



JDBC Connector

リリース **42.2.9.1**

EDB Postgres™ Advanced Server JDBC Connector Guide

2020 年 05 月 21 日

 目次

第 1 章	新機能	2
第 2 章	要件の概要	3
2.1	サポートされているバージョン	3
2.2	サポートされているプラットフォーム	3
第 3 章	AdvancedServerJDBCConnectorOverview	5
3.1	JDBC ドライバーのタイプ	5
3.2	JDBC インターフェース	7
3.3	JDBC クラスとインターフェース	8
3.4	JBCDriverManager	9
3.5	AdvancedServerJDBC コネクタの互換性	10
第 4 章	JDBC コネクタのインストールと設定	12
4.1	RPM パッケージを使用したコネクタのインストール	13
4.1.1	RPM インストールの更新	14
4.2	SLES12 ホストへのコネクタのインストール	16
4.3	Debian または Ubuntu ホストへの DEB パッケージのインストール	18
4.4	グラフィカルインストーラを使用してコネクタをインストールする	19
4.5	AdvancedServerJDBCConnector の設定	24
第 5 章	Java アプリケーションでの AdvancedServerJDBCConnector の使用	25
5.1	AdvancedServerJDBC コネクタの読み込み	27
5.2	データベースへの接続	28
5.2.1	追加の接続プロパティ	29
5.2.2	同期セカンダリデータベースサーバーの設定	32
5.3	ステートメントオブジェクトによる SQL ステートメントの実行	39
5.3.1	CallableStatement オブジェクトで名前付き表記を使用する	40

5.4	ResultSet オブジェクトからの結果の取得	42
5.5	リソースの解放	43
5.6	エラーの処理	44
第 6 章	executeUpdate () による SQL コマンドの実行	45
6.1	executeUpdate () を使用してデータを挿入する	46
6.1.1	executeUpdate () の構文例	48
第 7 章	グラフィカルインターフェイスを Java プログラムに追加する	49
第 8 章	高度な JDBC コネクタ機能	53
8.1	クライアント側のリソース要件の削減	54
8.1.1	ステートメントオブジェクトのバッチサイズの変更	55
8.2	PreparedStatement を使用して SQL コマンドを送信する	56
8.3	ストアドプロシージャの実行	58
8.3.1	ストアドプロシージャの呼び出し	59
8.4	Java で REF カーソルを使用する	69
8.4.1	REFCURSOR を使用して ResultSet を取得する	69
8.5	Java で BYTEA データを使用する	72
8.5.1	BYTEA データを AdvancedServer データベースに挿入する	73
8.5.2	AdvancedServer データベースから BYTEA データを取得する	75
8.6	Java でのオブジェクトタイプとコレクションの使用	77
8.6.1	オブジェクトタイプの使用	78
8.6.2	コレクションを使用する	81
8.7	NoticeListener による非同期通知処理	84
第 9 章	セキュリティと暗号化	87
9.1	SSL の使用	87
9.1.1	サーバーの設定	88
9.1.2	クライアントの設定	89
9.1.3	SSLJDBC 接続のテスト	90
9.1.4	証明書認証の使用（パスワードなし）	92
9.2	ScramCompatibility	92
第 10 章	AdvancedServerJDBCConnectorLogging	93
10.1	接続プロパティを使用したロギングの有効化（静的）	93
10.2	logging.properties によるロギングの有効化（動的）	95
第 11 章	リファレンス-JDBC データ型	96
第 12 章	結論	98
索引		101

JDBC コネクタは、Java アプリケーションと AdvancedServer データベース間の接続を提供します。このガイドでは、JDBC コネクタの AdvancedServer 固有の機能を示すインストール手順、使用手順、および例を示します。

JDBC コネクタは Java で記述されており、Sun の JDK アーキテクチャに準拠しています。詳細については、[:ref: JDBC ドライバertype<jdbc_driver_types>](#)を参照してください。

JDBC コネクタは、PostgreSQL コミュニティドライバのすべての機能に基づいて構築され、サポートされています。ドライバの機能の詳細については、コミュニティのドキュメンテーション<<https://jdbc.postgresql.org/documentation/head/index.html>>を参照してください。

新機能

AdvancedServerJDBCConnector 42.2.9.1 を作成するために、次の機能が追加されます:

- EDBJDBC コネクタは、RedHatEnterpriseLinux および CentOS (x86_64) 8.x でサポートされるようになりました。

第 2

要件の概要

2.1 サポートされているバージョン

AdvancedServerJDBCCConnector は、AdvancedServer バージョン 9.4 以上で認定されています。

2.2 サポートされているプラットフォーム

AdvancedServerJDBCCConnector ネイティブパッケージは、次の 64 ビット Linux プラットフォームでサポートされています。

- RedHatEnterpriseLinux および CentOS (x86_64) 6.x、7.x、および 8.x。
- OELLinux6.x および 7.x
- RHEL または CentOS7.x を実行している PPC-LE8
- SLES12.x
- Debian9.x
- Ubuntu18.04

AdvancedServerJDBCCConnector グラフィカルインストーラーは、次の Windows プラットフォームでサポートされています。

64 ビット Windows:

- WindowsServer2019
- WindowsServer2016
- WindowsServer2012R2

32 ビット Windows:

- Windows10
- Windows8
- Windows7

第 3

AdvancedServerJDBCConnectorOverview

SunMicrosystems は、Java アプリケーションを JavaDatabaseConnectivity (JDBC) と呼ばれるデータベースに接続するための標準化されたインターフェースを作成しました。AdvancedServerJDBC コネクタは、Java アプリケーションを Postgres データベースに接続します。

3.1 JDBC ドライバーのタイプ

現在、4 種類の JDBC ドライバーがあり、それぞれ固有の実装、使用、制限があります。AdvancedServerJDBC コネクタは、タイプ 4 ドライバーです。

タイプ 1 ドライバー

- このドライバータイプは JDBC-ODBC ブリッジです。
- ローカルでの実行に制限されています。
- ODBC がコンピューターにインストールされている必要があります。
- 特定のデータベース用の ODBC ドライバーがコンピューターにインストールされている必要があります。
- 通常、ネイティブメソッド呼び出しのためにアプレット内で実行することはできません。

タイプ 2 ドライバー

- これは、ネイティブデータベースライブラリドライバーです。
- コンピューター上のネイティブデータベースライブラリを使用してデータベースにアクセスします。

- 通常、ネイティブメソッド呼び出しのためにアプレット内で実行することはできません。
- クライアントにデータベースライブラリがインストールされている必要があります。

タイプ 3 ドライバー

- 100 % Java ドライバー、ネイティブメソッドなし。
- クライアントに事前インストールする必要はありません。
- Java クラスファイルと同じように、ダウンロードしてすぐに設定できます。
- ミドルウェアサーバーとの通信に独自のプロトコルを使用します。
- ミドルウェアサーバーは、独自の呼び出しから DBMS 固有の呼び出しに変換します

タイプ 4 ドライバー

- 100 % Java ドライバー、ネイティブメソッドなし。
- クライアントに事前インストールする必要はありません。
- Java クラスファイルと同じように、ダウンロードしてすぐに設定できます。
- タイプ III ドライバーとは異なり、DBMS サーバーと直接通信します。
- JDBC 呼び出しをデータベース固有の呼び出しに直接変換します。

3.2 JDBC インターフェース

次の図は、JDBC 仕様のコア API インターフェースとそれらの相互関係を示しています。これらのインターフェースは `java.sql` パッケージに実装されています。

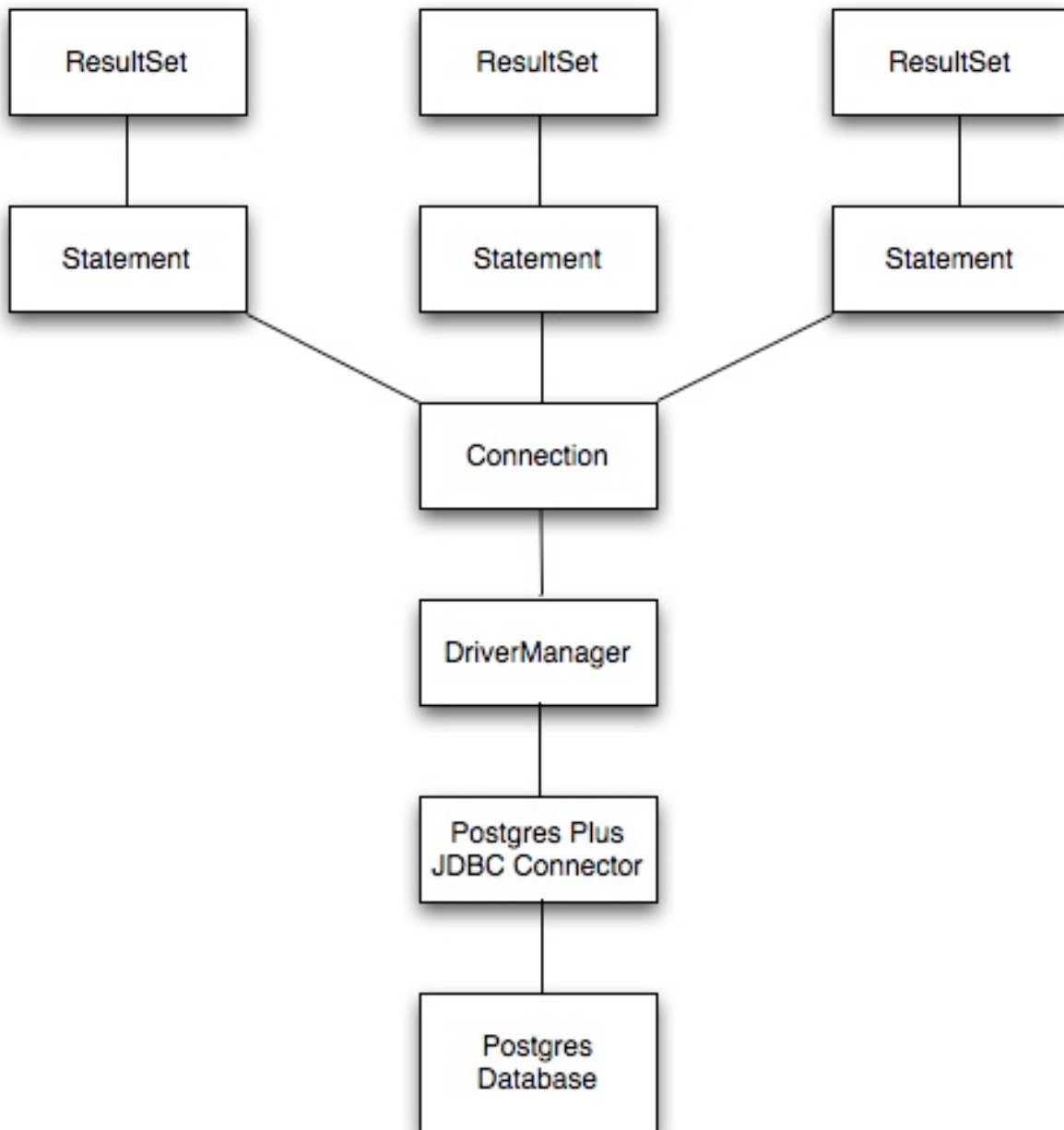


図 1 JDBC クラスの関係

3.3 JDBC クラスとインターフェース

コア API はクラスとインターフェースで構成されています。これらのクラスとインターフェースは、次のように連携して機能します。

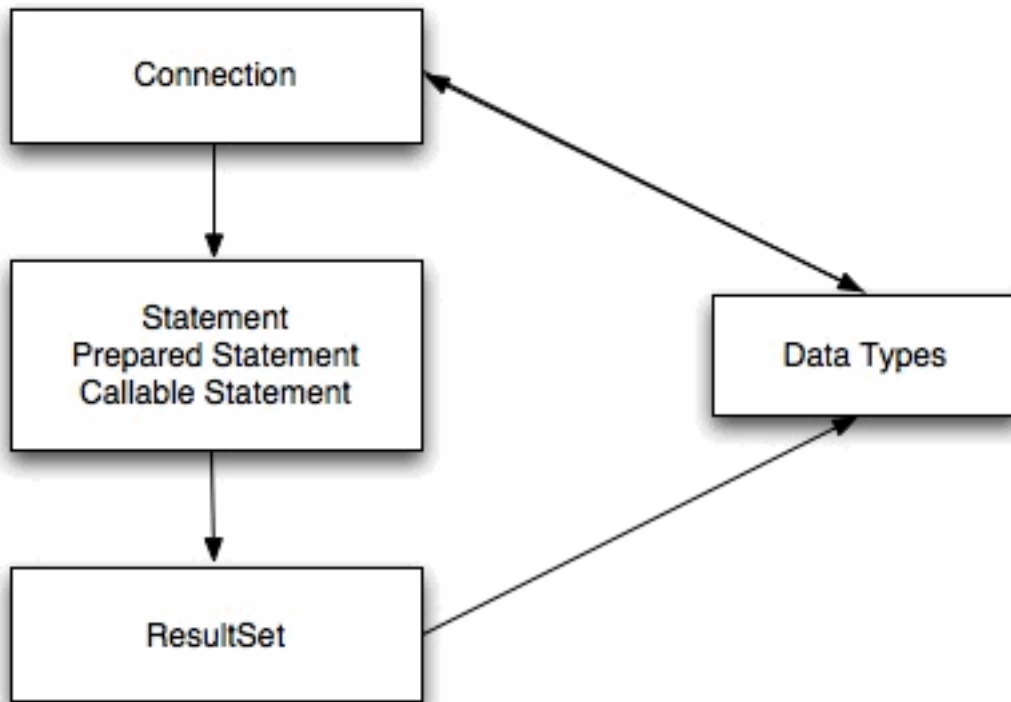


図 2 コアクラスとインターフェース

3.4 JDBCDriverManager

The figure below depicts the role of the `DriverManager` class in a typical JDBC application. The `DriverManager` acts as the bridge between a Java application and the backend database and determines which JDBC driver to use for the target database.

