...
    EXEC SQL SELECT ... END-EXEC.
    ...
```

D.7.2 sqlca

より強力にエラーを扱うために、埋め込みSQLインタフェースは以下の集合変数であるSQLCA（SQL通信領域）という名前のグローバル変数を提供します。

```
01  sqlca_t.
    10 sqlcaid PIC X(8).
    10 sqlabc PIC S9(9) COMP-5.
    10 sqlcode PIC S9(9) COMP-5.
    10 sqlerrm.
        20 sqlerrml PIC S9(9) COMP-5.
        20 sqlerrmc PIC X(150).
    10 sqlerrp PIC X(8).
    10 sqlerrd PIC S9(9) COMP-5 OCCURS 6.
    10 sqlwarn PIC X(8).
    10 sqlstate PIC X(5).
```

SQLCAは警告とエラーの両方を対象としています。1つのSQL文の実行時に複数の警告やエラーが発生した場合、SQLCAは最後のものに関した情報のみを含みます。

直前のSQL文でエラーがなければ、SQLCODEは0に、SQLSTATEは"00000"になります。警告やエラーが発生した場合は、SQLCODEは負の値に、SQLSTATEは"00000"以外になります。正のSQLCODEは、直前の問い合わせが0行を返したなどの無害な条件を示します。SQLCODEとSQLSTATEは2つの異なるエラーコードスキーマです。後で詳細に説明します。

直前のSQL文が成功すると、SQLERRD(2)は処理された行のOIDが、もしあれば、格納されます。また、もしそのコマンドで適切ならば、SQLERRD(3)は処理された、もしくは返された行数が格納されます。

エラーもしくは警告の場合、SQLERRMCには、そのエラーを説明する文字列が格納されます。SQLERRMLフィールドにはSQLERRMCに格納されたエラーメッセージ長が格納されます（FUNCTION STORED-CHAR-LENGTHの結果です）。一部のメッセージは固定長のSQLERRMC配列には長過ぎることに注意してください。この場合は切り詰められます。

警告の場合、SQLWARNの3文字目はWに設定されます（他のすべての場合では、これはW以外の何かに設定されます）。SQLWARNの2文字目がWに設定された場合、ホスト変数に代入する際に値が切り詰められています。他の要素が警告を示すように設定された場合、SQLWARNの最初の文字はWに設定されます。

今のところ、SQLCAID、SQLCABC、SQLERRPならびにSQLERRDとSQLWARNの上記以外の要素は有用な情報を持ちません。

SQLCAは標準SQLでは定義されていません。しかし、複数の他のSQLデータベースシステムで実装されています。その定義は基本部分は似ていますが、移植性を持つアプリケーションを作成する場合は実装の違いを注意して調査しなければなりません。

ここでWHENEVERとSQLCAを組み合わせて使用して、エラーが発生した時にSQLCAの内容を表示する、1つの例を示します。これはおそらく、より"ユーザ向け"のエラー処理を組み込む前の、アプリケーションのデバッグまたはプロトタイプで有用です。

```
EXEC SQL WHENEVER SQLERROR GOTO PRINT_SQLCA END-EXEC.

PRINT_SQLCA.
    DISPLAY "==== sqlca ====".
    DISPLAY "SQLCODE: " SQLCODE.
    DISPLAY "SQLERRML: " SQLERRML.
    DISPLAY "SQLERRMC: " SQLERRMC.
    DISPLAY "SQLERRD: " SQLERRD(1) " " SQLERRD(2) " " SQLERRD(3)" " SQLERRD(4) " "
SQLERRD(5) " " SQLERRD(6).
    DISPLAY "SQLSTATE: " SQLSTATE.
    DISPLAY "===============".
```

結果は以下のようになります（ここでのエラーはテーブル名の誤記述によるものです）。

```
==== sqlca ====
sqlcode: -000000400
SQLERRML: +000000064
SQLERRMC: relation "pg_databasep" does not exist (10292) on line 93
sqlerrd: +000000000 +000000000 +000000000 +000000000 +000000000 +000000000
sqlstate: 42P01
===============
```

D.7.3 SQLSTATE対SQLCODE

SQLSTATEとSQLCODEはエラーコードを提供する異なる2つの機構です。共に標準SQLから派生されたものですが、SQLCODEはSQL-92版では廃れたものとされ、以降の版から削除されました。したがって、新規アプリケーションではSQLSTATEを使用することを強く勧めます。

SQLSTATEは5要素の文字配列です。この5文字は、各種のエラー条件、警告条件のコードを表現する数字、大文字から構成されます。SQLSTATEは階層を持った機構です。最初の2文字は条件を汎化したクラスを示し、残り3文字は汎化クラスの副クラスを示します。成功状態は00000というコードで示されます。SQLSTATEコードのほとんどは標準SQLで定義されています。PostgreSQLサーバは本質的にSQLSTATEエラーコードをサポートしています。したがって、すべてのアプリケーションでこのエラーコードを使用することで、高度な一貫性を達成することができます。

参照

詳細については“PostgreSQL Documentation”の“Appendixes”の“PostgreSQL Error Codes”を参照してください。

廃止されたエラーコードの機構であるSQLCODEは単なる整数です。0という値は成功を意味し、正の値は追加情報を持った成功を、負の値はエラーを示します。標準SQLでは、直前のコマンドが0行を返す、もしくは0行に影響したことを示す+100という正の値のみを定義しています。負の値は規定されていません。したがって、この機構では低い移植性しか達成できず、また、コード体系も階層を持っていません。歴史的に、PostgreSQLの埋め込みSQLプロセッサには、いくつかの特殊なSQLCODEの値が専用に割り当てられていました。以下に、その数値とそのシンボル名の一覧を示します。これらは他のSQL実装への移植性がないことを忘れないでください。アプリケーションのSQLSTATE機構への移行を簡易化するために、対応するSQLSTATEも示しています。しかし、2つのしくみの間の関係は1対1ではなく1対多です（実際は多対多です）。ですので、場合ごとにグローバルな各SQLSTATEを参照しなければなりません。

参照

“PostgreSQL Documentation”の“Appendixes”の“PostgreSQL Error Codes”を参照してください。

以下は割り当て済みのSQLCODEです。

0

エラーがないことを示す。(SQLSTATE 00000)

100

これは、最後に実行したコマンドが取り出した、または、処理した行がゼロ行であったこと、あるいは、カーソルの最後であることを示す、害のない条件です。(SQLSTATE 02000)

以下のように、カーソルをループ内で処理する時、ループを中断する時を検知する方法として、このコードを使用することができます。

```
PERFORM NO LIMIT
EXEC SQL FETCH ... END-EXEC
IF SQLCODE = 100 THEN
GO TO FETCH-END
END-IF
END-PERFORM.
```

しかし、WHENEVER NOT FOUND GOTO…はこれを内部で効率的に行います。このため、通常、外部で明示的に記述する利点はありません。

-12

仮想メモリ不足を示します。この数値は-ENOMEMとして定義します。(SQLSTATE YE001)

-200

ライブラリが把握していない何かをプリプロセッサが生成したことを示します。おそらく、互換性がないプリプロセッサとライブラリのバージョンを使用しています。(SQLSTATE YE002)

-201

コマンドの想定より多くのホスト変数が指定されたことを意味します。(SQLSTATE 07001もしくは07002)

-202

コマンドの想定よりも少ないホスト変数が指定されたことを意味します。(SQLSTATE 07001もしくは07002)

-203

問い合わせが複数行を返したけれども、SQL文では1つの結果の格納の準備だけしかしていなかったことを意味します(例えば、指定された変数が配列ではなかった)。(SQLSTATE 21000)

-204

ホスト変数の型が符号あり整数ですが、データベース内のデータ型が異なり、その値を符号あり整数として解釈させることができませんでした。ライブラリはこの変換にstrtol()を使用します。(SQLSTATE 42804)

-205

ホスト変数の型が符号なし整数ですが、データベース内のデータ型が異なり、その値を符号なし整数として解釈させることができませんでした。ライブラリはこの変換にstrtoul()を使用します。(SQLSTATE 42804)

-206

ホスト変数の型が浮動小数点型ですが、データベース内のデータ型が異なり、その値を 浮動小数点型として解釈させることができませんでした。ライブラリはこの変換にstrtod()を使用します。(SQLSTATE 42804)

-207

ホスト変数の型がDECIMALかDISPLAYですが、データベース内のデータ型が異なり、その値をDECIMALとして解釈させることができませんでした。DISPLAYの場合、データベース内の値が大きすぎて、値を変換できない場合に、このエラーが起こります。(SQLSTATE 42804)

-208

ホスト変数の型がINTERVALであり、データベース内のデータが他の型であり、INTERVAL値として解釈することができない値を含みます。(SQLSTATE 42804)

-209

ホスト変数の型がDATEであり、データベース内のデータが他の型であり、DATE値として解釈することができない値を含みます。(SQLSTATE 42804)

-210

ホスト変数の型がTIMESTAMPであり、データベース内のデータが他の型であり、TIMESTAMP値として解釈することができない値を含みます。(SQLSTATE 42804)

-211

これは、ホスト変数の型がBOOLですが、データベース内のデータが't'でも'f'でもなかったことを意味します。（SQLSTATE 42804）

-212

PostgreSQLサーバに送信されたSQL文が空でした（通常埋め込みSQLプログラムでは発生しません。ですので、これは内部エラーを示しているかもしれません）。（SQLSTATE YE002）

-213

NULL値が返されましたが、NULL用の指示子変数が与えられていませんでした。（SQLSTATE 22002）

-214

配列が必要な箇所に普通の変数が使用されていました。（SQLSTATE 42804）

-215

配列値が必要な箇所にデータベースが普通の変数を返しました。（SQLSTATE 42804）

-220

存在しない接続にプログラムがアクセスしようとしました。（SQLSTATE 08003）

-221

存在するが開いていない接続にプログラムがアクセスしようとしました（これは内部エラーです）。（SQLSTATE YE002）

-230

使用しようとしたSQL文がプリペアされていませんでした。（SQLSTATE 26000）

-240

指定した記述子が見つかりませんでした。使用しようとしたSQL文はプリペアされていませんでした。（SQLSTATE 33000）

-241

記述子のインデックスが範囲外でした。（SQLSTATE 07009）

-242

無効な記述子項目が要求されました。（これは内部エラーです。）（SQLSTATE YE002）

-243

動的なSQL文の実行時にデータベースが数値を返しましたが、ホスト変数が数値でありませんでした。（SQLSTATE 07006）

-244

動的なSQL文の実行時にデータベースが数値以外を返しましたが、ホスト変数が数値でした。（SQLSTATE 07006）

-400

PostgreSQLサーバで何らかのエラーが発生しました。このメッセージはPostgreSQLサーバからのエラーメッセージを含みます。

-401

PostgreSQLサーバがトランザクションのコミットやロールバックを始めることができないことを通知しました。(SQLSTATE 08007)

-402

データベースへの接続試行に失敗しました。(SQLSTATE 08001)

-403

重複キーエラー。一意性制約違反。(SQLSTATE 23505)

-404

副問い合わせの結果が単一行ではありません。(SQLSTATE 21000)

-602

無効なカーソル名が指定されました。(SQLSTATE 34000)

-603

トランザクションが進行中です。(SQLSTATE 25001)

-604

活動中(進行中)のトランザクションがありません。(SQLSTATE 25P01)

-605

既存のカーソル名が指定されました。(SQLSTATE 42P03)

D.8 プリプロセッサ指示子

ecobpgプリプロセッサがファイルを解析および処理する方法を変更することができる、プリプロセッサ指示子が複数あります。

D.8.1 ファイルのインクルード

埋め込みSQLプログラムに外部ファイルをインクルードするには、以下を使用します。

```
EXEC SQL INCLUDE filename END-EXEC.
EXEC SQL INCLUDE <filename> END-EXEC.
EXEC SQL INCLUDE "filename" END-EXEC.
```

埋め込みSQLプリプロセッサは、*filename*.pcoという名前のファイルを探し、その前処理を行い、最終的にC出力の中に含めます。このようにして、ヘッダファイル内の埋め込みSQL文が正しく扱われます。

ecobpgプリプロセッサはデフォルトではカレントディレクトリからファイルを検索します。これは、ecobpgコマンドのコマンドラインオプションで変更可能です

カレントディレクトリの中で、プリプロセッサはまず指定されたファイル名を探します。

ファイルが見つからず、かつそのファイル名が接尾語に.pcoをもっていない場合、ファイル名に.pcoを付けて再検索します。

EXEC SQL INCLUDEとCOPY文の違いは、読み込まれたファイルにプレコンパイラによる埋め込みSQLの評価がなされるかどうかです。埋め込みSQLを含むファイルを読み込む場合は、EXEC SQL INCLUDEを使用してください。

注意

通常のSQLの大文字小文字の区別規則に従うEXEC SQL INCLUDEコマンドの一部であったとしても、インクルードファイルの名前は大文字小文字が区別されます。

D.8.2 defineおよびundef指示子

Cで既知の#define指示子と同様、埋め込みSQLでも似たような概念を持ちます。

```
EXEC SQL DEFINE name END-EXEC.
EXEC SQL DEFINE name value END-EXEC.
```

このため、以下のように名前を定義することができます。

```
EXEC SQL DEFINE HAVE_FEATURE END-EXEC.
```

また、定数を定義することもできます。

```
EXEC SQL DEFINE MYNUMBER 12 END-EXEC.
EXEC SQL DEFINE MYSTRING 'abc' END-EXEC.
```

事前の定義を削除するにはundefを使用します。

```
EXEC SQL UNDEF MYNUMBER END-EXEC.
```

SQL文内の定数は、EXEC SQL DEFINEによって置き換えられるだけであることに注意してください。置換によって1行の文字数が変わる場合がありますが、ecobpgは置換後の1行の文字数の妥当性をチェックしません。1行の文字数の制限に収まるようにDEFINEを使用してください。

D.8.3 ifdef, ifndef, else, elif, endif指示子

以下の指示子を使用して、コンパイルするコード部分を選択することができます。

```
EXEC SQL ifdef name END-EXEC.
```

*name*を検査し、その*name*がEXEC SQL define *name*で作成されていた場合に後続の行を処理します。

```
EXEC SQL ifndef name END-EXEC.
```

*name*を検査し、その*name*がEXEC SQL define *name*で作成されて*いない*場合に後続の行を処理します。

```
EXEC SQL else END-EXEC.
```

EXEC SQL ifdef *name*またはEXEC SQL ifndef *name*で導入されたセクションの代替セクションを開始します。

```
EXEC SQL elif name END-EXEC.
```

*name*を検査し、その*name*がEXEC SQL define *name*で作成されている場合に代替セクションを開始します。

```
EXEC SQL endif END-EXEC.
```

代替セクションを終了します。

以下に例を示します。

```
EXEC SQL ifndef TZVAR END-EXEC.
EXEC SQL SET TIMEZONE TO 'GMT' END-EXEC.
EXEC SQL elif TZNAME END-EXEC.
EXEC SQL SET TIMEZONE TO TZNAME END-EXEC.
EXEC SQL else END-EXEC.
EXEC SQL SET TIMEZONE TO TZVAR END-EXEC.
EXEC SQL endif END-EXEC.
```

D.9 埋め込みSQLプログラムの処理

ここまでで、埋め込みSQL COBOLプログラムの作成方法は理解できたと思います。ここからはそのコンパイル方法についてお話しします。コンパイルの前に、そのファイルを埋め込みSQLCOBOLプレコンパイラに通します。これは、使用するSQL文を特別な関数呼び出しに変換します。コンパイル後、必要な関数を持つ特別なライブラリとリンクしなければなりません。これらの関数は引数から情報を取り出し、libpqを使用してそのSQLを実行し、出力用に指定された引数にその結果を格納します。

プリプロセッサプログラムはecobpgという名前です。通常、埋め込みSQLプログラムの拡張子は.pcoとします。prog1.pcoという名前のプログラムファイルがある場合、単純に以下を呼び出すことで前処理を行うことができます。