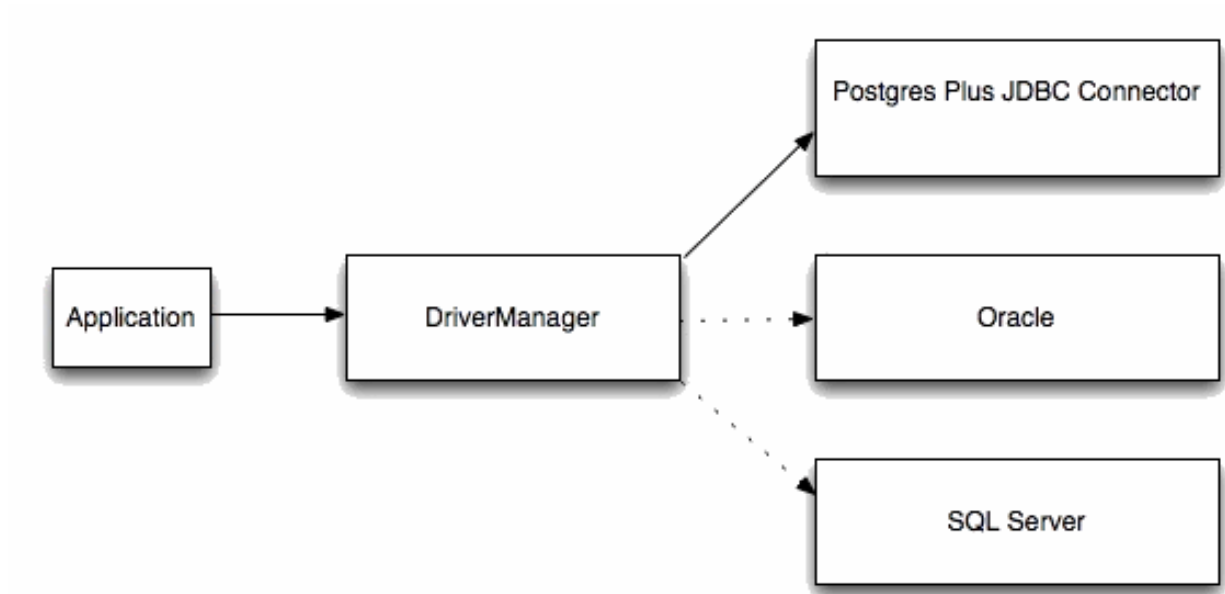


図 3 *DriverManager/Drivers*

3.5 AdvancedServerJDBC コネクタの互換性

EDB は、Java 仮想マシンの各バージョンに適切な JDBC ドライバーを提供することにより、複数の JRE/JDK バージョンのサポートを提供します。次の表を使用して、JRE/JDK バージョンと AdvancedServerDriver の互換性を確認してください。

表-高度なサーバー JDBC ドライバーの互換性

JRE/JDK バージョン	JDBC 仕様	AdvancedServerJDBC ドライバー	コミュニティ JDBC ドライバー
1.4、1.5（注 1 を参照）	3	edb-jdbc15.jar	postgresql-9.3.1103.JDBC3.jar
1.6（注 2 を参照）	4	edb-jdbc16.jar	postgresql-42.2.9.jre6.jar
1.7	4.1	edb-jdbc17.jar	postgresql-42.2.9.jre7.jar
1.8	4.2	edb-jdbc18.jar	postgresql-42.2.9.jar

注釈:

1. JDBC ドライバーファイル edb-jdbc15.jar が提供されなくなったため、JRE/JDK バージョン 1.4 および 1.5 は AdvancedServerJDBCConnector でサポートされなくなりました。
2. edb-jdbc16.jar ファイルは、PowerPC64 リトルエンディアン (ppc64le) 上の Linux、Debian/Ubuntu、および RHEL8 プラットフォームでは利用できません。
3. edb-jdbc17.jar ファイルは Debian/Ubuntu および RHEL8 プラットフォームでは利用できません。
4. コミュニティのバージョン番号は、PostgreSQL バージョンではなく、pgjdbc バージョンに基づいています。
5. AdvancedServerJDBC リリースは、EDBPostgresAdvancedServer リリースと分離されています。

次の JDBC 互換性ルールが適用されます。

- コミュニティ Web サイトから: PostgreSQL JDBC ドライバーには、そのためのコードの開発を開始する前に知っておくべきいくつかの固有のプロパティがあります。現在の開発ドライバーは、6 つのサーバーバージョンと 6 つの Java 環境をサポートしています。これは、すべての機能がすべての組み合わせで機能する必要があることを意味するものではありませんが、サポートされていないバージョンでは合理的な動作を提供する必要があります。この追加の互換性は多くの作業のように思えますが、実際の目標は、1 つのコードベースのみを維持することで作業量を削減することです。
- From the pgjdbc readme file: “PgJDBC regression tests are run against all PostgreSQL versions since 8.4, including “build PostgreSQL from git master” version. Don't assume pgjdbc 9.4.x is only PostgreSQL 9.4 compatible.”

サポートされているコミュニティバージョンについては、`PostgreSQLJDBCdriverWeb サイト<<https://jdbc.postgresql.org/download.html>>`を参照してください。

JDBC コネクタのインストールと設定

この章では、AdvancedServerJDBCConnector のインストールおよび構成方法について説明します。

JDBC コネクタをインストールする前に、システムに Java をインストールする必要があります。OracleJavaDownloads の `website<<http://www.oracle.com/technetwork/java/javase/downloads/index.html>>` から環境に合った Java インストーラーをダウンロードできます。詳細なインストール手順を含むドキュメントは、同じページの関連する Docs Installation Instruction リンクから入手できます。

AdvancedServer グラフィカルインストーラーまたは RPM パッケージを使用して、JDBC コネクタをインストールに追加できます。

次のセクションでは、これらのインストール方法について説明します。

4.1 RPM パッケージを使用したコネクタのインストール

JDBC Connector をインストールする前に、以下を行う必要があります。

epel-release パッケージをインストールします:

- RHEL または CentOS7 の場合:

```
yum -y install https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm
```

- RHEL または CentOS8 の場合:

```
dnf -y install https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm
```

注釈: CentOS-Base.repo ファイル (/etc/yum.repos.d にある) で [extras] リポジトリ定義を有効にする必要があるかもしれません。

EnterpriseDB リポジトリへのアクセスを許可する資格情報も必要です。資格情報のリクエストについては、次をご覧ください。

<https://info.enterprisedb.com/rs/069-ALB-339/images/Repository%20Access%2004-09-2019.pdf>

リポジトリの資格情報を受け取った後、次のことができます。

1. リポジトリ構成ファイルを作成します。
2. ユーザー名とパスワードを指定して、ファイルを変更します。
3. edb-jdbc をインストールします。

リポジトリ設定ファイルの作成

リポジトリ構成ファイルを作成するには、スーパーユーザー特権を引き受けて、次のコマンドを呼び出します。

- RHEL または CentOS7 の場合:

```
yum -y install https://yum.enterprisedb.com/edb-repo-rpms/edb-repo-latest.noarch.rpm
```

- RHEL または CentOS8 の場合:

```
dnf -y install https://yum.enterprisedb.com/edb-repo-rpms/edb-repo-latest.noarch.rpm
```

リポジトリ設定ファイルの名前は edb.repo です。ファイルは /etc/yum.repos.d にあります。

ファイルの変更、ユーザー名とパスワードの入力

After creating the `edb.repo` file, use your choice of editor to ensure that the value of the `enabled` parameter is 1, and replace the `username` and `password` placeholders in the `baseurl` specification with the name and password of a registered EnterpriseDB user.

```
[edb]
name=EnterpriseDB RPMs $releasever - $basearch
baseurl=https://<username>:<password>@yum.enterprisedb.com/edb/redhat/rhel-$releasever-
↪$basearch
enabled=1
gpgcheck=1
gpgkey=file:///etc/pki/rpm-gpg/ENTERPRISEDB-GPG-KEY
```

JDBC コネクタのインストール

設定ファイルへの変更を保存した後、`yum install` コマンドを使用して、RHEL または CentOS7 に JDBC Connector をインストールできます。

- RHEL または CentOS7 の場合:

```
yum install edb-jdbc
```

- RHEL または CentOS8 の場合:

```
dnf install edb-jdbc
```

システムで認識されないソースによって署名された RPM パッケージをインストールすると、`yum` はローカルサーバーにキーをインポートする許可を求め場合があります。プロンプトが表示され、パッケージが信頼できるソースからのものであることに満足したら、`y` を入力し、Return を押して続行します。

インストール中に、`yum` は解決できない依存関係に遭遇する場合があります。存在する場合、手動で解決する必要がある依存関係のリストが提供されます。

4.1.1 RPM インストールの更新

既存の JDBC Connector RPM インストールがある場合、`yum` または `dnf` を使用して、リポジトリ構成ファイルをアップグレードし、より新しい製品バージョンに更新できます。`edb.repo` ファイルを更新するには、スーパーユーザー権限を引き受けて次のように入力します:

- RHEL または CentOS7 の場合:

```
yum upgrade edb-repo
```

- RHEL または CentOS8 の場合:

```
dnf upgrade edb-repo
```

yum or dnf will update the `edb.repo` file to enable access to the current EDB repository, configured to connect with the credentials specified in your `edb.repo` file. Then, you can use yum or dnf to upgrade any installed packages:

- RHEL または CentOS7 の場合:

```
yum upgrade edb-jdbc
```

- RHEL または CentOS8 の場合:

```
dnf upgrade edb-jdbc
```

4.2 SLES12 ホストへのコネクタのインストール

zypper パッケージマネージャーを使用して、SLES12 ホストにコネクタをインストールできます。zypper は、パッケージをインストールするときにパッケージの依存関係を満たそうとしますが、EnterpriseDB でホストされていない特定のリポジトリへのアクセスが必要です。

コネクタをインストールする前に、次のコマンドを使用して EnterpriseDB リポジトリ構成ファイルを SLES ホストに追加します。

```
zypper addrepo https://zypp.enterprisedb.com/suse/epas12-sles.repo
zypper addrepo https://zypp.enterprisedb.com/suse/epas-sles-tools.repo
zypper addrepo https://zypp.enterprisedb.com/suse/epas-sles-dependencies.repo
```

各コマンドは、/etc/zypp/repos.d ディレクトリにリポジトリ設定ファイルを作成します。ファイルの名前は次のとおりです。

- edbas12suse.repo
- edbasdependencies.repo
- edbastools.repo

リポジトリ設定ファイルを作成したら、zypper refresh コマンドを使用して SLES ホストのメタデータを更新し、EnterpriseDB リポジトリを含めます。

```
/etc/zypp/repos.d # zypper refresh
Repository 'SLES12-12-0' is up to date.
Repository 'SLES12-Pool' is up to date.
Repository 'SLES12-Updates' is up to date.
Retrieving repository 'EDB Postgres Advanced Server 12 12 - x86_64'
metadata -----[\\]

Authentication required for
'https://zypp.enterprisedb.com/12/suse/suse-12-x86_64'
User Name:
Password:

Retrieving repository 'EDB Postgres Advanced Server 12 12 - x86_64'
metadata.....[done]
Building repository 'EDB Postgres Advanced Server 12 12 - x86_64'
cache.....[done]
All repositories have been refreshed.
...
```

User Name および Password の入力を求められたら、EnterpriseDB リポジトリの接続認証情報を入力します。資格情報が必要な場合は、`EnterpriseDB<<https://www.enterprisedb.com/repository-access-request/>>`_にお問い合わせください。

EDBPostgresAdvancedServer またはサポートコンポーネントをインストールする前に、SUSEConnect と SUSEPackageHub 拡張を SLES ホストに追加し、ホストを SUSE に登録して、SUSE リポジトリへのアクセスを許可する必要があります。次のコマンドを使用します。

```
zypper install SUSEConnect
SUSEConnect -p PackageHub/12/x86_64
SUSEConnect -p sle-sdk/12/x86_64
```

SUSE ホストの登録の詳細については、`SUSEWeb サイト<<https://www.suse.com/support/kb/doc/?id=7016626>>`_をご覧ください。

次に、zypper ユーティリティを使用してコネクタをインストールできます。

```
zypper install edb-jdbc
```

4.3 Debian または Ubuntu ホストへの DEB パッケージのインストール

Debian または Ubuntu ホストに DEB パッケージをインストールするには、EnterpriseDB リポジトリへのアクセスを許可する資格情報が必要です。リポジトリの資格情報を要求するには、EDBWeb サイト<<https://www.enterprisedb.com/repository-access-request>>`\_`にアクセスしてください。

次の手順では、EnterpriseDBapt リポジトリを使用して DEB パッケージをインストールする手順を説明します。コマンドを使用する場合、username と password を EnterpriseDB が提供する資格情報に置き換えます。

1. スーパーユーザー特権を想定します。

```
sudo su -
```

2. EnterpriseDB リポジトリを構成します。

```
sh -c 'echo /"deb https://username:password@apt.enterprisedb.com/
$(lsb_release -cs)-edb $(lsb_release -cs) main/" > /etc/apt/sources.
list.d/edb-$(lsb_release -cs).list'
```

3. 安全な APT リポジトリのサポートをシステムに追加します。

```
apt-get install apt-transport-https
```

4. EDB 署名キーを追加します。

```
wget -q -O - https://username:password@apt.enterprisedb.com/edb-deb.
gpg.key | apt-key add -
```

5. リポジトリのメタデータを更新します。

```
apt-get update
```

6. DEB パッケージをインストールします。

```
apt-get install edb-jdbc
```

注釈: デフォルトでは、Debian9x および Ubuntu18.04 プラットフォームは Java バージョン 10 をインストールします。EDBJava ベースのコンポーネントを実行するには、システムに Java バージョン 8 をインストールしてください。

4.4 グラフィカルインストーラを使用してコネクタをインストールする

EnterpriseDB コネクタインストールウィザードを使用して、システムに JDBC コネクタを追加できます。このウィザードは、EnterpriseDBAdvancedDownloads ページ<<https://www.enterprisedb.com/advanced-downloads>>から利用できます。

このセクションでは、インストールウィザードを使用して Windows システムにコネクタをインストールする方法を示します。(インストーラーをダウンロードし、インストーラーアイコンを右クリックして、コンテキストメニューから Run As Administrator を選択します。)