```
ecobpg prog1.pco
```

これはprog1.cobという名前のファイルを作成します。入力ファイルがこの提案パターンに従った名前でない場合、-o オプションを使用して明示的に出力ファイルを指定することができます。

前処理後のファイルは普通にコンパイルできます。使用するコンパイラのマニュアルに従ってください。

生成されたCOBOLソースファイルはecobpgのインストレーションに付随する登録文ファイルをインクルードします。ですので、デフォルトで検索されない場所にecobpgをインストールした場合は、コンパイル用のコマンドラインにインクルードファイルの検索パスを指定するオプションを追加する必要があります。

埋め込みSQLプログラムをリンクするためには、libecpgライブラリを含めなければなりません。

繰り返しになりますが、コマンドラインにライブラリ検索パスを指定するオプションを追加する必要があります。

大規模プロジェクトの構築処理をmakeを使用して管理している場合、以下の暗黙規則をMakefileに含めておくと便利です。

```
ECOBPG = ecobpg

%.cob: %.pco
$(ECOBPG) $<
```

ecobpgコマンドの完全な構文は“D.12.1 ecobpg”に説明があります。

現在、ecobpgはマルチスレッドには対応していません。

D.10 ラージオブジェクト

ラージオブジェクトはECOBPGで直接サポートされていません。

ラージオブジェクトにアクセスする必要がある場合は、libpqのラージオブジェクトインターフェースを使用してください。

D.11 埋め込みSQLコマンド

本節では、埋め込みSQL固有のSQLコマンドをすべて説明します。

コマンド名	説明
ALLOCATE DESCRIPTOR	SQL記述子領域を割り当てます。
CONNECT	データベース接続を確立します。
DEALLOCATE DESCRIPTOR	SQL記述子領域の割り当てを解除します。
DECLARE	カーソルを定義します。
DESCRIBE	プリペアード・ステートメントまたは結果セットに関する情報を入手します。
DISCONNECT	データベース接続を終了します。
EXECUTE IMMEDIATE	SQL文を動的にプリペアし、実行します。
GET DESCRIPTOR	SQL記述子領域から情報を入手します。
OPEN	動的カーソルを開きます。
PREPARE	実行のためにSQL文をプリペアします。
SET AUTOCOMMIT	現在のセッションの自動コミット動作を設定します。
SET CONNECTION	データベース接続を選択します。
SET DESCRIPTOR	SQL記述子領域に情報を設定します。
TYPE	新しいデータ型を定義します。

コマンド名	説明
WHENEVER	SQL文により特定の分類の条件が発生する時に行う動作を指定します。

参照

言及がない限り、埋め込みSQLでも使用することができる、“PostgreSQL Documentation”の“Reference”の“SQL Commands”に列挙されたSQLコマンドを参照してください。

D.11.1 ALLOCATE DESCRIPTOR

名前

ALLOCATE DESCRIPTOR -- SQL記述子領域を割り当てます。

記述形式

```
ALLOCATE DESCRIPTOR name
```

説明

ALLOCATE DESCRIPTORは、PostgreSQLサーバとホストプログラムとの間のデータ交換のために使用することができる、新しい名前付きSQL記述子領域を割り当てます。

記述子領域は、使用した後でDEALLOCATE DESCRIPTORコマンドを使用して解放しなければなりません。

パラメータ

name

SQL記述子の名前です。これはSQL識別子またはホスト変数になることができます。

使用例

```
EXEC SQL ALLOCATE DESCRIPTOR mydesc END-EXEC.
```

互換性

ALLOCATE DESCRIPTORは標準SQLで規定されています。

関連項目

DEALLOCATE DESCRIPTOR, GET DESCRIPTOR, SET DESCRIPTOR

D.11.2 CONNECT

名前

CONNECT -- データベース接続を確立します。

記述形式

```
CONNECT TO connection_target [ AS connection_name ] [ USER connection_user_name ]

CONNECT TO DEFAULT

CONNECT connection_user_name

DATABASE connection_target
```

説明

CONNECTコマンドはクライアントとPostgreSQLサーバとの間の接続を確立します。

パラメータ

connection_target

connection_target 以下の複数の形式の1つを使用して、接続する対象サーバを指定します。

[*database_name*] [*@host*] [*:port*]

TCP/IPを介した接続。

unix:postgresql://*host* [*:port*] / [*database_name*] [*?connection_option*]

Unixドメインソケットを介した接続。

tcp:postgresql://*host* [*:port*] / [*database_name*] [*?connection_option*]

TCP/IPを介した接続。

SQL文字列定数

上記形式のいずれかで記述された値を持ちます。

ホスト変数

上記形式のいずれかで記述された値を持つVARCHAR型のホスト変数。

connection_name

他のコマンドで参照することができる、このデータベース接続の識別子です。省略可能です。これはSQL識別子またはホスト変数とすることができます。

connection_user_name

データベース接続用のユーザ名です。

このパラメータは、*user_name*/*password*、*user_name* IDENTIFIED BY *password*、*user_name* USING *password*のいずれかの形式を使用して、ユーザ名とパスワードを指定することができます。

ユーザ名とパスワードは、SQL識別子、文字列定数、ホスト変数とすることができます。

DEFAULT

libpqで定義された、デフォルトの接続パラメータすべてを使用します。

使用例

以下に接続パラメータを指定する複数の種類を示します。

```
EXEC SQL CONNECT TO "connectdb" AS main END-EXEC.
EXEC SQL CONNECT TO "connectdb" AS second END-EXEC.
EXEC SQL CONNECT TO "unix:postgresql://localhost/connectdb" AS main USER connectuser END-EXEC.
EXEC SQL CONNECT TO 'connectdb' AS main END-EXEC.
EXEC SQL CONNECT TO 'unix:postgresql://localhost/connectdb' AS main USER :user END-EXEC.
EXEC SQL CONNECT TO :db AS :id END-EXEC.
EXEC SQL CONNECT TO :db USER connectuser USING :pw END-EXEC.
EXEC SQL CONNECT TO @localhost AS main USER connectdb END-EXEC.
EXEC SQL CONNECT TO REGRESSDB1 as main END-EXEC.
EXEC SQL CONNECT TO AS main USER connectdb END-EXEC.
EXEC SQL CONNECT TO connectdb AS :id END-EXEC.
EXEC SQL CONNECT TO connectdb AS main USER connectuser/connectdb END-EXEC.
EXEC SQL CONNECT TO connectdb AS main END-EXEC.
EXEC SQL CONNECT TO connectdb@localhost AS main END-EXEC.
EXEC SQL CONNECT TO tcp:postgresql://localhost/ USER connectdb END-EXEC.
EXEC SQL CONNECT TO tcp:postgresql://localhost/connectdb USER connectuser IDENTIFIED BY connectpw END-EXEC.
EXEC SQL CONNECT TO tcp:postgresql://localhost:20/connectdb USER connectuser IDENTIFIED BY connectpw END-EXEC.
EXEC SQL CONNECT TO unix:postgresql://localhost/ AS main USER connectdb END-EXEC.
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb AS main USER connectuser END-EXEC.
```

```
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb USER connectuser IDENTIFIED BY "connectpw" END-EXEC.
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb USER connectuser USING "connectpw" END-EXEC.
EXEC SQL CONNECT TO unix:postgresql://localhost/connectdb?connect_timeout=14 USER connectuser END-EXEC.
```

以下にホスト変数を使用して接続パラメータを指定する方法を示すプログラム例を示します。

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
*   database name
    01 DBNAME PIC X(6).
*   connection user name
    01 USER PIC X(8).
*   connection string
    01 CONNECTION PIC X(38).

    01 VER PIC X(256).
EXEC SQL END DECLARE SECTION END-EXEC.

    MOVE "testdb" TO DBNAME.
    MOVE "testuser" TO USER.
    MOVE "tcp:postgresql://localhost:5432/testdb" TO CONNECTION.

    EXEC SQL CONNECT TO :DBNAME USER :USER END-EXEC.
    EXEC SQL SELECT version() INTO :VER END-EXEC.
    EXEC SQL DISCONNECT END-EXEC.


    DISPLAY "version: " VER.

    EXEC SQL CONNECT TO :CONNECTION USER :USER END-EXEC.
    EXEC SQL SELECT version() INTO :VER END-EXEC.
    EXEC SQL DISCONNECT END-EXEC.

    DISPLAY "version: " VER.
```