Language Selection ポップアップが開いたら、インストール言語を選択し、OK をクリックして Setup ウィンドウに進みます。

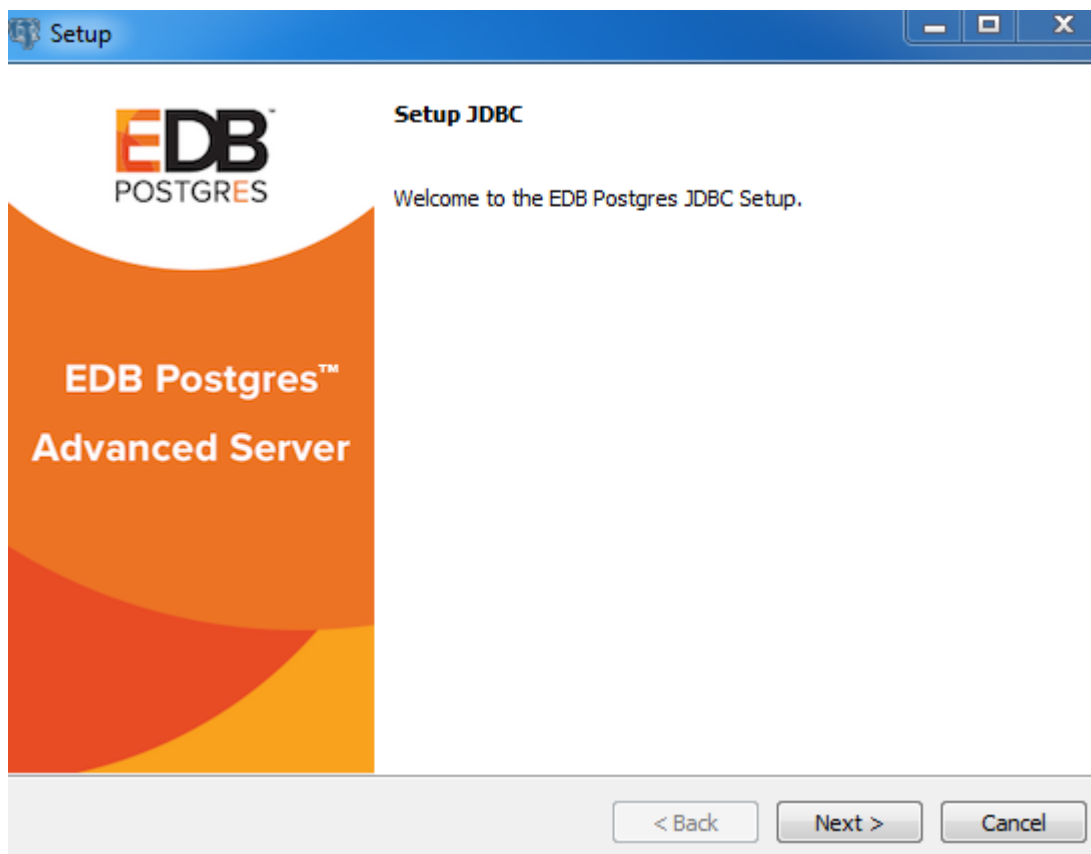


図 1 JDBC コネクタインストールウィザード

Next をクリックして続行します。

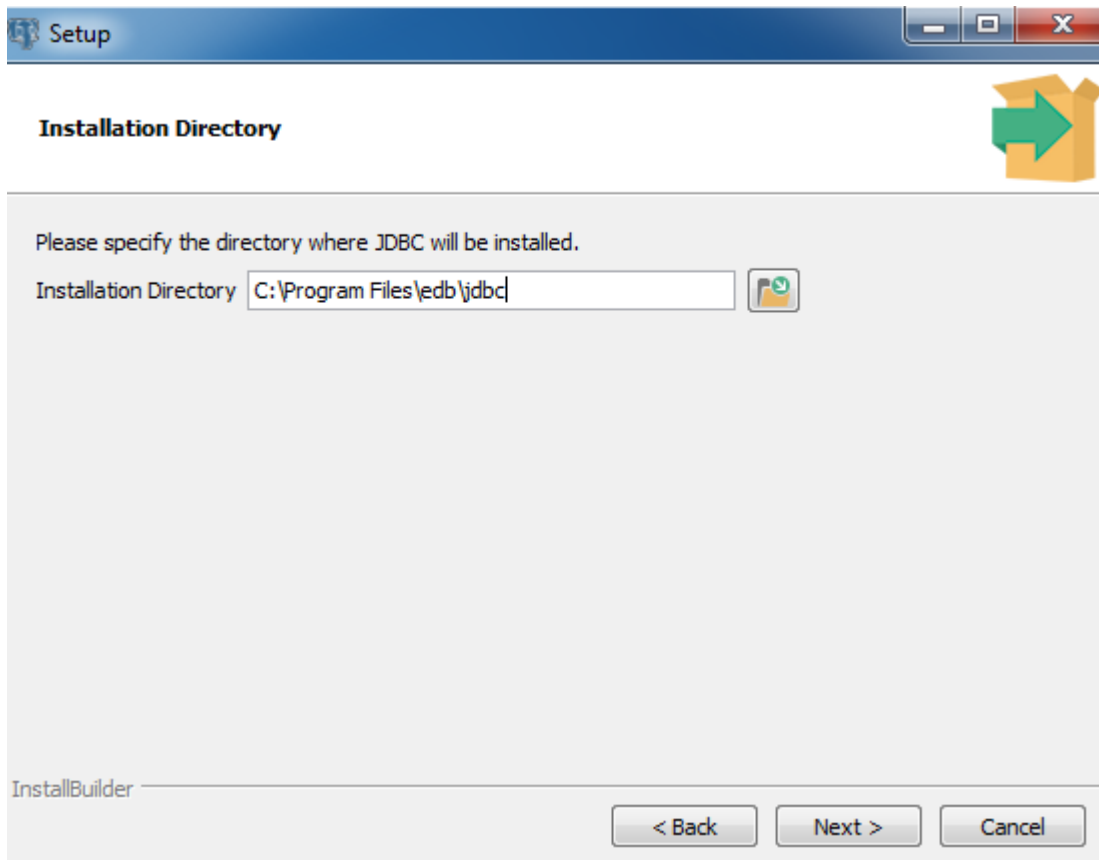


図 2 インストールダイアログ

Installation Directory ダイアログを使用してコネクタをインストールするディレクトリを指定し、Next をクリックして続行します。

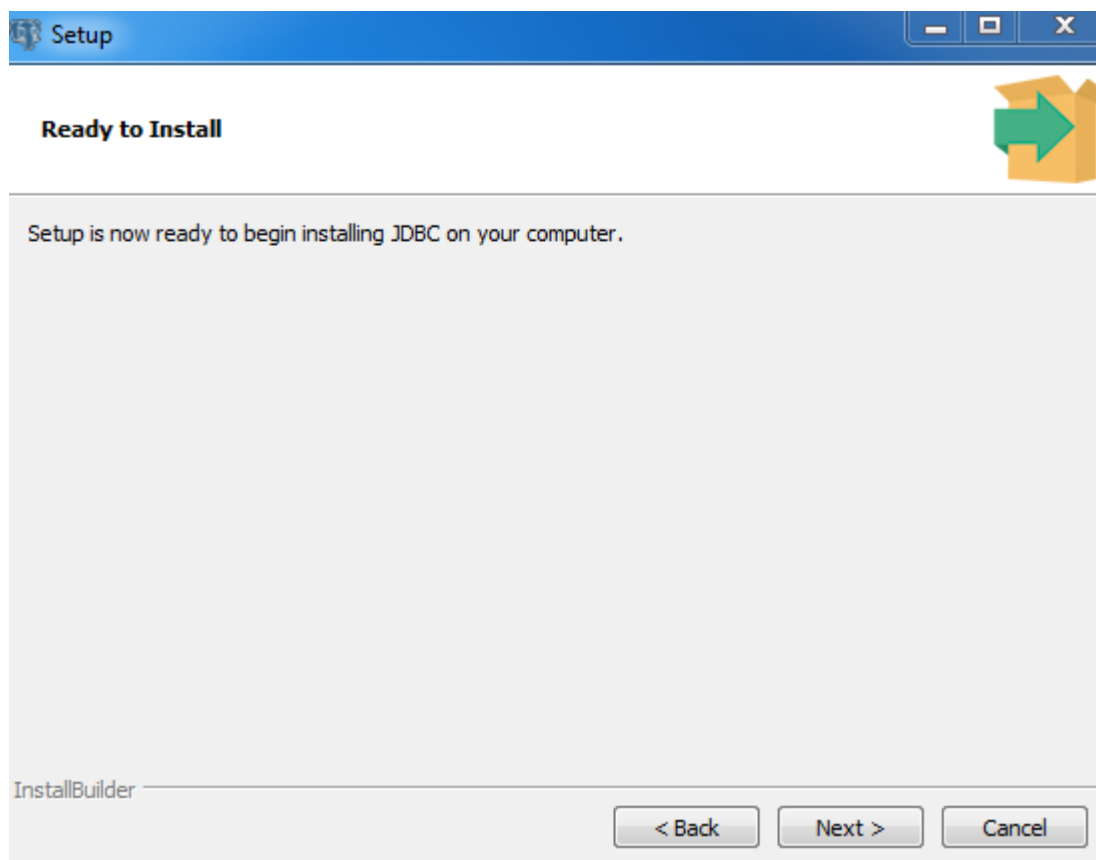


図 3 インストール準備ダイアログ

Ready to Install ダイアログで Next をクリックしてインストールを開始します。ポップアップダイアログがインストールウィザードの進行状況を確認します。

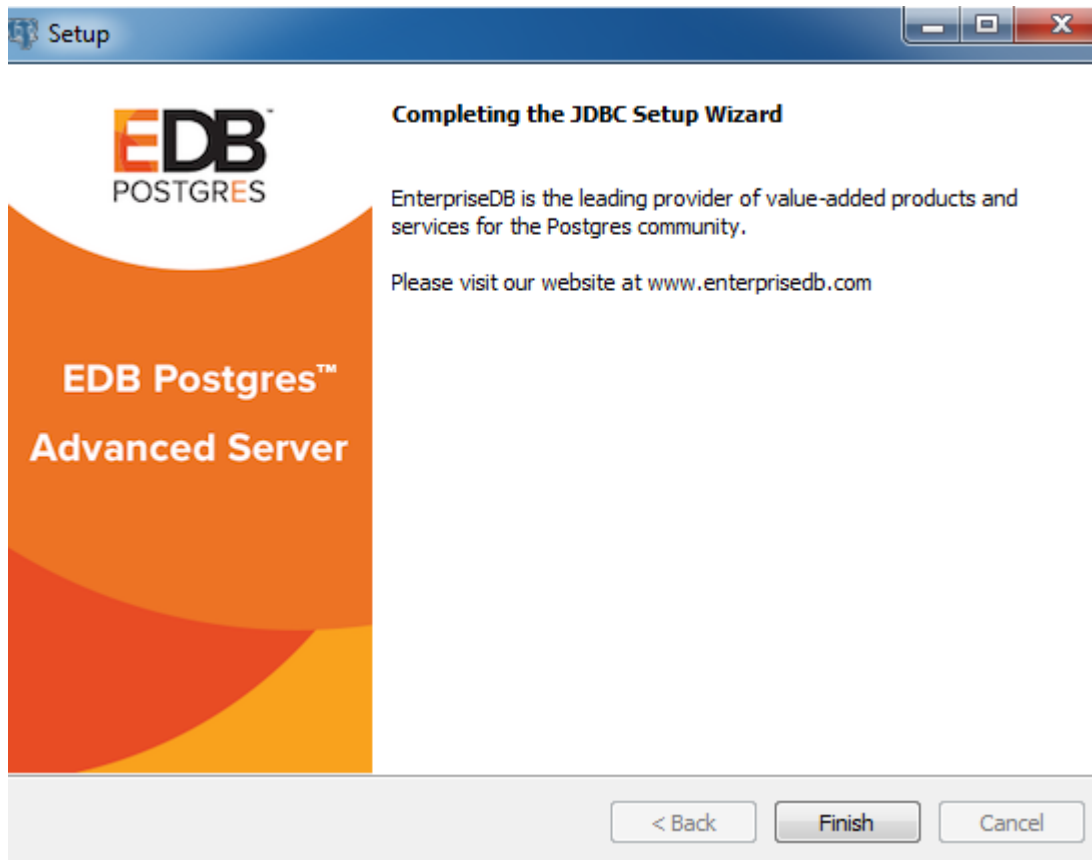


図 4 インストールが完了しました

ウィザードがセットアップの完了を通知したら、Finish ボタンをクリックしてダイアログを終了します。

StackBuilderPlus を使用して、既存の AdvancedServer インストールのコネクタを追加または更新することもできます。StackBuilderPlus を開くには、Windows の Apps メニューまたは Linux の Applications メニューから StackBuilderPlus を選択します。

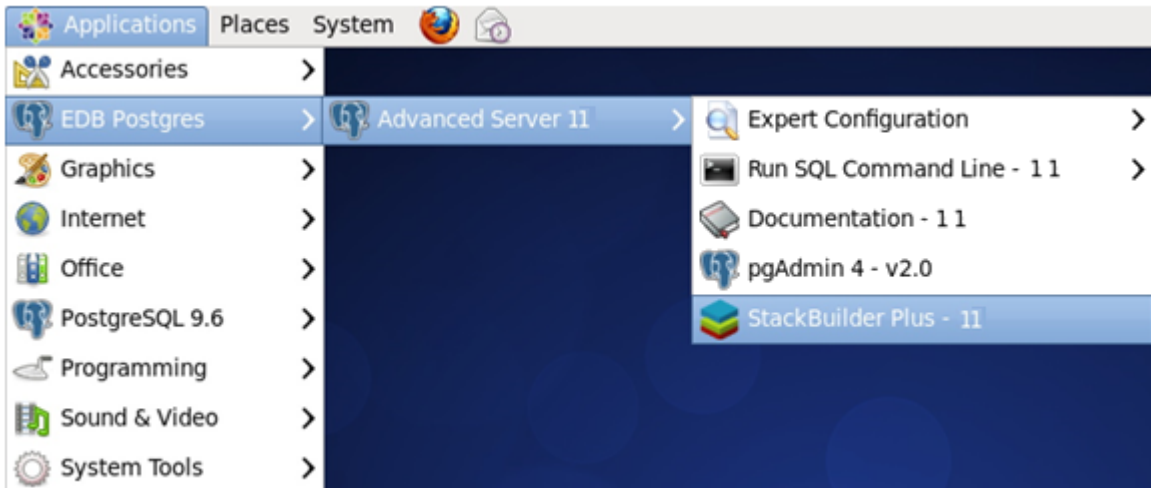


図 5 StackBuilderPlus の起動

StackBuilderPlus が開いたら、画面の指示に従います。ツリーコントロールの Database Drivers ノードから EnterpriseDB JDBC Connector オプションを選択します。

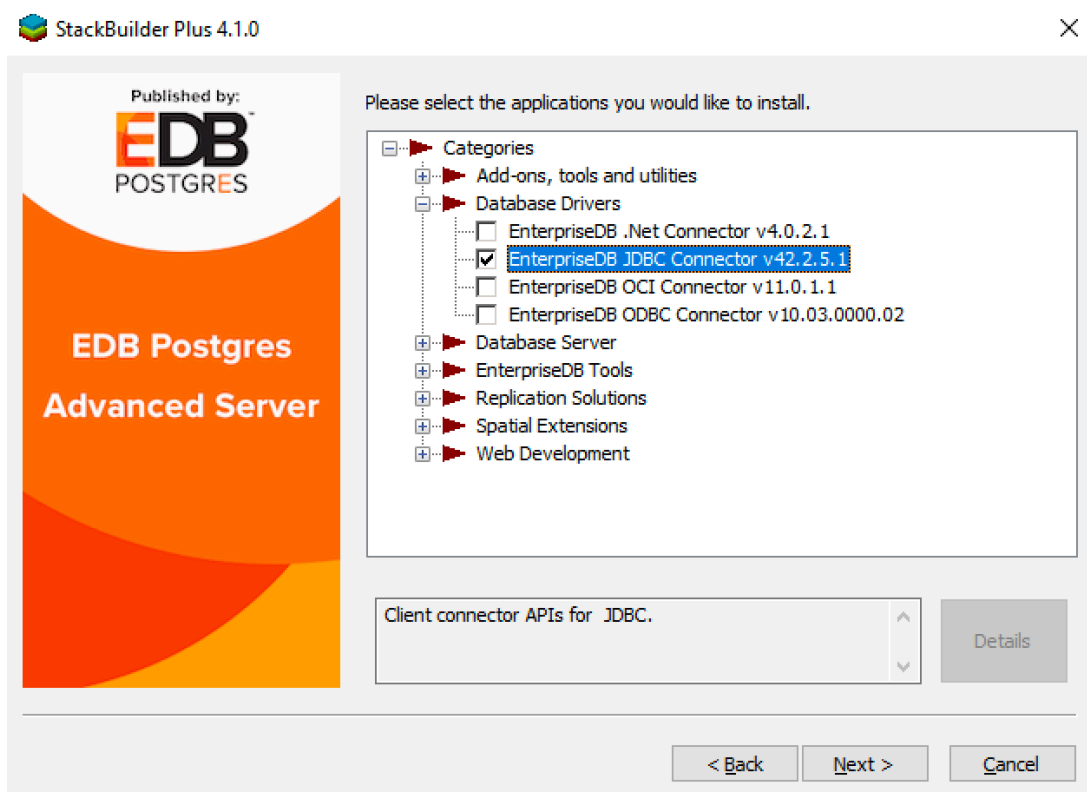


図 6 コネクタインストーラーの選択

オンスクリーンウィザードの指示に従って、EnterpriseDB コネクタのインストールを追加または更新します。

4.5 AdvancedServerJDBCConnector の設定

AdvancedServer には、次の JDBC ドライバーが付属しています。

- edb-jdbc16.jar は JDBC バージョン 4 をサポートします
- edb-jdbc17.jar は JDBC バージョン 4.1 をサポートします
- edb-jdbc18.jar は JDBC バージョン 4.2 をサポートします

注釈: edb-jdbc16.jar ファイルは LinuxonPowerPC64 リトルエンディアン (ppc64le) では利用できません。

JDBC ドライバーを Java で使用できるようにするには、使用している JDBC バージョンの適切な java .jar ファイルを \$java_home/jre/lib/ext ディレクトリーにコピーするか、.jar ファイルの場所を追加する必要があります CLASSPATH 環境変数に。

jar ファイルの場所を CLASSPATH 環境変数に追加する場合、完全なパス名を含める必要があることに注意してください。

```
/usr/edb/jdbc/edb-jdbcxx.jar
```

第 5

Java アプリケーションでの AdvancedServerJDBCConnector の使用

Java および AdvancedServerJDBCConnector を配置すると、Java アプリケーションは AdvancedServer データベースにアクセスできます。リスト 1.1 は、クエリを実行して結果セットを出力するアプリケーションを作成します。

リスト 1.1

```
import java.sql.*;
public class ListEmployees
{
    public static void main(String[] args)
    {
        try
        {
            Class.forName("com.edb.Driver");
            String url      = "jdbc:edb://localhost:5444/edb";
            String user     = "enterprisedb";
            String password = "enterprisedb";
            Connection con  = DriverManager.getConnection(url, user, password);
            Statement stmt  = con.createStatement();
            ResultSet rs    = stmt.executeQuery("SELECT * FROM emp");
            while(rs.next())
            {
                System.out.println(rs.getString(1));
            }

            rs.close();
            stmt.close();
            con.close();
        }
    }
}
```

(次のページに続く)

(前のページからの続き)

```
        System.out.println("Command successfully executed");
    }
    catch (ClassNotFoundException e)
    {
        System.out.println("Class Not Found : " + e.getMessage());
    }
    catch (SQLException exp)
    {
        System.out.println("SQL Exception: " + exp.getMessage());
        System.out.println("SQL State:      " + exp.getSQLState());
        System.out.println("Vendor Error:  " + exp.getErrorCode());
    }
}
}
```

この例は単純ですが、Java アプリケーションから AdvancedServer データベースと対話するために必要な基本的な手順を示しています。

- JDBC ドライバーをロードする
- 接続プロパティを構築する
- データベースサーバーに接続する
- SQL ステートメントを実行する
- 結果セットを処理する
- 片付け
- 発生する可能性のあるエラーを処理する

5.1 AdvancedServerJDBC コネクタの読み込み

The Advanced Server JDBC driver is written in Java and is distributed in the form of a compiled JAR (Java Archive) file. Use the `Class.forName()` method to load the driver. The `forName()` method dynamically loads a Java class at runtime. When an application calls the `forName()` method, the JVM (Java Virtual Machine) attempts to find the compiled form (the bytecode) that implements the requested class.