互換性

CONNECTは標準SQLで規定されていますが、接続パラメータの形式は、特有の実装です。

関連項目

DISCONNECT, SET CONNECTION

D.11.3 DEALLOCATE DESCRIPTOR

名前

DEALLOCATE DESCRIPTOR--SQL記述子領域の割り当てを解除します。

記述形式

```
DEALLOCATE DESCRIPTOR name
```

説明

DEALLOCATE DESCRIPTORは名前付きSQL記述子領域の割り当てを解除します。

パラメータ

name

割り当てを解除する記述子の名前です。これはSQL識別子またはホスト変数にすることができます。

使用例

```
EXEC SQL DEALLOCATE DESCRIPTOR mydesc END-EXEC.
```

互換性

DEALLOCATE DESCRIPTORは標準SQLで規定されています。

関連項目

ALLOCATE DESCRIPTOR, GET DESCRIPTOR, SET DESCRIPTOR

D.11.4 DECLARE

名前

DECLARE -- カーソルを定義します。

記述形式

```
DECLARE cursor_name [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ] CURSOR [ { WITH | WITHOUT } HOLD ] FOR
prepared_name

DECLARE cursor_name [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ] CURSOR [ { WITH | WITHOUT } HOLD ] FOR query
```

説明

DECLAREは、プリペアード・ステートメントの結果セット全体を繰り返し処理するカーソルを宣言します。このコマンドは直接的なSQLコマンドのDECLAREとは多少異なる意味を持ちます。こちらは問い合わせを実行し、取り出し用の結果セットの準備を行いますが、埋め込みSQLコマンドでは、問い合わせの結果セット全体を繰り返す“ループ変数”の名前を宣言するだけです。実際の実行はOPENコマンドでカーソルが開いた時に起こります。

パラメータ

cursor_name

カーソル名です。これはSQL識別子またはホスト変数とすることができます。

prepared_name

プリペアード・クエリの名前です。SQL識別子またはホスト変数のいずれかです。

query

このカーソルで返される行を供給するSELECTまたはVALUESコマンドです。

カーソルオプションの意味についてはDECLAREを参照してください。

参照

SELECT、VALUE、DECLAREコマンドの詳細については、“PostgreSQL Documentation”の“Reference”の“SQL Commands”を参照してください。

使用例

以下に問い合わせ用のカーソルを宣言する例を示します。

```
EXEC SQL DECLARE C CURSOR FOR SELECT * FROM My_Table END-EXEC.
EXEC SQL DECLARE C CURSOR FOR SELECT Item1 FROM T END-EXEC.
EXEC SQL DECLARE cur1 CURSOR FOR SELECT version() END-EXEC.
```

プリペアード・ステートメント用のカーソルを宣言する例を示します。

```
EXEC SQL PREPARE stmt1 AS SELECT version() END-EXEC.
EXEC SQL DECLARE cur1 CURSOR FOR stmt1 END-EXEC.
```

互換性

DECLAREは標準SQLで規定されています。

関連項目

OPEN, CLOSE, DECLARE

参照

CLOSE、DECLAREコマンドの詳細については、“PostgreSQL Documentation”の“Reference”の“SQL Commands”を参照してください。

D.11.5 DESCRIBE

名前

DESCRIBE -- プリペアード・ステートメントまたは結果セットに関する情報を入手します。

記述形式

DESCRIBE [OUTPUT] *prepared_name* USING SQL DESCRIPTOR *descriptor_name*

DESCRIBE [OUTPUT] *prepared_name* INTO SQL DESCRIPTOR *descriptor_name*

説明

DESCRIBEは、実際に行を取り込むことなく、プリペアード・ステートメントに含まれる結果列に関するメタデータ情報を取り出します。

パラメータ

prepared_name

プリペアード・ステートメントの名前です。これはSQL識別子またはホスト変数とすることができます。

descriptor_name

記述子の名前です。これはSQL識別子またはホスト変数とすることができます。

使用例

```
EXEC SQL ALLOCATE DESCRIPTOR mydesc END-EXEC.
EXEC SQL PREPARE stmt1 FROM :sql_stmt END-EXEC.
EXEC SQL DESCRIBE stmt1 INTO SQL DESCRIPTOR mydesc END-EXEC.
EXEC SQL GET DESCRIPTOR mydesc VALUE 1 :charvar = NAME END-EXEC.
EXEC SQL DEALLOCATE DESCRIPTOR mydesc END-EXEC.
```

互換性

DESCRIBEは標準SQLで規定されています。

関連項目

ALLOCATE DESCRIPTOR, GET DESCRIPTOR

D.11.6 DISCONNECT

名前

DISCONNECT -- データベース接続を終了します。

記述形式

```
DISCONNECT connection_name

DISCONNECT [ CURRENT ]

DISCONNECT DEFAULT

DISCONNECT ALL
```

説明

DISCONNECTはデータベースとの接続(またはすべての接続)を閉ざします。

パラメータ

connection_name

CONNECTコマンドで確立したデータベース接続の名前です。

CURRENT

直前に開いた接続またはSET CONNECTIONコマンドで設定された接続のいずれかである、"現在の"接続を閉ざします。これはDISCONNECTに引数が与えられなかった場合のデフォルトです。

DEFAULT

デフォルトの接続を閉ざします。

ALL

開いているすべての接続を閉じます。

使用例

```
EXEC SQL CONNECT TO testdb AS DEFAULT USER testuser END-EXEC.
EXEC SQL CONNECT TO testdb AS con1 USER testuser END-EXEC.
EXEC SQL CONNECT TO testdb AS con2 USER testuser END-EXEC.
EXEC SQL CONNECT TO testdb AS con3 USER testuser END-EXEC.

*   close con3
    EXEC SQL DISCONNECT CURRENT END-EXEC.
*   close DEFAULT
    EXEC SQL DISCONNECT DEFAULT END-EXEC.
*   close con2 and con1
    EXEC SQL DISCONNECT ALL END-EXEC.
```

互換性

DISCONNECTは標準SQLで規定されています。

関連項目

CONNECT, SET CONNECTION

D.11.7 EXECUTE IMMEDIATE

名前

EXECUTE IMMEDIATE -- SQL文を動的にプリペアし、実行します。

記述形式

EXECUTE IMMEDIATE *string*

説明

EXECUTE IMMEDIATEは動的に指定されたSQL文を、結果行を受け取ることなく、即座にプリペアし実行します。

パラメータ

string

実行するSQL文を含む文字列リテラルまたはホスト変数です。

使用例

以下に、EXECUTE IMMEDIATEとcommandホスト変数を使用してINSERTを実行する例を示します。

```
MOVE "INSERT INTO test (name, amount, letter) VALUES ('db: ''r1''', 1, 'f')" TO ARR OF cmd.
COMPUTE LEN OF cmd = FUNCTION STORED-CHAR-LENGTH(ARR OF cmd).
EXEC SQL EXECUTE IMMEDIATE :cmd END-EXEC.
```

互換性

EXECUTE IMMEDIATEは標準SQLで規定されています。

D.11.8 GET DESCRIPTOR

名前

GET DESCRIPTOR -- SQL記述子領域から情報を入手します。

記述形式

GET DESCRIPTOR *descriptor_name :hostvariable* = *descriptor_header_item* [, ...]

GET DESCRIPTOR *descriptor_name* VALUE *column_number :hostvariable* = *descriptor_item* [, ...]

説明

GET DESCRIPTORはSQL記述子領域から問い合わせ結果セットに関する情報を取り出し、それをホスト変数に格納します。記述子領域は通常、このコマンドを使用してホスト言語変数に情報を転送する前に、FETCHまたはSELECTを用いて値が投入されます。

このコマンドには2つの構文があります。1番目の構文では、そのまま結果セットに適用されている記述子の"ヘッダ"項目を取り出します。行数が1つの例です。列番号を追加のパラメータとして必要とする2番目の構文では特定の列に関する情報を取り出します。例えば、列名と列の実際の値です。

パラメータ

descriptor_name

記述子の名前です。

descriptor_header_item

どのヘッダ情報を取り出すかを識別するトークンです。結果セット内の列数を入手するCOUNTのみが現在サポートされています。

column_number

情報を取り出す列の番号です。1から数えます。

descriptor_item

どの列に関する情報を取り出すかを識別するトークンです。サポートされる項目のリストについては“D.6.1 名前付きSQL記述子領域”を参照してください。

hostvariable

記述子領域から取り出したデータを受け取るホスト変数です。

使用例

この例は結果セット内の列数を取り出します。

```
EXEC SQL GET DESCRIPTOR d :d_count = COUNT END-EXEC.
```

この例は最初の列のデータ長を取り出します。

```
EXEC SQL GET DESCRIPTOR d VALUE 1 :d_returned_octet_length = RETURNED_OCTET_LENGTH END-EXEC.
```

この例は、2番目の列のデータ本体を文字列として取り出します。

```
EXEC SQL GET DESCRIPTOR d VALUE 2 :d_data = DATA END-EXEC.
```

以下は、SELECT current_database();を実行し、列数、列のデータ長、列のデータを表示する手続き全体を示す例です。

```
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
    01 D-COUNT PIC S9(9) COMP-5.
    01 D-DATA PIC X(1024).
    01 D-RETURNED-OCTET-LENGTH PIC S9(9) COMP.
EXEC SQL END DECLARE SECTION END-EXEC.

    EXEC SQL CONNECT TO testdb AS con1 USER testuser END-EXEC.
    EXEC SQL ALLOCATE DESCRIPTOR d END-EXEC.

*    Declare, open a cursor, and assign a descriptor to the cursor
    EXEC SQL DECLARE cur CURSOR FOR SELECT current_database() END-EXEC.
    EXEC SQL OPEN cur END-EXEC.
    EXEC SQL FETCH NEXT FROM cur INTO SQL DESCRIPTOR d END-EXEC.

*    Get a number of total columns
    EXEC SQL GET DESCRIPTOR d :D-COUNT = COUNT END-EXEC.
    DISPLAY "d_count = " D-COUNT.

*    Get length of a returned column
    EXEC SQL GET DESCRIPTOR d VALUE 1 :D-RETURNED-OCTET-LENGTH = RETURNED_OCTET_LENGTH END-EXEC.
    DISPLAY "d_returned_octet_length = " D-RETURNED-OCTET-LENGTH.

*    Fetch the returned column as a string
    EXEC SQL GET DESCRIPTOR d VALUE 1 :D-DATA = DATA END-EXEC.
    DISPLAY "d_data = " D-DATA.

*    Closing
    EXEC SQL CLOSE cur END-EXEC.
    EXEC SQL COMMIT END-EXEC.

    EXEC SQL DEALLOCATE DESCRIPTOR d END-EXEC.
    EXEC SQL DISCONNECT ALL END-EXEC.
```

この例を実行すると、結果は以下のようになります。

```
d_count = +000000001
d_returned_octet_length = +000000006
d_data = testdb
```

互換性

GET DESCRIPTORは標準SQLで規定されています。

関連項目

ALLOCATE DESCRIPTOR, SET DESCRIPTOR

D.11.9 OPEN

名前

OPEN -- 動的カーソルを開きます。

記述形式

OPEN *cursor_name*

OPEN *cursor_name* USING *value* [, ...]

OPEN *cursor_name* USING SQL DESCRIPTOR *descriptor_name*

説明

OPENはカーソルを開き、省略することができますが、実際の値をカーソル定義内のプレースホルダにバインドします。カーソルは事前にDECLAREコマンドを用いて宣言されていなければなりません。OPENの実行により問い合わせがサーバ上で実行を開始されます。

パラメータ

cursor_name

開くカーソルの名前です。これはSQL識別子またはホスト変数とすることができます。

value

カーソル内のプレースホルダにバインドされる値です。これは、SQL定数、ホスト変数、指示子を持つホスト変数とすることができます。

descriptor_name

カーソル内のプレースホルダにバインドされる値を含む記述子の名前です。これはSQL識別子またはホスト変数とすることができます。

使用例

```
EXEC SQL OPEN a END-EXEC.
EXEC SQL OPEN d USING 1, 'test' END-EXEC.
EXEC SQL OPEN c1 USING SQL DESCRIPTOR mydesc END-EXEC.
EXEC SQL OPEN :curname1 END-EXEC.
```

互換性

OPENは標準SQLで規定されています。

関連項目

DECLARE, CLOSE

参照

CLOSEコマンドの詳細については、“PostgreSQL Documentation”の“Reference”の“SQL Commands”を参照してください。

D.11.10 PREPARE

名前

PREPARE -- 実行のためにSQL文をプリペアします。

記述形式

PREPARE *name* FROM *string*

説明

PREPAREは実行用に文字列として動的に指定されたSQL文をプリペアします。これは、埋め込みプログラム内でも使用することができる、直接的なSQL文とは異なります。EXECUTEコマンドを使用して、どちらの種類のプリペアード・ステートメントを実行することができます。

パラメータ

prepared_name

プリペアード・クエリ用の識別子です。

string

プリペアするSELECT/INSERT/UPDATE/DELETE文のいずれかを含む、文字列リテラルまたはホスト変数です。

使用例

```
MOVE "SELECT * FROM test1 WHERE a = ? AND b = ?" TO ARR OF STMT.
COMPUTE LEN OF STMT = FUNCTION STORED-CHAR-LENGTH (ARR OF STMT).
EXEC SQL ALLOCATE DESCRIPTOR indesc END-EXEC.
EXEC SQL ALLOCATE DESCRIPTOR outdesc END-EXEC.
EXEC SQL PREPARE foo FROM :STMT END-EXEC.

EXEC SQL EXECUTE foo USING SQL DESCRIPTOR indesc INTO SQL DESCRIPTOR outdesc END-EXEC.
```

互換性

PREPAREは標準SQLで規定されています。

関連項目

EXECUTE

参照

EXECUTEコマンドの詳細については、“PostgreSQL Documentation”の“Reference”の“SQL Commands”を参照してください。

D.11.11 SET AUTOCOMMIT

名前

SET AUTOCOMMIT -- 現在のセッションの自動コミット動作を設定します。

記述形式

```
SET AUTOCOMMIT { = | TO } { ON | OFF }
```

説明

SET AUTOCOMMITは現在のデータベースセッションの自動コミット動作を設定します。デフォルトでは埋め込みSQLプログラムは自動コミットモードでは*ありません*。このためCOMMITコマンドを必要なところで明示的に発行しなければなりません。このコマンドはセッションを、個々のSQL文それぞれが暗黙的にコミットされる、自動コミットモードに変更することができます。

互換性

SET AUTOCOMMITはPostgreSQL ECOBPGの拡張です。

D.11.12 SET CONNECTION

名前

SET CONNECTION -- データベース接続を選択します。

記述形式

SET CONNECTION [TO | =] *connection_name*

説明

SET CONNECTIONは、上書きされない限りすべてのコマンドが使用する、"現在の"データベース接続を設定します。

パラメータ

connection_name

CONNECTコマンドで確立したデータベース接続の名前です。

DEFAULT

接続をデフォルトの接続に設定します。

使用例

```
EXEC SQL SET CONNECTION TO con2 END-EXEC.
EXEC SQL SET CONNECTION = con1 END-EXEC.
```

互換性

SET CONNECTIONは標準SQLで規定されています。

関連項目

CONNECT, DISCONNECT

D.11.13 SET DESCRIPTOR

名前

SET DESCRIPTOR -- SQL記述子領域に情報を設定します。

記述形式

```
SET DESCRIPTOR descriptor_name descriptor_header_item = value [, ... ]

SET DESCRIPTOR descriptor_name VALUE number descriptor_item = value [, ...]
```

説明

SET DESCRIPTORはSQL記述子領域に値を投入します。その後、通常、記述子領域はプリペアード・クエリ実行においてパラメータをバインドするために使用されます。

このコマンドは2つの構文があります。最初の構文は、特定のデータと独立した、記述子の“ヘッダ”に適用します。2番目の構文は、番号で識別される特定のデータに値を割り当てます。