AdvancedServerJDBC ドライバーの名前は `com.edb.Driver` です:

```
Class.forName("/com.edb.Driver/");
```

ドライバーのバイトコードを読み込んだ後、ドライバーは、現在のシステムにインストールされているすべての JDBC ドライバーの管理を担当する別の JDBC クラス (`DriverManager` という名前) に自身を登録します。

JVM が名前付きドライバーを見つけることができない場合、`ClassNotFoundException` 例外 (プログラムの終わり近くで `catch` ブロックでインターセプトされます) をスローします。 `DriverManager` は、複数の JDBC ドライバーオブジェクトを処理するように設計されています。JDBC を介して複数のデータベースシステムに接続する Java アプリケーションを作成できます。次のセクションでは、特定のドライバーを選択する方法について説明します。

5.2 データベースへの接続

ドライバーが `DriverManager` で自身をロードして登録した後、次のコードフラグメントに示すように、`ListEmployees` クラスはデータベースサーバーへの接続を試行できます。

```
String url = "jdbc:edb://localhost:5444/edb";
String user = "enterprisedb";
String password = "enterprisedb";
Connection con = DriverManager.getConnection(url, user, password);
```

All JDBC connections start with the `DriverManager`. The `DriverManager` class offers a static method called `getConnection()` that is responsible for creating a connection to the database. When you call the `getConnection()` method, the `DriverManager` must decide which JDBC driver to use to connect to the database; that decision is based on a URL (Uniform Resource Locator) that you pass to `getConnection()`.

JDBCURL は次の一般的な形式を取ります。

```
jdbc:<driver>:<connection parameters>
```

JDBCURL の最初のコンポーネントは常に `jdbc` です。AdvancedServerJDBCCConnector を使用する場合、2 番目のコンポーネント（ドライバー）は `edb` です。

AdvancedServerJDBCURL は、次のいずれかの形式を取ります。

```
jdbc:edb:<database>
```

```
jdbc:edb://<host>/<database>
```

```
jdbc:edb://<host>:<port>/<database>
```

次の表に、さまざまな接続パラメーターを示します。

表-接続パラメーター

お名前	説明
ホスト	サーバーのホスト名。デフォルトは <code>localhost</code> です。
港	サーバーがリスンしているポート番号。デフォルトは AdvancedServer の標準ポート番号 (5444) です。
データベース	データベース名。

5.2.1 追加の接続プロパティ

In addition to the standard connection parameters, the Advanced Server JDBC driver supports connection properties that control behavior specific to EnterpriseDB. You can specify these properties in the connection URL or as a `Properties` object parameter passed to `DriverManager.getConnection()`. Listing 1.2 demonstrates how to use a `Properties` object to specify additional connection properties:

リスト 1.2

```
String url = "jdbc:edb://localhost/edb";
Properties props = new Properties();

props.setProperty("user", "enterprisedb");
props.setProperty("password", "enterprisedb");
props.setProperty("sslfactory", "com.edb.ssl.NonValidatingFactory");
props.setProperty("ssl", "true");

Connection con = DriverManager.getConnection(url, props);
```

注釈: By default the combination of `SSL=true` and setting the connection URL parameter `sslfactory=org.postgresql.ssl.NonValidatingFactory` encrypts the connection but does not validate the SSL certificate. To enforce certificate validation, you must use a Custom `SSLSocketFactory`. For more details about writing a Custom `SSLSocketFactory`, review [the PostgreSQL JDBC driver documentation](#).

URL に追加の接続プロパティを指定するには、疑問符とキーワードと値のペアのアンパサンドで区切られたリストを追加します。

```
String url = /"jdbc:edb://localhost/edb?user=enterprisedb&ssl=true/";
```

追加の接続プロパティの一部を次の表に示します。

表 5-2-追加の接続プロパティ

お名前	種類	説明
ユーザー	ひも	接続を確立するデータベースユーザー。
パスワード	ひも	データベースユーザーのパスワード。
ssl	ブール	認証済みの暗号化された SSL 接続を要求します
ログレベル	整数	loglevel の値は、LogStream または LogWriter の DriverManager の現在の値に出力される詳細の量を決定します。現在、次の値をサポートしています。 com.edb.Driver.DEBUG com.edb.Driver.INFO loglevel の値を INFO に設定してスパースログ情報を含めるか、DEBUG に設定して重要な詳細を生成します。
charSet	ひも	charSet の値は、データベースとの間で送受信されるデータに使用される文字セットを決定します。
prepareThreshold	整数	prepareThreshold の値は、サーバー側の準備済みステートメントに切り替える前に必要な PreparedStatement の実行回数を決定します。デフォルトは 5 です。
loadBalanceHosts	ブール	デフォルトモード（無効）では、ホストは指定された順序で接続されます。有効な場合、ホストは適切な候補のセットからランダムに選択されます。
targetServerType	ひも	必要な状態のサーバーのみへの接続を開くことができます。許可される値は、any、master、secondary、preferSecondary、preferSyncSecondary です。現在、マスター/セカンダリの区別は、サーバーが書き込みを許可しているかどうかを観察することによって行われます。値 preferSecondary は、使用可能なセカンダリがある場合はセカンダリへの接続を試み、それ以外の場合はマスターへの接続を許可します。AdvancedServerJDBCCConnector は preferSyncSecondary をサポートします。これは、同期セカンダリがアクティブでない場合、同期セカンダリまたはマスターへの接続のみを許可します。preferSyncSecondary については、セクション 5.2.2 を参照してください。注:値 slave、preferSlave、preferSyncSlave は、`セカンダリ`値に置き換えられたため、非推奨になりました。`スレーブ`値は現在もサポートされており、`スレーブ`と`セカンダリ`という用語の使用は同じ機能を提供しますが、`セカンダリ`値を使用することをお勧めします。
skipQuotesOnReturning	ブール	true に設定すると、RETURNING 句の列名は引用されません。これにより、大文字と小文字が区別される列名の比較がなくなります。false（デフォルト設定）に設定すると、列名が引用されます。
5.2. データベースへの接続		

5.2.2 同期セカンダリデータベースサーバーの設定

AdvancedServerJDBC コネクタは、表 5-2 に示されているように、`targetServerType` 接続プロパティの `preferSyncSecondary` オプションをサポートしています。

注釈: The `targetServerType` values `slave`, `preferSlave`, and `preferSyncSlave` all provide the same corresponding set of functionality as the `secondary`, `preferSecondary`, and `preferSyncSecondary` values; the `slave` values have been deprecated. We advise you to use the `secondary` values.

`preferSyncSecondary` オプションは、フェイルオーバー接続用の同期スタンバイサーバーを優先するため、非同期サーバーを無視します。

接続 URL でのこの機能の仕様は、次の構文で示されます。

```
jdbc:edb://master:port,secondary_1:port_1,secondary_2:port_2,.../  
database?targetServerType=preferSyncSecondary
```

パラメータ

`master:port`

マスターデータベースサーバーに割り当てられた IP アドレスまたは名前の後にポート番号が続きます。master が名前の場合、Java プログラムを実行しているホスト上の `/etc/hosts` ファイルで IP アドレスとともに指定する必要があります。注: マスターデータベースサーバーは、リスト内の任意の場所で指定できます。セカンダリデータベースサーバーの前に置く必要はありません。

`secondary_n:port_n`

スタンバイのセカンダリデータベースサーバーに割り当てられた IP アドレスまたは名前の後にポート番号が続きます。secondary_n が名前の場合、Java プログラムを実行しているホスト上の `/etc/hosts` ファイルで IP アドレスとともに指定する必要があります。

`database`

接続先のデータベースの名前。

以下は、接続 URL の例です。

```
String url = "jdbc:edb://master:5300,secondary1:5400/edb?  
↳targetServerType=preferSyncSecondary";  
con = DriverManager.getConnection(url, "enterprisedb", "edb");
```

以下の特性が `preferSyncSecondary` オプションに適用されます:

- マスターデータベースサーバーは、接続リストの任意の場所で指定できます。

- Java プログラムで使用するためにデータベースにアクセスするための接続は、同期セカンダリで最初に試行されます。セカンダリサーバーは読み取り専用操作に使用できます。
- 非同期モードで実行されているサーバーへの接続は試行されません。
- 接続試行が行われる順序は、表 5-2 で説明されている `loadBalanceHosts` 接続プロパティによって決定されます。デフォルト設定である無効にすると、接続リストで指定された左から右の順に接続が試行されます。有効にすると、接続の試行がランダムに行われます。
- 同期セカンダリに接続できない場合、マスターデータベースサーバーへの接続が使用されます。マスターデータベースサーバーがアクティブでない場合、接続の試行は失敗します。

`preferSyncSecondary` オプションに使用される同期セカンダリは、ホットスタンバイで使用するよう構成する必要があります。

次のセクションでは、ホットスタンバイ、同期レプリケーション用のマスターデータベースサーバーとセカンダリデータベースサーバーのセットアップの概要を簡単に説明します。

マスターおよびセカンダリデータベースサーバーの構成の概要

マスターデータベースサーバーとセカンダリデータベースサーバーを構成するプロセスは、PostgreSQL のドキュメントで説明されています。

For general information on hot standby usage, which is needed for the `preferSyncSecondary` option, see [the PostgreSQL core documentation](#).

マスターからセカンダリデータベースサーバーのベースバックアップを作成する方法については、セクション 25.3.2、[*ベースバックアップの作成*](#) (`pg_basebackup` ユーティリティプログラムの使用方法について説明) またはセクション 25.3.3、[*ベースバックアップの作成*](#) `ThePostgreSQLCoreDocumentation` (<https://www.postgresql.org/docs/12/static/continuous-archiving.html>) のセクション 25.3 `*ContinuousArchivingandPoint-in-TimeRecovery (PITR) *内の低レベル API*`。

ホットスタンバイで使用するために設定する必要がある設定パラメーターについては、[セクション 19.6、レプリケーション](#) (<https://www.postgresql.org/docs/12/static/runtime-config-replication.html>) を参照してください。

次のセクションでは、マスターデータベースサーバーとセカンダリデータベースサーバーをセットアップする基本的な例を示します。

例:マスターおよびセカンダリデータベースサーバー

次の例では、次のとおりです。

- マスターデータベースサーバーはホスト 192.168.2.24、ポート 5444 にあります
- セカンダリデータベースサーバーの名前は secondary1 で、ホスト 192.168.2.22、ポート 5445 に存在します
- セカンダリデータベースサーバーの名前は secondary2 であり、ホスト 192.162.2.24、ポート 5446 にあります（マスターと同じホスト）

マスターデータベースサーバーの pg_hba.conf ファイルには、すべてのセカンダリデータベースサーバーの一意のレプリケーションデータベース USER/ADDRESS の組み合わせごとにレプリケーションエントリが必要です。次の例では、データベーススーパーユーザー enterprisedb が、192.168.2.22 上の secondary1 データベースサーバーと、ローカルにある secondary2 データベースサーバーの両方のレプリケーションデータベースユーザーとして使用されます。マスター。

#	TYPE	DATABASE	USER	ADDRESS	METHOD
host	replication		enterprisedb	192.168.2.22/32	md5
host	replication		enterprisedb	127.0.0.1/32	md5

マスターデータベースサーバーが pg_hba.conf ファイルとともに postgresql.conf ファイルで設定された後、データベースサーバー secondary1 は ` のホスト 192.168.2.22 で次のコマンドを呼び出して作成されます 2":

```
su - enterprisedb
Password:
-bash-4.1$ pg_basebackup -D /opt/secondary1 -h 192.168.2.24 -p 5444 -Fp -R -X stream -
↪l 'Secondary1'
```

セカンダリデータベースサーバー /opt/secondary1 では、recovery.conf ファイルがデータベースクラスターに生成されます。これは、primary_conninfo の一部として application_name=secondary1 設定を追加することにより、次の例で編集されています ` pg\_basebackup` によって自動的に生成される他の不要なオプションの一部の文字列と削除。 standby_mode = 'on' パラメーターの使用にも注意してください。

```
standby_mode = 'on'
primary_conninfo = 'user=enterprisedb password=password host=192.168.2.24 port=5444_
↪application_name=secondary1'
```

アプリケーション名 secondary1 は、マスターデータベースサーバーの postgresql.conf ファイルの synchronous_standby_names パラメーターに含める必要があります。

セカンダリデータベースサーバー（secondary2）は、マスターが使用する同じホスト上に別の方法で作成されます。

```

su - enterprisedb
Password:
-bash-4.1$ psql -d edb -c "SELECT pg_start_backup('Secondary2') "
Password:
pg_start_backup
-----
0/6000028
(1 row)

-bash-4.1$ cp -rp /var/lib/edb/as12/data/opt/secondary2
-bash-4.1$ psql -d edb -c "SELECT pg_stop_backup() "
Password:
NOTICE:  pg_stop_backup complete, all required WAL segments have been archived
pg_stop_backup
-----
0/6000130
(1 row)

```

セカンダリデータベースサーバー /opt/secondary2 で、データベースクラスターに recovery.conf ファイルを作成します。次の例に示すように、primary_conninfo 文字列の一部として application_name=secondary2 設定に注意してください。また、standby_mode = 'on' パラメータを必ず含めてください。

```

standby_mode = 'on'
primary_conninfo = 'user=enterprisedb password=password host=localhost port=5444_
↪application_name=secondary2'

```

アプリケーション名 secondary2 は、マスターデータベースサーバーの postgresql.conf ファイルの synchronous_standby_names パラメーターに含める必要があります。

セカンダリデータベースサーバーの postgresql.conf ファイルの構成パラメーター設定が適切に設定されていることを確認する必要があります（特に hot_standby=on）。

注釈: EDBPostgresAdvancedServerV12 では、recovery.conf ファイルは無効になりました。standby.signal ファイルに置き換えられます。その結果、primary_conninfo は recovery.conf ファイルから postgresql.conf ファイルに移動します。standby.signal ファイルの存在は、スタンバイモードで実行するようにクラスターに通知します。手動で recovery.conf ファイルを作成して data ディレクトリに保存しようとしても、サーバーは起動に失敗してエラーをスローすることに注意してください。

パラメーター standby_mode=on は EDBPostgresAdvancedServerV12 から削除され、trigger_file パラメーター名は promote_trigger_file に変更されます。