パラメータ

descriptor_name

記述子の名前です。

descriptor_header_item

設定するヘッダ情報項目を識別するトークンです。記述子項目数を設定するCOUNTのみが現在サポートされています。

number

設定する記述子項目の番号です。番号は1から数えます。

descriptor_item

記述子内のどの項目の情報を設定するかを識別するトークンです。サポートされる項目のリストについては“D.6.1 名前付きSQL記述子領域”を参照してください。

value

記述子項目に格納する値です。これはSQL定数またはホスト変数とすることができます。

使用例

```
EXEC SQL SET DESCRIPTOR indesc COUNT = 1 END-EXEC.
EXEC SQL SET DESCRIPTOR indesc VALUE 1 DATA = 2 END-EXEC.
EXEC SQL SET DESCRIPTOR indesc VALUE 1 DATA = :val1 END-EXEC.
EXEC SQL SET DESCRIPTOR indesc VALUE 2 DATA = 'some string', INDICATOR = :val1 END-EXEC.
EXEC SQL SET DESCRIPTOR indesc VALUE 2 INDICATOR = :val2null, DATA = :val2 END-EXEC.
```

互換性

SET DESCRIPTORは標準SQLで規定されています。

関連項目

ALLOCATE DESCRIPTOR, GET DESCRIPTOR

D.11.14 TYPE

名前

TYPE -- 新しいデータ型を定義します。

記述形式

TYPE *type_name* IS *ctype*

説明

TYPEコマンドは新しいCOBOLの型を定義します。これは宣言セクションにTYPEDEFを記述することと同じです。

ecobpgpgが-cオプション付きで実行された場合にのみこのコマンドは認識されます。

type_nameの項目には自動的にレベル番号01が付加されます。type_nameにレベル番号は不要です。集団項目を定義する場合、従属項目のレベル番号は設定してください。

パラメータ

type_name

新しい型の名前です。これは有効なCOBOLの型名でなければなりません。

ctype

COBOLの型指定です。(表現形式などの指定も含む)

使用例

```
EXEC SQL TYPE CUSTOMER IS
    02 NAME PIC X(50) VARYING.
    02 PHONE PIC S9(9) COMP. END-EXEC.

EXEC SQL TYPE CUST-IND IS
    02 NAME_IND PIC S9(4) COMP.
    02 PHONE_IND PIC S9(4) COMP. END-EXEC.

EXEC SQL TYPE INTARRAY IS
    02 INT PIC S9(9) OCCURS 20. END-EXEC.
EXEC SQL TYPE STR IS PIC X(50) VARYING. END-EXEC.
EXEC SQL TYPE STRING IS PIC X(10). END-EXEC.
```

以下にEXEC SQL TYPEを使用するプログラム例を示します。

```
EXEC SQL TYPE TT IS
    02 V PIC X(256) VARYING.
    02 I PIC S9(9) COMP. END-EXEC.

EXEC SQL TYPE TT-IND IS
    02 V-IND PIC S9(4) COMP.
    02 I-IND PIC S9(4) COMP. END-EXEC.

EXEC SQL BEGIN DECLARE SECTION END-EXEC.
    01 T TYPE TT.
    01 T-IND TYPE TT-IND.
EXEC SQL END DECLARE SECTION END-EXEC.

    EXEC SQL CONNECT TO testdb AS con1 END-EXEC.

    EXEC SQL SELECT current_database(), 256 INTO :T :T-IND LIMIT 1 END-EXEC.

    DISPLAY "t.v = " ARR OF V OF T.
    DISPLAY "t.i = " I OF T.

    DISPLAY "t_ind.v_ind = " V-IND OF T-IND.
    DISPLAY "t_ind.i_ind = " I-IND OF T-IND.

    EXEC SQL DISCONNECT con1 END-EXEC.
```

互換性

TYPEコマンドはPostgreSQLの拡張です。

D.11.15 WHENEVER

名前

WHENEVER -- SQL文により特定の分類の条件が発生する時に行う動作を指定します。

記述形式

```
WHENEVER { NOT FOUND | SQLERROR | SQLWARNING } action
```

説明

SQL実行の結果において特殊な状態(行がない、SQL警告またはSQLエラー)で呼び出される動作を定義します。

パラメータ

パラメータの説明については"D.7.1 コールバックの設定"を参照してください。

使用例

```
EXEC SQL WHENEVER NOT FOUND CONTINUE END-EXEC.
EXEC SQL WHENEVER SQLWARNING SQLPRINT END-EXEC.
EXEC SQL WHENEVER SQLWARNING DO "warn" END-EXEC.
EXEC SQL WHENEVER SQLERROR sqlprint END-EXEC.
EXEC SQL WHENEVER SQLERROR CALL "print2" END-EXEC.
EXEC SQL WHENEVER SQLERROR DO handle_error USING "select" END-EXEC.
EXEC SQL WHENEVER SQLERROR DO sqlnotice USING 0 1 END-EXEC.
EXEC SQL WHENEVER SQLERROR DO "sqlprint" END-EXEC.
EXEC SQL WHENEVER SQLERROR GOTO error_label END-EXEC.
EXEC SQL WHENEVER SQLERROR STOP END-EXEC.
```

以下は、結果セットを通したループ処理を扱うためにWHENEVER NOT FOUND GOTOを使用する典型的なアプリケーションです。

```
    EXEC SQL CONNECT TO testdb AS con1 END-EXEC.
    EXEC SQL ALLOCATE DESCRIPTOR d END-EXEC.
    EXEC SQL DECLARE cur CURSOR FOR SELECT current_database(), 'hoge', 256 END-EXEC.
    EXEC SQL OPEN cur END-EXEC.

*    when end of result set reached, break out of while loop
    EXEC SQL WHENEVER NOT FOUND GOTO NOTFOUND END-EXEC.

    PERFORM NO LIMIT
        EXEC SQL FETCH NEXT FROM cur INTO SQL DESCRIPTOR d END-EXEC
        ...
    END-PERFORM.

    NOTFOUND.
    EXEC SQL CLOSE cur END-EXEC.
    EXEC SQL COMMIT END-EXEC.
    EXEC SQL DEALLOCATE DESCRIPTOR d END-EXEC.
    EXEC SQL DISCONNECT ALL END-EXEC.
```

互換性

WHENEVERは標準SQLで規定されていますが、ほとんどの動作はPostgreSQLの拡張です。

D.12 PostgreSQLのクライアントアプリケーション

ここには、PostgreSQLクライアントアプリケーションとユーティリティについてのリファレンス情報があります。これらのコマンドがすべて汎用的なユーティリティであるという訳ではありません。一部は特定の権限を必要とします。これらアプリケーションの共通機能は、データベースサーバが稼働しているかどうかに依存しない、どのホストでも実行できるという点です。

コマンドラインから指定された場合、ユーザ名とデータベース名の大文字小文字は保持されます。空白文字や特殊文字がある場合は引用符付けが必要かもしれません。テーブル名やその他の識別子では文書化されていない限り大文字小文字は保持されませんので、引用符付けが必要かもしれません。

D.12.1 ecobpg

名前

ecobpg -- 埋め込みSQL用COBOLプリプロセッサを使用する

記述形式

ecobpg [*option*...] *file*...

説明

ecobpgは、COBOLプログラム用の埋め込みSQLプリプロセッサです。SQL呼び出しを特殊な関数呼び出しに置き換えることによって、埋め込みSQL文を含むCOBOLプログラムを、通常のCOBOLコードに変換します。これにより、出力ファイルは、COBOLコンパイラツールを使用して処理することができます。

ecobpgは、コマンドラインで指定される各入力ファイルを対応するCの出力ファイルに変換します。入力ファイルに.pcoという拡張子を付けておくと、出力ファイル名の拡張子が.cobに置き換えられるので便利です。入力ファイルの拡張子が.pcoでない場合、そのファイルのフルネームの末尾に.cobを追加したものが出力ファイル名となります。出力ファイル名は、-oオプションによって上書きすることもできます。

パラメータ

ecobpgは、以下のコマンドライン引数を受け付けます。

-c

SQLコードから有効なCOBOLコードを自動的に生成します。現在、このオプションはEXEC SQL TYPEに対して使用できます。

-I directory

追加のインクルード用パスを指定します。これは、EXEC SQL INCLUDEを使用してインクルードされるファイルを検索する際に使用されます。デフォルトでは.(現ディレクトリ)のみです。

-o filename

ecobpgが全ての出力を*filename*に書き込むことを指定します。

-f format

COBOLコードの記法を指定します。formatには以下のいずれかを指定します。指定がない場合はfixed(固定形式)が指定されたものとして扱います。

fixed

固定形式を指定します。B領域は72カラムまで指定できます。73カラム以降の文字はプレコンパイルされたソースから削除されます。

variable

可変形式を指定します。B領域は251カラムまで指定できます。252カラム以降の文字はプレコンパイルされたソースから削除されます。

-r option

実行時の動作を選択します。以下のいずれかを*option*として取ることができます。

prepare

すべての文を使用する前に準備します。libecpgは準備済みの文のキャッシュを保持し、再実行される場合に文を再利用します。キャッシュが満杯になった場合、libecpgは最も古い文を解放します。

questionmarks

互換性という理由でクエスチョンマークをプレースホルダとして許します。これは長い間デフォルトとして使用されていました。

-t

トランザクションの自動コミットを有効にします。このモードでは、各SQLコマンドは明示的なトランザクションブロックの内部にない限り、自動的にコミットされます。デフォルトのモードでは、EXEC SQL COMMITが発行された時にのみコマンドがコミットされます。

--varchar-with-named-member

VARCHAR型ホスト変数を変換するとき、プレフィックスとしてホスト変数の名前を付与します。

LENとARRの替わりに、(ホスト変数名)-LENと(ホスト変数名)-ARRが使用されます。

--bytea-with-named-member

bytea型ホスト変数を変換するとき、プレフィックスとしてホスト変数の名前を付与します。

LENとARRの替わりに、(ホスト変数名)-LENと(ホスト変数名)-ARRが使用されます。

-E encode

COBOLソースのエンコードを設定します。エンコードは“UTF8”、“SJIS”、“EUC_JP”のいずれかを設定することができます。

本オプションを省略した場合はロケールで処理します。

-v

バージョンやインクルード用パスなどの補足情報を表示します。

--version

ecobpgのバージョンを表示し、終了します。

-?
--help

ecobpgのコマンドライン引数の使用方法を表示し、終了します。

注釈

前処理されたCOBOLコードファイルをコンパイルする際、コンパイラはPostgreSQLのインクルードディレクトリ内にある登録集ファイルを検索できるようにしなければなりません。

SQLが埋め込まれたCOBOLプログラムには、リンカのライブラリに関するオプションを使用するなどして、libecpgライブラリをリンクする必要があります。

使用するシステムにおいて上記の2つに対応するディレクトリを調べるには、pg_configを使用します。

参照

pg_configの詳細については、“PostgreSQL Documentation”の“Reference”の“pg_config”を参照してください。

使用例

埋め込みSQLを使用したprog1.pcoというCOBOLソースファイルがある場合、次のコマンドを実行すれば、コンパイル可能なソースコードを作成することができます。ソースコードのコンパイル方法は、使用するコンパイラの情報を参照してください。

```
ecobpg prog1.pco
```

付録E　定量制限

定量制限は以下のとおりです。

表E.1 識別子の長さ

項目	定量制限
データベース名	63バイト以下(注1)(注2)
スキーマ名	63バイト以下(注1)(注2)
テーブル名	63バイト以下(注1)(注2)
ビュー名	63バイト以下(注1)(注2)
インデックス名	63バイト以下(注1)(注2)
テーブル空間名	63バイト以下(注1)(注2)
カーソル名	63バイト以下(注1)(注2)
関数名	63バイト以下(注1)(注2)
集約関数名	63バイト以下(注1)(注2)
トリガ名	63バイト以下(注1)(注2)
制約名	63バイト以下(注1)(注2)
変換名	63バイト以下(注1)(注2)
ロール名	63バイト以下(注1)(注2)
キャスト名	63バイト以下(注1)(注2)
照合順名	63バイト以下(注1)(注2)
符号化方式変換名	63バイト以下(注1)(注2)
ドメイン名	63バイト以下(注1)(注2)
拡張名	63バイト以下(注1)(注2)
演算子名	63バイト以下(注1)(注2)
演算子クラス名	63バイト以下(注1)(注2)
演算子族名	63バイト以下(注1)(注2)
書き換えルール名	63バイト以下(注1)(注2)
シーケンス名	63バイト以下(注1)(注2)
テキスト検索設定名	63バイト以下(注1)(注2)
テキスト検索辞書名	63バイト以下(注1)(注2)
テキスト検索パーサ名	63バイト以下(注1)(注2)
テキスト検索テンプレート名	63バイト以下(注1)(注2)
データ型名	63バイト以下(注1)(注2)
列挙型のラベル	63バイト以下(注1)(注2)

注1)サーバ文字セットの文字コードで換算した場合の文字列のバイト長です。

注2)63バイトを超えた長さの識別子が指定された場合、超えた分の文字を切り捨てて処理します。

表E.2 データベースオブジェクト

項目	定量制限
データベースの数	4,294,967,296個未満(注1)

項目	定量制限
スキーマの数	4,294,967,296個未満(注1)
テーブルの数	4,294,967,296個未満(注1)
ビューの数	4,294,967,296個未満(注1)
インデックスの数	4,294,967,296個未満(注1)
テーブル空間の数	4,294,967,296個未満(注1)
関数の数	4,294,967,296個未満(注1)
集約関数の数	4,294,967,296個未満(注1)
トリガの数	4,294,967,296個未満(注1)
制約の数	4,294,967,296個未満(注1)
変換の数	4,294,967,296個未満(注1)
ロールの数	4,294,967,296個未満(注1)
キャストの数	4,294,967,296個未満(注1)
照合順の数	4,294,967,296個未満(注1)
符号化方式変換の数	4,294,967,296個未満(注1)
ドメインの数	4,294,967,296個未満(注1)
拡張の数	4,294,967,296個未満(注1)
演算子の数	4,294,967,296個未満(注1)
演算子クラスの数	4,294,967,296個未満(注1)
演算子族の数	4,294,967,296個未満(注1)
書き換えルールの数	4,294,967,296個未満(注1)
シーケンスの数	4,294,967,296個未満(注1)
テキスト検索設定の数	4,294,967,296個未満(注1)
テキスト検索辞書の数	4,294,967,296個未満(注1)
テキスト検索パーサの数	4,294,967,296個未満(注1)
テキスト検索テンプレートの数	4,294,967,296個未満(注1)
データ型の数	4,294,967,296個未満(注1)
列挙型のラベルの数	4,294,967,296個未満(注1)
ALTER DEFAULT PRIVILEGES文で定義したデフォルトのアクセス権限の数	4,294,967,296個未満(注1)
ラージオブジェクトの数	4,294,967,296個未満(注1)
インデックスアクセスメソッドの数	4,294,967,296個未満(注1)

注1)すべてのデータベースオブジェクトの合計数を4,294,967,296個未満にする必要があります。

表E.3 スキーマ要素

項目	定量制限
1テーブルに定義可能な列数	250～1600個以下(データ型により異なります)
テーブルの行長	400ギガバイト以下
一意性制約を構成する列数	32列以下

項目	定量制限
一意性制約を構成するデータ長	2000バイト未満(注1)(注2)
テーブルのサイズ	32テラバイト以下
トリガ定義文の検索条件の文字列長	800メガバイト以下(注1)(注2)
項目のサイズ	1ギカバイト以下

注1)定量制限の範囲外で運用を行ったときでも、正しく動作する場合があります。

注2)サーバ文字セットの文字コードで換算した場合の文字列のバイト長です。

表E.4 インデックス

項目	定量制限
キー構成列数(VCI含む)	32列以下
キーの長さ(VCI以外)	2000バイト未満(注1)

注1)サーバ文字セットの文字コードで換算した場合の文字列のバイト長です。

表E.5 扱えるデータ種と属性

項目			定量制限
文字	データ長		1ギガバイト-53バイト以下(注1)
	指定長(n)		10,485,760文字以下(注1)
数字	外部10進表現		小数点前131072桁以下、小数点以降16383桁以下
	内部2進表現	2バイト	-32,768 から 32,767
		4バイト	-2,147,483,648から2,147,483,647
		8バイト	-9,223,372,036,854,775,808から9,223,372,036,854,775,807
	内部10進表現		小数点前131072桁以下、小数点以降16383桁以下
	浮動小数点表現	4バイト	-3.4E+38から-7.1E-46、0、7.1E-46から3.4E+38
		8バイト	-1.7E+308から-2.5E-324、0、2.5E-324から1.7E+308
bytea			1ギガバイト-53バイト以下
ラージオブジェクト			2ギガバイト以下