次の表に、セカンダリデータベースサーバーと比較した、マスターデータベースサーバーの基本的な postgresql.conf 構成パラメーター設定を示します。

表-マスター/セカンダリ構成パラメーター

パラメータ	マスター	二次	説明
archive_mode	に	オフ	アーカイブストレージに送信された完了した WAL セグメント
archive_command	cp % p/archiver_dir/ % f	該当なし	完了した WAL セグメントをアーカイブする
wal_level	hot_standby (9.5 以前)、レプリカ (9.6 以降)	最小限の	WAL セグメントに書き込まれる情報
max_wal_senders	*n* (正の整数)	0	スタンバイサーバーからの最大同時接続
wal_keep_segments	*n* (正の整数)	0	スタンバイサーバー用に保持する過去のログセグメントの最小数
synchronous_standby_names	nl (secondary1、secondary2、...)	該当なし	同期レプリケーションのスタンバイサーバーのリスト。同期レプリケーションを有効にするには、存在する必要があります。これらは、各スタンバイサーバーの recovery.conf ファイルの primary_conninfo パラメータの application_name オプションから取得されます。
hot_standby	オフ	に	クライアントアプリケーションは、スタンバイモードでセカンダリサーバーに接続してクエリを実行できます。

セカンダリデータベースサーバー (secondary1) が起動します:

```
-bash-4.1$ pg_ctl start -D /opt/secondary1 -l logfile -o "-p 5445"
server starting
```

セカンダリデータベースサーバー (secondary2) が起動します:

```
-bash-4.1$ pg_ctl start -D /opt/secondary2/data -l logfile -o "-p 5446"
server starting
```

セカンダリデータベースサーバーが同期モードで適切にセットアップされていることを確認するには、マスターデータベースサーバーで次のクエリを使用します。sync_state 列には、アプリケーション secondary1 および secondary2 が同期としてリストされていることに注意してください。

```
edb=# SELECT username, application_name, client_addr, client_port, sync_state FROM pg_
↳stat_replication;
```

username	application_name	client_addr	client_port	sync_state
enterprisedb	secondary1	192.168.2.22	53525	sync
enterprisedb	secondary2	127.0.0.1	36214	sync

(2 rows)

接続 URL は次のとおりです。

```
String url = "jdbc:edb://master:5444,secondary1:5445,secondary2:5446/edb?
↳targetServerType=preferSyncSecondary";
con = DriverManager.getConnection(url, "enterprisedb", "password");
```

Java プログラムを実行しているホスト上の /etc/hosts ファイルには、接続 URL 文字列で指定されたサーバー名を持つ次のエントリが含まれます。

```
192.168.2.24      localhost.localdomain master
192.168.2.22      localhost.localdomain secondary1
192.168.2.24      localhost.localdomain secondary2
```

For this example, the preferred synchronous secondary connection option results in the first usage attempt made on secondary1, then on secondary2 if secondary1 is not active, then on the master if both secondary1 and secondary2 are not active as demonstrated by the following program that displays the IP address and port of the database server to which the connection is made.

```
import java.sql.*;
public class InetServer
{
    public static void main(String[] args)
    {
        try
        {
            Class.forName("com.edb.Driver");
            String url =
"jdbc:edb://master:5444,secondary1:5445,secondary2:5446/edb?
↳targetServerType=preferSyncSecondary";
            String user = "enterprisedb";
            String password = "password";
            Connection con = DriverManager.getConnection(url, user, password);

            ResultSet rs = con.createStatement().executeQuery("SELECT inet_server_addr() ||
↳':' || inet_server_port()");
```

(次のページに続く)

(前のページからの続き)

```
rs.next();
System.out.println(rs.getString(1));

rs.close();
con.close();
System.out.println("Command successfully executed");
}
catch(ClassNotFoundException e)
{
    System.out.println("Class Not Found : " + e.getMessage());
}
catch(SQLException exp)
{
    System.out.println("SQL Exception: " + exp.getMessage());
    System.out.println("SQL State:      " + exp.getSQLState());
    System.out.println("Vendor Error:   " + exp.getErrorCode());
}
}
}
```

ケース 1: すべてのデータベースサーバーがアクティブな場合、192.168.2.22 ポート 5445 の secondary1 に接続されます。

```
$ java InetServer
192.168.2.22/32:5445
Command successfully executed
```

ケース 2: secondary1 がシャットダウンされると、192.168.2.24 ポート 5446 の secondary2 に接続されます。

```
bash-4.1$ /usr/edb/as12/bin/pg_ctl stop -D /opt/secondary1
waiting for server to shut down.... done
server stopped

$ java InetServer
192.168.2.24/32:5446
Command successfully executed
```

ケース 3: secondary2 もシャットダウンされると、192.168.2.24 ポート 5444 の master に接続されます。

```
bash-4.1$ /usr/edb/as12/bin/pg_ctl stop -D /opt/secondary2/data
waiting for server to shut down.... done
server stopped

$ java InetServer
192.168.2.24/32:5444
Command successfully executed
```

5.3 ステートメントオブジェクトによる SQL ステートメントの実行

`AdvancedServerJDBCCConnector` ドライバーを読み込んでサーバーに接続した後、サンプルアプリケーションのコードは `JDBC Statement` オブジェクトを構築し、SQL クエリを実行し、結果を表示します。

`Statement` オブジェクトは SQL ステートメントをデータベースに送信します。`Statement` オブジェクトには 3 種類あります。それぞれが特定のタイプの SQL ステートメントを送信するために特化されています。

- `Statement` オブジェクトは、パラメーターなしで単純な SQL ステートメントを実行するために使用されます。
- `PreparedStatement` オブジェクトは、IN パラメーターの有無にかかわらず、プリコンパイルされた SQL ステートメントを実行するために使用されます。
- `CallableStatement` オブジェクトは、データベースストアードプロシージャの呼び出しを実行するために使用されます。

You must construct a `Statement` object before executing an SQL statement. The `Statement` object offers a way to send a SQL statement to the server (and gain access to the result set). Each `Statement` object belongs to a `Connection`; use the `createStatement()` method to ask the `Connection` to create the `Statement` object.

`Statement` オブジェクトは、さまざまなタイプの SQL ステートメントを実行するためのいくつかのメソッドを定義します。サンプルアプリケーションでは、`executeQuery()` メソッドは `SELECT` ステートメントを実行します:

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM emp");
```

The `executeQuery()` method expects a single argument: the SQL statement that you want to execute. `executeQuery()` returns data from the query in a `ResultSet` object. If the server encounters an error while executing the SQL statement provided, it throws an `SQLException` (and does not return a `ResultSet`).

5.3.1 CallableStatement オブジェクトで名前付き表記を使用する

JDBC コネクタ（AdvancedServer バージョン 9.6 以降）は、CallableStatement オブジェクトをインスタンス化する際に名前付きパラメーターの使用をサポートします。この構文は、JDBC がサポートする構文の拡張であり、JDBC 標準に準拠していません。

You can use a CallableStatement object to pass parameter values to a stored procedure. You can assign values to IN, OUT, and INOUT parameters with a CallableStatement object.

CallableStatement クラスを使用する場合、序数表記または名前表記を使用して、実際の引数の値を指定できます。ステートメント内の各 IN または INOUT パラメーターマーカーの値を設定する必要があります。

序数表記を使用して CallableStatement オブジェクトに値を渡す場合、パラメーターの型に対応するセッターメソッドを使用する必要があります。たとえば、STRING 値を渡すときは、setString セッターメソッドを使用します。ステートメント（？）内の各パラメーターマーカーは順序値を表します。順序パラメータを使用する場合は、プロシージャ定義内で仮引数が指定されている順序で、実際のパラメータ値をステートメントに渡す必要があります。

CallableStatement オブジェクトの引数値を指定するときに名前付きパラメーター表記を使用することもできます。名前付きパラメーター表記法を使用すると、プロシージャに必要なパラメーターのみに値を指定でき、受け入れ可能なデフォルト値を持つパラメーターは省略できます。名前付きパラメーターを任意の順序で指定することもできます。

名前付き表記を使用する場合、各パラメーター名は DatabaseMetaData.getProcedureColumns メソッドの呼び出しによって返される COLUMN_NAME に対応する必要があります。ステートメント呼び出しに名前付きパラメーターを含める場合は、=> トークンを使用する必要があります。

registerOutParameter メソッドを使用して、ステートメント内の各 OUT または INOUT パラメーターマーカーを識別します。

例

次の例では、CallableStatement メソッドを使用して、次のシグネチャを持つプロシージャにパラメータを提供します。

```
PROCEDURE hire_emp (ename VARCHAR2
empno NUMBER
job VARCHAR2
sal NUMBER
hiredate DATE DEFAULT now(),
mgr NUMBER DEFAULT 7100,
deptno NUMBER
)
```

次の例では、序数表記を使用してパラメーターを提供しています。

```
CallableStatement cstmt = con.prepareCall
("{CALL hire_emp(?,?,?,?,?,?,?)}");
//Bind a value to each parameter.
cstmt.setString(1, "SMITH");
cstmt.setInt(2, 8888);
cstmt.setString(3, "Sales");
cstmt.setInt(4, 5500);
cstmt.setDate(5, "2016-06-01");
cstmt.setInt(6, 7566);
cstmt.setInt(7, 30);
```

次の例では、名前付き表記を使用してパラメーターを提供しています。名前付き表記を使用すると、デフォルト値を持つパラメーターを省略したり、パラメーターを並べ替えることができます。

```
CallableStatement cstmt = con.prepareCall
("{CALL hire_emp(ename => ?,
job => ?,
empno => ?,
sal => ?,
deptno => ?
)}");

//Bind a value to each parameter.
cstmt.setString("ename", "SMITH");
cstmt.setInt("empno", 8888);
cstmt.setString("job", "Sales");
cstmt.setInt("sal", 5500);
cstmt.setInt("deptno", 30);
```

5.4 ResultSet オブジェクトからの結果の取得

A `ResultSet` object is the primary storage mechanism for the data returned by an SQL statement. Each `ResultSet` object contains both data and metadata (in the form of a `ResultSetMetaData` object). `ResultSetMetaData` includes useful information about results returned by the SQL command: column names, column count, row count, column length, and so on.

To access the row data stored in a `ResultSet` object, an application calls one or more getter methods. A getter method retrieves the value in particular column of the current row. There are many different getter methods; each method returns a value of a particular type. For example, the `getString()` method returns a `STRING` type; the `getDate()` method returns a `Date`, and the `getInt()` method returns an `INT` type. When an application calls a getter method, JDBC tries to convert the value into the requested type.

各 `ResultSet` は、現在の行を指す内部ポインターを保持します。 `executeQuery()` メソッドが `ResultSet` を返すと、ポインターは最初の行の前に配置されます。アプリケーションがポインターを移動する前に getter メソッドを呼び出すと、getter メソッドは失敗します。次の（または最初の）行に進むには、`ResultSet` の `next()` メソッドを呼び出します。 `ResultSet.next()` はブール型メソッドです。 `ResultSet` に別の行がある場合は `TRUE` を返し、最後の行を超えて移動した場合は `FALSE` を返します。

ポインターを最初の行に移動した後、サンプルアプリケーションは `getString()` getter メソッドを使用して最初の列の値を取得し、その値を出力します。 `ListEmployees` はループで `rs.next()` と `rs.getString()` を呼び出すため、結果セットの各行を処理します。 `ListEmployees` は、 `rs.next()` がポインターを最後の行を超えて移動し、 `FALSE` を返すとループを終了します。

```
while(rs.next())
{
    System.out.println(rs.getString(1));
}
```

`ResultSet` インターフェースを使用するときは、次のことに注意してください:

- 値を読み込む前に `next()` を呼び出す必要があります。別の行が利用可能な場合、`next()` は `true` を返し、処理のために行を準備します。
- JDBC 仕様では、アプリケーションは `ResultSet` の各行に 1 回だけアクセスする必要があります。現時点では、`AdvancedServerJDBC` ドライバーを使用すると、必要な回数だけフィールドにアクセスできますが、この規則を守るのが最も安全です。
- `ResultSet` の使用が終了したら、`close()` メソッドを呼び出して、そのオブジェクトが保持しているリソースを解放します。

5.5 リソースの解放

すべての JDBC オブジェクトは、いくつかのリソースを消費します。たとえば、ResultSet オブジェクトには、クエリによって返されるすべての行のコピーが含まれる場合があります。Statement オブジェクトには最後に実行されたコマンドのテキストなどが含まれます。通常、アプリケーションでリソースが不要になったら、それらのリソースを解放することをお勧めします。サンプルアプリケーションは、各オブジェクトの `close()` メソッドを呼び出すことにより、Result、Statement、および Connection オブジェクトによって消費されるリソースを解放します。

```
rs.close();  
stmt.close();  
con.close();
```

JDBC オブジェクトを閉じた後に使用しようとする、そのオブジェクトはエラーをスローします。

5.6 エラーの処理

外部リソース（データベースサーバーなど）に接続すると、エラーが発生します。コードには、これらのエラーを処理する方法を含める必要があります。JDBC と AdvancedServerJDBC コネクターの両方は、さまざまなタイプのエラー処理を提供します。ListEmployees クラスの例<using_the_advanced_server_jdbc_connector_with_java_applications>は、`try/catch` ブロックを使用してエラーを処理する方法を示しています。

When a JDBC object throws an error (an object of type `SQLException` or of a type derived from `SQLException`), the `SQLException` object exposes three different pieces of error information:

- エラーメッセージ。
- SQL の状態。
- ベンダー固有のエラーコード。

この例では、次のコードは、エラーが発生した場合にこれらのコンポーネントの値を表示します。

```
System.out.println("SQL Exception: " + exp.getMessage());
System.out.println("SQL State: " + exp.getSQLState());
System.out.println("Vendor Error: " + exp.getErrorCode());
```

たとえば、サーバーが指定されたホストに存在しないデータベースに接続しようとする、次のエラーメッセージが表示されます。

```
SQL Exception: FATAL: database "acctg" does not exist
SQL State: 3D000
Vendor Error: 0
```

第 6

executeUpdate () による SQL コマンドの実行

前の例では、ListEmployees は Statement.executeQuery() メソッドを使用して SELECT ステートメントを実行しました。executeQuery() はクエリステートメントを実行するように設計されているため、クエリによって返されたデータを含む ResultSet を返します。Statement クラスは、他の種類のコマンド (UPDATE、INSERT、DELETE など) を実行するために使用する 2 番目のメソッドを提供します。行のコレクションを返す代わりに、executeUpdate() メソッドは実行する SQL コマンドの影響を受ける行の数を返します。

executeUpdate() メソッドのシグネチャは次のとおりです。

```
int executeUpdate(String sqlStatement)
```

このメソッドに、実行したい SQL コマンドを含む String 型の単一のパラメーターを提供します。

6.1 executeUpdate () を使用してデータを挿入する

次の例は、executeUpdate () メソッドを使用して emp テーブルに行を追加する方法を示しています。

注: 次の例は完全なアプリケーションではなく、メソッドにすぎません。このドキュメントの残りのサンプルには、Connection の設定と破棄に必要なコードは含まれていません。この例を試すには、サンプルコードを呼び出すクラスを提供する必要があります。

リスト 1.3

```
public void updateEmployee(Connection con)
{
    try
    {
        Console console = System.console();
        Statement stmt = con.createStatement();

        String empno = console.readLine("Employee Number :");
        String ename = console.readLine("Employee Name :");
        int rowcount = stmt.executeUpdate("INSERT INTO emp(empno, ename)
            VALUES ('"+empno+"', '"+ename+"')");
        System.out.println("");
        System.out.println("Success - "+rowcount+" rows affected.");
    }
    catch(Exception err)
    {
        System.out.println("An error has occurred.");
        System.out.println("See full details below.");
        err.printStackTrace();
    }
}
```

updateEmployee () メソッドは、呼び出し元からの単一の引数、AdvancedServer データベースに接続する必要がある Connection オブジェクトを必要とします。

```
public void updateEmployee(Connection con)
```

次に、updateEmployee () はユーザーに従業員の名前と番号の入力を求めます:

```
String empno = console.readLine("Employee Number :");
String ename = console.readLine("Employee Name :");
```

updateEmployee () は console.readLine () によって返された値を INSERT ステートメントに連結し、結果を executeUpdate () メソッドに渡します。

```
int rowcount = stmt.executeUpdate("INSERT INTO emp(empno, ename)
VALUES ('"+empno+"', '"+ename+"')");
```

たとえば、ユーザーが 6000 という従業員番号と Jones という名前を入力した場合、`executeUpdate()` に渡される INSERT ステートメントは次のようになります。

```
INSERT INTO emp(empno, ename) VALUES(6000, 'Jones');
```

`executeUpdate()` メソッドは、SQL ステートメントの影響を受ける行の数を返します（通常、INSERT は 1 つの行に影響を与えますが、UPDATE または DELETE ステートメントはさらに影響を与えます）。エラーをスローせずに `executeUpdate()` が返された場合、`System.out.println` を呼び出すと、影響を受けた行の数を示すメッセージがユーザーに表示されます。

```
System.out.println("");  
System.out.println("Success - "+rowcount+" rows affected.");
```

プログラムが例外を検出した場合、`catch` ブロックは適切なエラーメッセージをユーザーに表示します。

```
{  
System.out.println("An error has occurred.");  
System.out.println("See full details below.");  
err.printStackTrace();  
}
```

6.1.1 executeUpdate () の構文例

You can use `executeUpdate ()` with any SQL command that does not return a result set. Some simple syntax examples using `executeUpdate ()` with SQL commands follow:

`executeUpdate ()` で UPDATE コマンドを使用して行を更新するには:

```
stmt.executeUpdate("UPDATE emp SET ename='"+ename+"' WHERE empno="+empno);
```

DELETE コマンドと `executeUpdate ()` を使用してテーブルから行を削除するには:

```
stmt.executeUpdate("DELETE FROM emp WHERE empno="+empno);
```

`executeUpdate ()` で DROPTABLE コマンドを使用して、データベースからテーブルを削除するには:

```
stmt.executeUpdate("DROP TABLE tablename");
```

CREATE TABLE コマンドと `executeUpdate ()` を使用して、新しいテーブルをデータベースに追加するには:

```
stmt.executeUpdate("CREATE TABLE tablename (fieldname NUMBER(4,2),  
fieldname2 VARCHAR2(30))");
```

テーブルの属性を変更するために ALTER TABLE コマンドと `executeUpdate ()` を使用するには:

```
stmt.executeUpdate("ALTER TABLE tablename ADD COLUMN colname BOOLEAN");
```

第 7

グラフィカルインターフェイスを Java プログラムに追加する

With a little extra work, you can add a graphical user interface to a program - the next example (Listing 1.4) demonstrates how to write a Java application that creates a `JTable` (a spreadsheet-like graphical object) and copies the data returned by a query into that `JTable`.

注釈: 次のサンプルアプリケーションはメソッドであり、完全なアプリケーションではありません。このメソッドを呼び出すには、適切な `main ()` 関数とラッパークラスを提供します。

リスト 1.4

```
import java.sql.*;
import java.util.Vector;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTable;

...
public void showEmployees(Connection con)
{
    try
    {
        Statement stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery("SELECT * FROM emp");
        ResultSetMetaData rsmd = rs.getMetaData();
        Vector labels = new Vector();
```

(次のページに続く)

(前のページからの続き)

```
for(int column = 0; column < rsmd.getColumnCount(); column++)
    labels.addElement(rsmd.getColumnLabel(column + 1));

Vector rows = new Vector();
while(rs.next())
{
    Vector rowValues = new Vector();
    for(int column = 0; column < rsmd.getColumnCount(); column++)
        rowValues.addElement(rs.getString(column + 1));
    rows.addElement(rowValues);
}

JTable table = new JTable(rows, labels);
JFrame jf = new JFrame("Browsing table: EMP (from EnterpriseDB)");
jf.getContentPane().add(new JScrollPane(table));
jf.setSize(400, 400);
jf.setVisible(true);
jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
System.out.println("Command successfully executed");
}
catch(Exception err)
{
    System.out.println("An error has occurred.");
    System.out.println("See full details below.");
    err.printStackTrace();
}
}
```

showEmployees() メソッドを記述する前に、いくつかの **JDK** 提供クラスの定義をインポートする必要があります。

```
import java.sql.*;
import java.util.Vector;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTable;
```

showEmployees() メソッドは、呼び出し元から **Connection** オブジェクトが提供されることを想定しています。**Connection** オブジェクトは **AdvancedServer** に接続する必要があります:

```
public void showEmployees(Connection con)
```

showEmployees() は **Statement** を作成し、executeQuery() メソッドを使用して従業員リストを生成する **SQL** クエリを実行します。

```
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM emp");
```

ご想像のとおり、`executeQuery()` は `ResultSet` オブジェクトを返します。`ResultSet` オブジェクトには、結果セットの shape を記述するメタデータが含まれます（つまり、結果セットの行と列の数、各列のデータ型、各列の名前、など）。`getMetaData()` メソッドを呼び出すことで `ResultSet` からメタデータを抽出できます:

```
ResultSetMetaData rsmd = rs.getMetaData();
```

次に、`showEmployees()` は `Vector`（1 次元配列）を作成して列ヘッダーを保持し、次に各ヘッダーを `ResultSetMetaData` オブジェクトからベクターにコピーします。

```
Vector labels = new Vector();
for(int column = 0; column < rsmd.getColumnCount(); column++)
{
    labels.addElement(rsmd.getColumnLabel(column + 1));
}
```

With the column headers in place, `showEmployees()` extracts each row from the `ResultSet` and copies it into a new vector (named `rows`). The `rows` vector is actually a vector of vectors: each entry in the `rows` vector contains a vector that contains the data values in that row. This combination forms the two-dimensional array that you will need to build a `JTable`. After creating the `rows` vector, the program reads through each row in the `ResultSet` (by calling `rs.next()`). For each column in each row, a getter method extracts the value at that row/column and adds the value to the `rowValues` vector. Finally, `showEmployee()` adds each `rowValues` vector to the `rows` vector:

```
Vector rows = new Vector();
while(rs.next())
{
    Vector rowValues = new Vector();
    for(int column = 0; column < rsmd.getColumnCount(); column++)
        rowValues.addElement(rs.getString(column + 1));
    rows.addElement(rowValues);
}
```

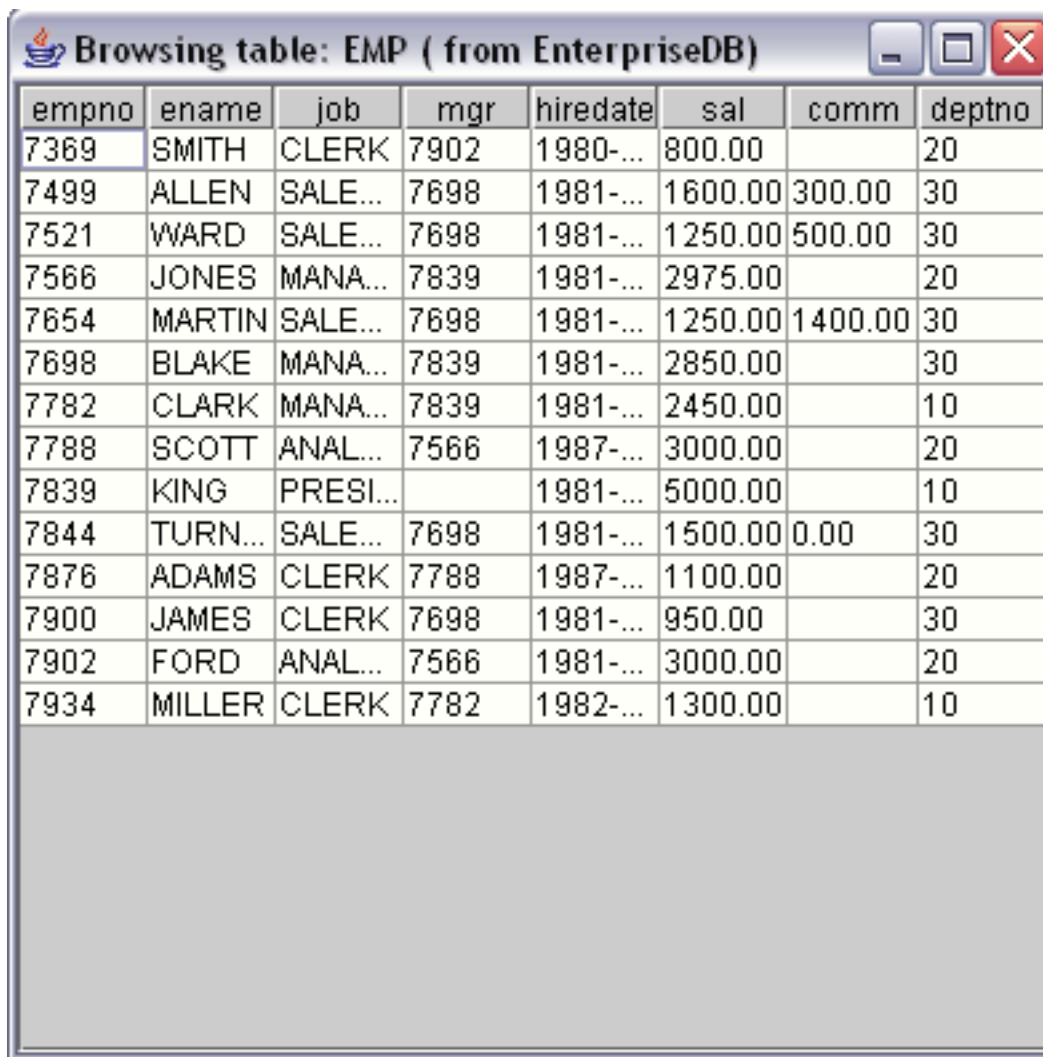
At this point, the vector (`labels`) contains the column headers, and a second two-dimensional vector (`rows`) contains the data for the table. Now you can create a `JTable` from the vectors and a `JFrame` to hold the `JTable`:

```
JTable table = new JTable(rows, labels);
JFrame jf = new JFrame("Browsing table: EMP (from EnterpriseDB)");
jf.getContentPane().add(new JScrollPane(table));
jf.setSize(400, 400);
jf.setVisible(true);
jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
System.out.println("Command successfully executed");
```

`showEmployees()` メソッドには、発生する可能性のあるエラーをインターセプトし、適切なメッセージをユーザーに表示する `catch` ブロックが含まれています。

```
catch(Exception err)
{
    System.out.println("An error has occurred.");
    System.out.println("See full details below.");
    err.printStackTrace();
}
```

`showEmployees()` メソッドを呼び出した結果を次の図に示します。



empno	ename	job	mgr	hiredate	sal	comm	deptno
7369	SMITH	CLERK	7902	1980-...	800.00		20
7499	ALLEN	SALE...	7698	1981-...	1600.00	300.00	30
7521	WARD	SALE...	7698	1981-...	1250.00	500.00	30
7566	JONES	MANA...	7839	1981-...	2975.00		20
7654	MARTIN	SALE...	7698	1981-...	1250.00	1400.00	30
7698	BLAKE	MANA...	7839	1981-...	2850.00		30
7782	CLARK	MANA...	7839	1981-...	2450.00		10
7788	SCOTT	ANAL...	7566	1987-...	3000.00		20
7839	KING	PRESI...		1981-...	5000.00		10
7844	TURN...	SALE...	7698	1981-...	1500.00	0.00	30
7876	ADAMS	CLERK	7788	1987-...	1100.00		20
7900	JAMES	CLERK	7698	1981-...	950.00		30
7902	FORD	ANAL...	7566	1981-...	3000.00		20
7934	MILLER	CLERK	7782	1982-...	1300.00		10

図 1 `showEmployees` ウィンドウ

第 8

高度な JDBC コネクタ機能

前の例では、結果セットを JTable で表示するグラフィカルユーザーインターフェイスを作成しました。次に、ギアを切り替えて、AdvancedServerJDBCCConnector のより高度な機能の一部を紹介します。

不要な混乱を避けるために、このドキュメントの残りのコードサンプルでは、グラフィカルな使用インターフェイスを作成する代わりに、コンソールを使用してユーザーと対話します。

8.1 クライアント側のリソース要件の削減

The Advanced Server JDBC driver retrieves the results of a SQL query as a `ResultSet` object. If a query returns a large number of rows, using a `batched ResultSet` will:

- 最初の行を取得するのにかかる時間を短縮します。
- 必要な行のみを取得して時間を節約します。
- クライアントのメモリ要件を減らします。

When you reduce the fetch size of a `ResultSet` object, the driver doesn't copy the entire `ResultSet` across the network (from the server to the client). Instead, the driver requests a small number of rows at a time; as the client application moves through the result set, the driver fetches the next batch of rows from the server.

バッチ結果セットは、すべての状況で利用できるわけではありません。以下の制限を守らないと、ドライバーは静かに `ResultSet` 全体を一度にフェッチします:

- クライアントアプリケーションは `autocommit` を無効にする必要があります。
- The `Statement` object must be created with a `ResultSet` type of `TYPE_FORWARD_ONLY` type (which is the default). `TYPE_FORWARD_ONLY` result sets can only step forward through the `ResultSet`.
- クエリは、単一の SQL ステートメントで構成する必要があります。

8.1.1 ステートメントオブジェクトのバッチサイズの変更

ResultSet オブジェクトのバッチサイズを制限すると、データの取得が高速化され、クライアント側アプリケーションに必要なリソースが削減されます。リスト 1.5 では、バッチサイズが 5 行に制限された Statement オブジェクトを作成します。

```
// Make sure autocommit is off
conn.setAutoCommit(false);

Statement stmt = conn.createStatement();
// Set the Batch Size.
stmt.setFetchSize(5);

ResultSet rs = stmt.executeQuery("SELECT * FROM emp");
while (rs.next())
    System.out.println("a row was returned.");

rs.close();
stmt.close();
```

conn.setAutoCommit(false) を呼び出すと、最初の行を取得する前にサーバーが ResultSet を閉じないことが保証されます。Connection を準備した後、Statement オブジェクトを構築できます：

```
Statement stmt = db.createStatement();
```

次のコードは、クエリを実行する前にバッチサイズを 5（行）に設定します。

```
stmt.setFetchSize(5);

ResultSet rs = stmt.executeQuery("SELECT * FROM emp");
```

ResultSet オブジェクトの各行に対して、println() の呼び出しは a row was returned を出力します。

```
System.out.println("a row was returned.");
```

ResultSet にはテーブル内のすべての行が含まれていますが、サーバーから一度に 5 行しかフェッチされないことに注意してください。クライアントの観点から見ると、batched の結果セットと unbatched の結果セットの唯一の違いは、バッチ処理された結果がより短い時間で最初の行を返す可能性があることです。

次に、特定の JDBC アプリケーションのパフォーマンスを向上させるために使用できる別の機能（the PreparedStatement）を見ていきます。

8.2 PreparedStatement を使用して SQL コマンドを送信する

多くのアプリケーションは同じ SQL ステートメントを繰り返し実行し、各反復の間にステートメント内の 1 つ以上のデータ値を変更します。Statement オブジェクトを使用して SQL ステートメントを繰り返し実行する場合、サーバーはステートメントを毎回解析、計画、および最適化する必要があります。JDBC は、このようなシナリオで必要な作業量を削減するために、もう 1 つの Statement 派生物、PreparedStatement を提供します。

リスト 1.6 は、従業員 ID と従業員名を受け入れ、その従業員情報を emp テーブルに挿入する PreparedStatement の呼び出しを示しています。

リスト 1.6

```
public void AddEmployee(Connection con)
{
    try
    {
        Console c = System.console();
        String command = "INSERT INTO emp(empno,ename) VALUES(?,?) ";
        PreparedStatement stmt = con.prepareStatement(command);
        stmt.setObject(1,new Integer(c.readLine("ID:")));
        stmt.setObject(2,c.readLine("Name:"));
        stmt.execute();