注1)サーバ文字セットの文字コードで換算した場合の文字列のバイト長です。

表E.6 関数定義

項目	定量制限
指定できる引数の数	100個以下
宣言部で指定できる変数名の数	無制限
関数の処理実装で指定できるSQL文または制御文の数	無制限

表E.7 データ操作文

項目	定量制限
アプリケーションの1プロセスあたりの最大コネクション数(リモートアクセス)	4000接続
選択リストに指定可能な式の数	1664個以下
FROM句に指定可能なテーブルの数	無制限
1つのSELECT文内の選択リスト/DISTINCT句/ORDER BY句/GROUP BY句に指定可能な一意の式の数	1664個以下
GROUP BY句に指定可能な式の数	無制限
ORDER BY句に指定可能な式の数	無制限
UNION句/INTERSECT句/EXCEPT句に指定可能なSELECT文の数	4000個以下(注1)
1つのビューに指定できる結合テーブルの入れ子の数	4000個以下(注1)
1つの式に指定できる関数または演算式の数	4000個以下(注1)
1つの行コンストラクタに指定できる式の数	1664個以下
UPDATE文のSET句に指定可能な式の数	1664個以下
VALUESリストの1行に指定可能な式の数	1664個以下
RETURNING句に指定可能な式の数	1664個以下
1つの関数指定の引数リストに指定可能な式の合計長	800メガバイト以下(注2)
1つのセッションで同時に処理可能なカーソル数	無制限
1つのSQL文の文字列長	800メガバイト以下(注1)(注3)
1つの動的SQL文に指定できる入力パラメータ指定の数	無制限
1つのSQL文に指定できるトークンの数	10000個以下
WHERE句のIN構文にリストとして指定できる値の数	無制限
USING句に指定できる式の数	無制限
結合テーブルに指定できるJOINの数	4000個以下(注1)
COALESCEに指定できる式の数	無制限
単純な形式または検索された形式のCASEに指定できるWHEN句の数	無制限
1つのSQL文で更新・挿入できる1レコードあたりのデータサイズ	1ギガバイト-53バイト以下
同時に共有ロック可能なオブジェクトの数	256,000個以下(注1)

注1)定量制限の範囲外で運用を行ったときでも、正しく動作する場合があります。

注2)すべてのデータベースオブジェクトの合計数を4,294,967,296個未満にする必要があります。

注3)サーバ文字セットの文字コードで換算した場合の文字列のバイト長です。

表E.8 データサイズ

項目	定量制限
入力データファイル(COPY文、psqlコマンドの¥copyメタコマンド)の1レコードあたりのデータサイズ	800メガバイト以下(注1)
出力データファイル(COPY文、psqlコマンドの¥copyメタコマンド)の1レコードあたりのデータサイズ	800メガバイト以下(注1)

注1)定量制限の範囲外で運用を行ったときでも、正しく動作する場合があります。

付録F　リファレンス

F.1　JDBCドライバ

参照

PostgreSQL JDBCドライバの詳細については、“Java APIリファレンス”を参照してください。

F.2　ODBCドライバ

F.2.1　API一覧

APIのサポート状況を以下に示します。

関数名	サポート状況
SQLAllocConnect	○
SQLAllocEnv	○
SQLAllocHandle	○
SQLAllocStmt	○
SQLBindCol	○
SQLBindParameter	○
SQLBindParam	○
SQLBrowseConnect	○
SQLBulkOperations	○
SQLCancel	○
SQLCancelHandle	×
SQLCloseCursor	○
SQLColAttribute	○
SQLColAttributeW	○
SQLColAttributes	○
SQLColAttributesW	○
SQLColumnPrivileges	○
SQLColumnPrivilegesW	○
SQLColumns	○
SQLColumnsW	○
SQLCompleteAsync	×
SQLConnect	○
SQLConnectW	○
SQLCopyDesc	○
SQLDataSources	○
SQLDataSourcesW	○

関数名	サポート状況
SQLDescribeCol	○
SQLDescribeColW	○
SQLDescribeParam	○
SQLDisconnect	○
SQLDriverConnect	○
SQLDriverConnectW	○
SQLDrivers	○
SQLEndTran	○
SQLError	○
SQLErrorW	○
SQLExecDirect	○
SQLExecDirectW	○
SQLExecute	○
SQLExtendedFetch	○
SQLFetch	○
SQLFetchScroll	○
SQLForeignKeys	○
SQLForeignKeysW	○
SQLFreeConnect	○
SQLFreeEnv	○
SQLFreeHandle	○
SQLFreeStmt	○
SQLGetConnectAttr	○
SQLGetConnectAttrW	○
SQLGetConnectOption	○
SQLGetConnectOptionW	○
SQLGetCursorName	○
SQLGetCursorNameW	○
SQLGetData	○
SQLGetDescField	○
SQLGetDescFieldW	○
SQLGetDescRec	○
SQLGetDescRecW	○
SQLGetDiagField	○
SQLGetDiagFieldW	○
SQLGetDiagRec	○
SQLGetDiagRecW	○
SQLGetEnvAttr	○
SQLGetFunctions	○

関数名	サポート状況
SQLGetInfo	○
SQLGetInfoW	○
SQLGetStmtAttr	○
SQLGetStmtAttrW	○
SQLGetStmtOption	○
SQLGetTypeInfo	○
SQLGetTypeInfoW	○
SQLMoreResults	○
SQLNativeSql	○
SQLNativeSqlW	○
SQLNumParams	○
SQLNumResultCols	○
SQLParamData	○
SQLParamOptions	○
SQLPrepare	○
SQLPrepareW	○
SQLPrimaryKeys	○
SQLPrimaryKeysW	○
SQLProcedureColumns	○
SQLProcedureColumnsW	○
SQLProcedures	○
SQLProceduresW	○
SQLPutData	○
SQLRowCount	○
SQLSetConnectAttr	○
SQLSetConnectAttrW	○
SQLSetConnectOption	○
SQLSetConnectOptionW	○
SQLSetCursorName	○
SQLSetCursorNameW	○
SQLSetDescField	○
SQLSetDescRec	○
SQLSetEnvAttr	○
SQLSetParam	○
SQLSetPos	○
SQLSetScrollOptions	×
SQLSetStmtAttr	○
SQLSetStmtAttrW	○
SQLSetStmtOption	○

関数名	サポート状況
SQLSpecialColumns	○
SQLSpecialColumnsW	○
SQLStatistics	○
SQLStatisticsW	○
SQLTablePrivileges	○
SQLTablePrivilegesW	○
SQLTables	○
SQLTablesW	○
SQLTransact	○

○: サポート
×: 未サポート

F.3 .NET Data Provider

Fujitsu Npgsql .NET Data Providerを利用して、アプリケーション開発するには2つの方法があります。

- Fujitsu Npgsql .NET Data ProviderのAPI(クラス・メソッド)を直接利用

  Fujitsu Npgsql .NET Data Provider は、オープンソース・ソフトウェアの Npgsql をベースに作成されています。各 API の詳細は、“Npgsql - .Net Data Provider for Postgresql”のドキュメントを参照してください。

  参照

  ベースとしている Npgsql のバージョンは、“導入ガイド(クライアント編)”を参照してください。

- .NETのSystem.Data.Common名前空間のAPIを利用

  System.Data.Common名前空間を利用するとプロバイダに依存しないアプリケーションを作成することができます。詳細は、MSDNライブラリの“ADO.NET でのプロバイダに依存しないコードの作成”を参照してください。

参照

Npgsql APIのクラスやメソッドの詳細については、“Npgsql APIリファレンス”を参照してください。

F.4 C言語用ライブラリ(libpq)

参照

“PostgreSQL Documentation”の“Client Interfaces”の“libpq - C Library”を参照してください。

F.5 C言語による埋め込みSQL

参照

“PostgreSQL Documentation”の“Client Interfaces”の“ECPG - Embedded SQL in C”を参照してください。