        System.out.println("The procedure successfully executed.");
    }
    catch(Exception err)
    {
        System.out.println("An error has occurred.");
        System.out.println("See full details below.");
        err.printStackTrace();
    }
}
```

SQL ステートメントにデータ値をハードコーディングする代わりに、placeholders を挿入して、各反復で変化する値を表します。リスト 1.6 は、2 つのプレースホルダー（それぞれが疑問符で表される）を含む INSERT ステートメントを示しています。

```
String command = "INSERT INTO emp(empno,ename) VALUES(?,?) ";
```

パラメータ化された SQL ステートメントを手にすると、AddEmployee() メソッドは Connection オブジェクトにそのステートメントを準備して PreparedStatement オブジェクトを返すように要求できます:

```
PreparedStatement stmt = con.prepareStatement(command);
```

この時点で、PreparedStatement は INSERT ステートメントを解析および計画しましたが、テーブルに追加する値はわかりません。PreparedStatement を実行する前に、setter メソッドを呼び出して各プレースホ

ルダの値を指定する必要があります。 `setObject()` には 2 つの引数が必要です。

- パラメータ番号。パラメータ番号 1 は最初の疑問符に対応し、パラメータ番号 2 は 2 番目の疑問符に対応します。
- プレースホルダーを置き換える値。

`AddEmployee()` メソッドは、ユーザーに従業員 ID と名前を入力を求め、ユーザーが指定した値で `setObject()` を呼び出します。

```
stmt.setObject(1, new Integer(c.readLine("ID:")));  
stmt.setObject(2, c.readLine("Name:"));
```

そして `PreparedStatement` オブジェクトにステートメントを実行するように要求します:

```
stmt.execute();
```

SQL ステートメントが期待どおりに実行される場合、`AddEmployee()` は実行を確認するメッセージを表示します。サーバーで例外が発生すると、エラー処理コードにエラーメッセージが表示されます。

8.3 ストアドプロシージャの実行

ストアドプロシージャは、EnterpriseDB の SPL で記述され、データベースに格納されるモジュールです。ストアドプロシージャは、プロシージャにデータを提供する入力パラメータと、プロシージャからデータを返す出力パラメータを定義できます。ストアドプロシージャはサーバー内で実行され、データベースアクセスコマンド (SQL)、制御ステートメント、およびデータベースから取得したデータを操作するデータ構造で構成されます。

ストアドプロシージャは、クライアントからのデータを保存する前に広範なデータ操作が必要な場合に特に役立ちます。ストアドプロシージャを使用してバッチプログラムでデータを操作することも効率的です。

8.3.1 ストアドプロシージャの呼び出し

CallableStatement クラスは、Java プログラムがストアドプロシージャを呼び出す方法を提供します。CallableStatement オブジェクトは、入力（IN パラメーター）、出力（OUT パラメーター）、またはその両方（IN OUT パラメーター）に使用される可変数のパラメーターを持つことができます。

JDBC でストアドプロシージャを呼び出すための構文を以下に示します。角括弧はオプションのパラメータを示していることに注意してください。それらはコマンド構文の一部ではありません。

```
{call procedure_name([?, ?, ...])}
```

結果パラメーターを返すプロシージャを呼び出す構文は次のとおりです。

```
{? = call procedure_name([?, ?, ...])}
```

Each question mark serves as a placeholder for a parameter. The stored procedure determines if the placeholders represent IN, OUT, or IN OUT parameters and the Java code must match. We will show you how to supply values for IN (or IN OUT) parameters and how to retrieve values returned in OUT (or IN OUT) parameters in a moment.

単純なストアードプロシージャの実行

リスト 1.7-a は、各従業員の給与を 10% 増やすストアードプロシージャを示しています。increaseSalary は呼び出し元からの引数を期待せず、情報を返しません:

```
CREATE OR REPLACE PROCEDURE increaseSalary
IS
BEGIN
    UPDATE emp SET sal = sal * 1.10;
END;
```

リスト 1.7-b は increaseSalary プロシージャを呼び出す方法を示しています。

```
public void SimpleCallSample(Connection con)
{
    try
    {
        CallableStatement stmt = con.prepareCall("{call increaseSalary()}");
        stmt.execute();
        System.out.println("Stored Procedure executed successfully");
    }
    catch(Exception err)
    {
        System.out.println("An error has occurred.");
        System.out.println("See full details below.");
        err.printStackTrace();
    }
}
```

Java アプリケーションからストアードプロシージャを呼び出すには、CallableStatement オブジェクトを使用します。CallableStatement クラスは Statement クラスから派生し、Statement クラスと同様に、Connection オブジェクトに 1 つ作成してもらうことで CallableStatement オブジェクトを取得しますあなた。Connection から CallableStatement を作成するには、prepareCall() メソッドを使用します:

```
CallableStatement stmt = con.prepareCall("{call increaseSalary()}");
```

As the name implies, the prepareCall() method prepares the statement, but does not execute it. As you will see in the next example, an application typically binds parameter values between the call to prepareCall() and the call to execute(). To invoke the stored procedure on the server, call the execute() method.

```
stmt.execute();
```

このストアードプロシージャ (increaseSalary) は IN パラメーターを予期せず、呼び出し側に情報を返さなかった (OUT パラメーターを使用) ため、プロシージャの呼び出しは単に CallableStatement オブジェクトを呼び出し、そのオブジェクトの execute() メソッドを呼び出します。

次のセクションでは、呼び出し元からのデータ (IN パラメーター) を必要とするストアードプロシージャを呼び出

す方法を示します。

IN パラメータを使用したストアードプロシージャの実行

次の例のコードは、最初に empInsert という名前のストアードプロシージャを作成して呼び出します。empInsert には、従業員情報を含む IN パラメーターが必要です: empno、ename、job、sal、comm、deptno、および mgr。empInsert はその情報を emp テーブルに挿入します。

リスト 1.8-a は、AdvancedServer データベースにストアードプロシージャを作成します。

```
CREATE OR REPLACE PROCEDURE empInsert (
    pEname  IN VARCHAR,
    pJob     IN VARCHAR,
    pSal     IN FLOAT4,
    pComm    IN FLOAT4,
    pDeptno  IN INTEGER,
    pMgr     IN INTEGER
)
AS
DECLARE
    CURSOR getMax IS SELECT MAX(empno) FROM emp;
    max_empno INTEGER := 10;
BEGIN
    OPEN getMax;
    FETCH getMax INTO max_empno;
    INSERT INTO emp(empno, ename, job, sal, comm, deptno, mgr)
        VALUES(max_empno+1, pEname, pJob, pSal, pComm, pDeptno, pMgr);
    CLOSE getMax;
END;
```

リスト 1.8-b は、Java からストアードプロシージャを呼び出す方法を示しています。

```
public void CallExample2(Connection con)
{
    try
    {
        Console c = System.console();
        String commandText = "{call empInsert(?,?,?,?,?,?)}";
        CallableStatement stmt = con.prepareCall(commandText);
        stmt.setObject(1, new String(c.readLine("Employee Name :")));
        stmt.setObject(2, new String(c.readLine("Job :")));
        stmt.setObject(3, new Float(c.readLine("Salary :")));
        stmt.setObject(4, new Float(c.readLine("Commission :")));
        stmt.setObject(5, new Integer(c.readLine("Department No :")));
        stmt.setObject(6, new Integer(c.readLine("Manager")));
        stmt.execute();
    }
    catch(Exception err)
    {
        System.out.println("An error has occurred.");
        System.out.println("See full details below.");
    }
}
```

(次のページに続く)

(前のページからの続き)

```
err.printStackTrace();  
}  
}
```

コマンド (`commandText`) 内の各プレースホルダー (?) は、後でデータに置き換えられるコマンド内のポイントを表します。

```
String commandText = "{call EMP_INSERT(?,?,?, ?, ?, ?)}";  
CallableStatement stmt = con.prepareCall(commandText);
```

The `setObject()` method binds a value to an IN or IN OUT placeholder. Each call to `setObject()` specifies a parameter number and a value to bind to that parameter:

```
stmt.setObject(1, new String(c.readLine("Employee Name :")));  
stmt.setObject(2, new String(c.readLine("Job :")));  
stmt.setObject(3, new Float(c.readLine("Salary :")));  
stmt.setObject(4, new Float(c.readLine("Commission :")));  
stmt.setObject(5, new Integer(c.readLine("Department No :")));  
stmt.setObject(6, new Integer(c.readLine("Manager")));
```

各プレースホルダーに値を指定した後、このメソッドは `execute()` メソッドを呼び出してステートメントを実行します。

OUT パラメータを使用したストアードプロシージャの実行

The next example creates and invokes an SPL stored procedure called `deptSelect`. This procedure requires one IN parameter (department number) and returns two OUT parameters (the department name and location) corresponding to the department number. The code in Listing 1.9-a creates the `deptSelect` procedure:

```
CREATE OR REPLACE PROCEDURE deptSelect
(
  p_deptno IN INTEGER,
  p_dname  OUT VARCHAR,
  p_loc    OUT VARCHAR
)
AS
DECLARE
  deptCursor IS SELECT dname, loc FROM dept WHERE deptno=p_deptno;
BEGIN
  OPEN deptCursor;
  FETCH deptCursor INTO p_dname, p_loc;

  CLOSE deptCursor;
END;
```

リスト 1.9-b は、`deptSelect` ストアドプロシージャを呼び出すために必要な Java コードを示しています。

```
public void GetDeptInfo(Connection con)
{
  try
  {
    Console c = System.console();
    String commandText = "{call deptSelect(?,?,?)}";
    CallableStatement stmt = con.prepareCall(commandText);
    stmt.setObject(1, new Integer(c.readLine("Dept No :")));
    stmt.registerOutParameter(2, Types.VARCHAR);
    stmt.registerOutParameter(3, Types.VARCHAR);
    stmt.execute();
    System.out.println("Dept Name: " + stmt.getString(2));
    System.out.println("Location : " + stmt.getString(3));
  }
  catch(Exception err)
  {
    System.out.println("An error has occurred.");
    System.out.println("See full details below.");
    err.printStackTrace();
  }
}
```

コマンド (`commandText`) 内の各ブレースホルダー (?) は、後でデータに置き換えられるコマンド内のポイントを表します。

```
String commandText = "{call deptSelect(?,?,?)}";
CallableStatement stmt = con.prepareCall(commandText);
```

The `setObject()` method binds a value to an IN or ``IN OUT placeholder. When calling `setObject()` you must identify a placeholder (by its ordinal number) and provide a value to substitute in place of that placeholder:

```
stmt.setObject(1, new Integer(c.readLine("Dept No :")));
```

各 OUT パラメーターの JDBC タイプは、`CallableStatement` オブジェクトを実行する前に登録する必要があります。JDBC タイプの登録は `registerOutParameter()` メソッドで行われます。

```
stmt.registerOutParameter(2, Types.VARCHAR);
stmt.registerOutParameter(3, Types.VARCHAR);
```

ステートメントを実行した後、`CallableStatement` のゲッターメソッドは OUT パラメーター値を取得します:VARCHAR 値を取得するには、`getString()` ゲッターメソッドを使用します。

```
stmt.execute();
System.out.println("Dept Name: " + stmt.getString(2));
System.out.println("Location : " + stmt.getString(3));
```

In the current example `GetDeptInfo()` registers two OUT parameters and (after executing the stored procedure) retrieves the values returned in the OUT parameters. Since both OUT parameters are defined as VARCHAR values, `GetDeptInfo()` uses the `getString()` method to retrieve the OUT parameters.

INOUT パラメータを使用したストアードプロシージャの実行

The code in the next example creates and invokes a stored procedure named `empQuery` defined with one IN parameter (`p_deptno`), two IN OUT parameters (`p_empno` and `p_ename`) and three OUT parameters (`p_job`, `p_hiredate` and `p_sal`). `empQuery` then returns information about the employee in the two IN OUT parameters and three OUT parameters.

リスト 1.10-a は `empQuery` という名前のストアードプロシージャを作成します。

```
CREATE OR REPLACE PROCEDURE empQuery
(
    p_deptno      IN      NUMBER,
    p_empno       IN OUT  NUMBER,
    p_ename       IN OUT  VARCHAR2,
    p_job         OUT    VARCHAR2,
    p_hiredate    OUT    DATE,
    p_sal         OUT    NUMBER
)
IS
BEGIN
    SELECT empno, ename, job, hiredate, sal
        INTO p_empno, p_ename, p_job, p_hiredate, p_sal
        FROM emp
        WHERE deptno = p_deptno
            AND (empno = p_empno
                OR ename = UPPER(p_ename));
END;
```

リスト 1.10-b は、`empQuery` プロシージャの呼び出し、IN パラメーターの値の提供、および OUT パラメーターと IN OUT パラメーターの処理を示しています。

```
public void CallSample4(Connection con)
{
    try
    {
        Console c = System.console();
        String commandText = "{call emp_query(?,?,?, ?, ?, ?)}";
        CallableStatement stmt = con.prepareCall(commandText);
        stmt.setInt(1, new Integer(c.readLine("Department No:")));
        stmt.setInt(2, new Integer(c.readLine("Employee No:")));
        stmt.setString(3, new String(c.readLine("Employee Name:")));
        stmt.registerOutParameter(2, Types.INTEGER);
        stmt.registerOutParameter(3, Types.VARCHAR);
        stmt.registerOutParameter(4, Types.VARCHAR);
        stmt.registerOutParameter(5, Types.TIMESTAMP);
        stmt.registerOutParameter(6, Types.NUMERIC);
        stmt.execute();
        System.out.println("Employee No: " + stmt.getInt(2));
    }
    catch (SQLException e)
    {
        e.printStackTrace();
    }
}
```

(次のページに続く)

(前のページからの続き)

```

        System.out.println("Employee Name: " + stmt.getString(3));
        System.out.println("Job : " + stmt.getString(4));
        System.out.println("Hiredate : " + stmt.getTimestamp(5));
        System.out.println("Salary : " + stmt.getBigDecimal(6));
    }
    catch(Exception err)
    {
        System.out.println("An error has occurred.");
        System.out.println("See full details below.");
        err.printStackTrace();
    }
}

```

コマンド (`commandText`) 内の各ブレースホルダー (`?`) は、後でデータに置き換えられるコマンド内のポイントを表します。

```

String commandText = "{call emp_query(?,?,?, ?, ?, ?)}";
CallableStatement stmt = con.prepareCall(commandText);

```

The `setInt()` method is a type-specific setter method that binds an `Integer` value to an IN or IN OUT placeholder. The call to `setInt()` specifies a parameter number and provides a value to substitute in place of that placeholder:

```

stmt.setInt(1, new Integer(c.readLine("Department No:")));
stmt.setInt(2, new Integer(c.readLine("Employee No:")));

```

`setString()` メソッドは、`String` 値を IN または IN OUT ブレースホルダーにバインドします:

```

stmt.setString(3, new String(c.readLine("Employee Name:")));

```

`CallableStatement` を実行する前に、`registerOutParameter()` メソッドを呼び出して、各 OUT パラメーターの JDBC タイプを登録する必要があります。

```

stmt.registerOutParameter(2, Types.INTEGER);
stmt.registerOutParameter(3, Types.VARCHAR);
stmt.registerOutParameter(4, Types.VARCHAR);
stmt.registerOutParameter(5, Types.TIMESTAMP);
stmt.registerOutParameter(6, Types.NUMERIC);

```

IN パラメータでプロシージャを呼び出す前に、`setter` メソッドでそのパラメータに値を割り当てる必要があることを忘れないでください。OUT パラメータを使用してプロシージャを呼び出す前に、そのパラメータのタイプを登録します。次に、`getter` メソッドを呼び出して返された値を取得できます。IN OUT パラメータを定義するプロシージャを呼び出す場合、3つのアクションすべてを実行する必要があります。

- パラメーターに値を割り当てます。

- パラメーターのタイプを登録します。
- `getter` メソッドで返された値を取得します。

8.4 Java で REF カーソルを使用する

A REF CURSOR is a cursor variable that contains a pointer to a query result set returned by an OPEN statement. Unlike a static cursor, a REF CURSOR is not tied to a particular query. You may open the same REF CURSOR variable any number of times with the OPEN statement containing different queries; each time, a new result set is created for that query and made available via the cursor variable. A REF CURSOR can also pass a result set from one procedure to another.

AdvancedServer は、strongly-typed と weakly-typed REF CURSORS の両方の宣言をサポートしています。強く型付けされたカーソルは、予想される結果セットの shape（各列の型）を宣言する必要があります。厳密に型指定されたカーソルは、宣言された列を返すクエリでのみ使用できます。異なる形状の結果セットを返すクエリでカーソルを開くと、サーバーは例外をスローします。一方、弱い型指定のカーソルは、任意の形状の結果セットで機能します。

強く型付けされた REF CURSOR を宣言するには:

```
TYPE <cursor_type_name> IS REF CURSOR RETURN <return_type>;
```

弱い型付けの REF CURSOR を宣言するには:

```
name SYS_REFCURSOR;

.. raw:: latex

    \newpage
```

8.4.1 REFCURSOR を使用して ResultSet を取得する

The stored procedure shown in Listing 1.11-a (getEmpNames) builds two REF CURSORS on the server; the first REF CURSOR contains a list of commissioned employees in the emp table, while the second REF CURSOR contains a list of salaried employees in the emp table:

リスト 1.11-a

```
CREATE OR REPLACE PROCEDURE getEmpNames
(
  commissioned IN OUT SYS_REFCURSOR,
  salaried IN OUT SYS_REFCURSOR
)
IS
BEGIN
  OPEN commissioned FOR SELECT ename FROM emp WHERE comm is NOT NULL;
  OPEN salaried FOR SELECT ename FROM emp WHERE comm is NULL;
END;
```

RefCursorSample() メソッド (リスト 1.11-b を参照) は getEmpName() ストアドプロシージャを呼び出し、2 つの REF CURSOR 変数のそれぞれに返された名前を表示します。

リスト 1.11-b

```
public void RefCursorSample(Connection con)
{
    try
    {
        con.setAutoCommit(false);
        String commandText = "{call getEmpNames(?,?)}";
        CallableStatement stmt = con.prepareCall(commandText);
        stmt.setNull(1, Types.REF);
        stmt.registerOutParameter(1, Types.REF);
        stmt.setNull(2, Types.REF);
        stmt.registerOutParameter(2, Types.REF);

        stmt.execute();
        ResultSet commissioned = (ResultSet)stmt.getObject(1);
        System.out.println("Commissioned employees:");
        while(commissioned.next())
        {
            System.out.println(commissioned.getString(1));
        }

        ResultSet salaried = (ResultSet)stmt.getObject(2);
        System.out.println("Salaried employees:");
        while(salaried.next())
        {
            System.out.println(salaried.getString(1));
        }
    }
    catch(Exception err)
    {
        System.out.println("An error has occurred.");
        System.out.println("See full details below.");
        err.printStackTrace();
    }
}
```

CallableStatement は各 REF CURSOR (commissioned と salaried) を準備します。各カーソルは、ストアドプロシージャ getEmpNames() の IN OUT パラメーターとして返されます。

```
String commandText = "{call getEmpNames(?,?)}";
CallableStatement stmt = con.prepareCall(commandText);
```

registerOutParameter() への呼び出しは、最初の REFCURSOR (委託) のパラメータータイプ (Types.REF) を登録します。

```
stmt.setNull(1, Types.REF);  
stmt.registerOutParameter(1, Types.REF);
```

registerOutParameter() への別の呼び出しは、2 番目の REF CURSOR (salaried) の 2 番目のパラメーター型 (Types.REF) を登録します:

```
stmt.setNull(2, Types.REF);  
stmt.registerOutParameter(2, Types.REF);
```

stmt.execute() への呼び出しはステートメントを実行します:

```
stmt.execute();
```

getObject() メソッドは、最初のパラメーターから値を取得し、結果を ResultSet にキャストします。次に、RefCursorSample はカーソルを反復処理し、各委託従業員の名前を出力します。

```
ResultSet commissioned = (ResultSet)stmt.getObject(1);  
while(commissioned.next())  
{  
    System.out.println(commissioned.getString(1));  
}
```

同じゲッターメソッドは、2 番目のパラメーターから ResultSet を取得し、そのカーソルを介して RefCursorExample を繰り返し、給与を受け取る各従業員の名前を出力します。

```
ResultSet salaried = (ResultSet)stmt.getObject(2);  
while(salaried.next())  
{  
    System.out.println(salaried.getString(1));  
}
```

8.5 Java で BYTEA データを使用する

The BYTEA data type stores a binary string in a sequence of bytes; digital images and sound files are often stored as binary data. Advanced Server can store and retrieve binary data via the BYTEA data type.

次の Java サンプルは、BYTEA データを AdvancedServer データベースに保存し、そのデータを取得する方法を示しています。この例では少しセットアップが必要です。リスト 1.12-a、1.12-b、および 1.12-c は、Java の例のサーバー側環境を作成します。

Listing 1.12-a creates a table (emp_detail) that stores BYTEA data. emp_detail contains two columns: the first column stores an employee's ID number (type INT) and serves as the primary key for the table; the second column stores a photograph of the employee in BYTEA format:

```
CREATE TABLE emp_detail
(
  empno INT4 PRIMARY KEY,
  pic   BYTEA
);
```

リスト 1.12-b は、emp_detail テーブルに行を挿入するプロシージャ (ADD_PIC) を作成します。

```
CREATE OR REPLACE PROCEDURE ADD_PIC(p_empno IN int4, p_photo IN bytea) AS
BEGIN
  INSERT INTO emp_detail VALUES(p_empno, p_photo);
END;
```

最後に、リスト 1.12-c は、特定の従業員の写真を返す関数 (GET_PIC) を作成します。

```
CREATE OR REPLACE FUNCTION GET_PIC(p_empno IN int4) RETURN BYTEA IS
DECLARE
  photo BYTEA;
BEGIN
  SELECT pic INTO photo FROM EMP_DETAIL WHERE empno = p_empno;
  RETURN photo;
END;
```

8.5.1 BYTEA データを AdvancedServer データベースに挿入する

リスト 1.13 は ADD_PIC プロシージャ（リスト 1.12-b を参照）を呼び出して、クライアントファイルシステムからサーバー上の emp_detail テーブルに写真をコピーする Java メソッドを示しています。

```
public void InsertPic(Connection con)
{
    try
    {
        Console c = System.console();
        int empno = Integer.parseInt(c.readLine("Employee No :"));
        String fileName = c.readLine("Image filename :");
        File f = new File(fileName);

        if(!f.exists())
        {
            System.out.println("Image file not found. Terminating...");
            return;
        }

        CallableStatement stmt = con.prepareCall("{call ADD_PIC(?, ?)}");
        stmt.setInt(1, empno);
        stmt.setBinaryStream(2, new FileInputStream(f), (int)f.length());
        stmt.execute();
        System.out.println("Added image for Employee "+empno);
    }
    catch(Exception err)
    {
        System.out.println("An error has occurred.");
        System.out.println("See full details below.");
        err.printStackTrace();
    }
}
```

InsertPic() は、従業員番号と画像ファイルの名前をユーザーに要求します:

```
int empno = Integer.parseInt(c.readLine("Employee No :"));
String fileName = c.readLine("Image filename :");
```

要求されたファイルが存在しない場合、InsertPic() はエラーメッセージを表示して終了します:

```
File f = new File(fileName);

if(!f.exists())
{
    System.out.println("Image file not found. Terminating...");
    return;
}
```

次に、`InsertPic()` は `ADD_PIC` プロシージャを呼び出す `CallableStatement` オブジェクト (`stmt`) を準備します。最初のプレースホルダー (?) は、`ADD_PIC (p_empno)` で予期される最初のパラメーターを表します。2 番目のプレースホルダーは 2 番目のパラメーター (`p_photo`) を表します。これらのプレースホルダーに実際の値を提供するために、`InsertPic()` は 2 つのセッターメソッドを呼び出します。最初のパラメーターは `INTEGER` 型であるため、`InsertPic()` は `setInt()` メソッドを呼び出して `p_empno` の値を提供します。2 番目のパラメーターは `BYTEA` 型であるため、`InsertPic()` はバイナリセッターメソッドを使用します。この場合、メソッドは `setBinaryStream()` です:

```
CallableStatement stmt = con.prepareCall("{call ADD_PIC(?, ?)}");
stmt.setInt(1, empno);
stmt.setBinaryStream(2, new FileInputStream(f), f.length());
```

プレースホルダーが実際の値にバインドされたので、`InsertPic()` は `CallableStatement` を実行します:

```
stmt.execute();
```

すべてうまくいけば、`InsertPic()` は画像がテーブルに追加されたことを確認するメッセージを表示します。エラーが発生した場合、`catch` ブロックはユーザーにメッセージを表示します:

```
System.out.println("Added image for Employee \"\"+empno);
catch(Exception err)
{
    System.out.println("An error has occurred.");
    System.out.println("See full details below.");
    err.printStackTrace();
}
```

8.5.2 AdvancedServer データベースから BYTEA データを取得する

Now that you know how to insert BYTEA data from a Java application, Listing 1.14 demonstrates how to retrieve BYTEA data from the server:

```
public static void GetPic(Connection con)
{
    try
    {
        Console c = System.console();
        int empno = Integer.parseInt(c.readLine("Employee No :"));
        CallableStatement stmt = con.prepareCall("{?=call GET_PIC(?)}");
        stmt.setInt(2, empno);
        stmt.registerOutParameter(1, Types.BINARY);
        stmt.execute();
        byte[] b = stmt.getBytes(1);

        String fileName = c.readLine("Destination filename :");
        FileOutputStream fos = new FileOutputStream(new File(fileName));
        fos.write(b);
        fos.close();
        System.out.println("File saved at \""+fileName+"\"");
    }
    catch(Exception err)
    {
        System.out.println("An error has occurred.");
        System.out.println("See full details below.");
        err.printStackTrace();
    }
}
```

GetPic() は、ユーザーに従業員 ID 番号の入力を求めることから始まります。

```
int empno = Integer.parseInt(c.readLine("Employee No :"));
```

次に、GetPic() は 1 つの IN パラメーターと 1 つの OUT パラメーターで CallableStatement を準備します。最初のパラメーターは、データベースから取得した写真を含む OUT パラメーターです。写真は BYTEA データであるため、GetPic() はパラメーターを Type.BINARY として登録します。2 番目のパラメーターは従業員番号 (INT) を保持する IN パラメーターであるため、GetPic() は setInt() メソッドを使用して 2 番目のパラメーターの値を提供します。

```
CallableStatement stmt = con.prepareCall("{?=call GET_PIC(?)}");
stmt.setInt(2, empno);
stmt.registerOutParameter(1, Types.BINARY);
```

次に、GetPic() は getBytes ゲッターメソッドを使用して CallableStatement から BYTEA データを取得します。

```
stmt.execute();  
byte[] b = stmt.getBytes(1);
```

プログラムは、写真を保存するファイルの名前をユーザーに要求します。

```
String fileName = c.readLine("Destination filename :");
```

FileOutputStream オブジェクトは、写真を含むバイナリデータを宛先ファイル名に書き込みます。

```
FileOutputStream fos = new FileOutputStream(new File(fileName));  
fos.write(b);  
fos.close();
```

最後に、GetPic() はファイルが新しい場所に保存されたことを確認するメッセージを表示します:

```
System.out.println("File saved at \""+fileName+"\"");
```

8.6 Java でのオブジェクトタイプとコレクションの使用

SQL の CREATE TYPE コマンドは、AdvancedServer データベースに保存されるユーザー定義の object type を作成するために使用されます。CREATE TYPE コマンドは、一般に array と呼ばれる collection を作成するためにも使用され、これは AdvancedServer データベースにも保存されます。

これらのユーザー定義型は、SPL プロシージャ、SPL 関数、および Java プログラム内で参照できます。

基本的なオブジェクトタイプは、CREATE TYPE AS OBJECT コマンドと CREATE TYPE BODY コマンドのオプションの使用法で作成されます。

nested table type コレクションは CREATE TYPE AS TABLE OF コマンドを使用して作成されます。CREATE TYPE VARRAY コマンドで varray type コレクションが作成されます。

オブジェクトタイプとコレクションの使用例を次のセクションに示します。

リスト 1.15 は、両方の例で AdvancedServer データベースへの接続を確立するために使用される Java メソッドを示しています。

```
public static Connection getEDBConnection() throws
    ClassNotFoundException, SQLException {
    String url = "jdbc:edb://localhost:5444/test";
    String user = "enterprisedb";
    String password = "edb";
    Class.forName("com.edb.Driver");
    Connection conn = DriverManager.getConnection(url, user, password);
    return conn;
}
```

8.6.1 オブジェクトタイプの使用

AdvancedServer データベースにオブジェクトタイプを作成します。オブジェクトタイプ `addr_object_type` はアドレスの属性を定義します:

```
CREATE OR REPLACE TYPE addr_object_type AS OBJECT
(
    street          VARCHAR2(30),
    city            VARCHAR2(20),
    state           CHAR(2),
    zip             NUMBER(5)
);
```

オブジェクトタイプ `emp_obj_typ` は従業員の属性を定義します。これらの属性の 1 つは、前述のオブジェクトタイプ `ADDR_OBJECT_TYPE` であることに注意してください。オブジェクトタイプの本体には、従業員情報を表示するメソッドが含まれています。

```
CREATE OR REPLACE TYPE emp_obj_typ AS OBJECT
(
    empno           NUMBER(4),
    ename           VARCHAR2(20),
    addr            ADDR_OBJECT_TYPE,
    MEMBER PROCEDURE display_emp (SELF IN OUT emp_obj_typ)
);

CREATE OR REPLACE TYPE BODY emp_obj_typ AS
MEMBER PROCEDURE display_emp (SELF IN OUT emp_obj_typ)
IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('Employee No   : ' || SELF.empno);
    DBMS_OUTPUT.PUT_LINE('Name         : ' || SELF.ename);
    DBMS_OUTPUT.PUT_LINE('Street        : ' || SELF.addr.street);
    DBMS_OUTPUT.PUT_LINE('City/State/Zip: ' || SELF.addr.city || ', ' ||
        SELF.addr.state || ' ' || LPAD(SELF.addr.zip,5,'0'));
END;
END;
```

リスト 1.16 は、これらのユーザー定義のオブジェクトタイプを含む Java メソッドです。

```
public static void testUDT() throws SQLException {
    Connection conn = null;
    try {
        conn = getEDBConnection();
        String commandText = "{call emp_obj_typ.display_emp(?)}";
        CallableStatement stmt = conn.prepareCall(commandText);

        // initialize emp_obj_typ structure
```

(次のページに続く)

(前のページからの続き)

```

// create addr_object_type structure
Struct address = conn.createStruct("addr_object_type",
    new Object[]{"123 MAIN STREET", "EDISON", "NJ", 8817});
Struct emp      = conn.createStruct("emp_obj_typ",
    new Object[]{9001, "JONES", address});

// set emp_obj_typ type param
stmt.registerOutParameter(1, Types.STRUCT, "emp_obj_typ");
stmt.setObject(1, emp);
stmt.execute();

// extract emp_obj_typ object
emp = (Struct)stmt.getObject(1);
Object[] attrEmp = emp.getAttributes();
System.out.println("empno: " + attrEmp[0]);
System.out.println("ename: " + attrEmp[1]);

// extract addr_object_type attributes
address = (Struct) attrEmp[2];
Object[] attrAddress = address.getAttributes();
System.out.println("street: " + attrAddress[0]);
System.out.println("city: " + attrAddress[1]);
System.out.println("state: " + attrAddress[2]);
System.out.println("zip: " + attrAddress[3]);
} catch (ClassNotFoundException cnfe) {
    System.err.println("Error: " + cnfe.getMessage());
} finally {
    if (conn != null) {
        conn.close();
    }
}
}

```

CallableStatement オブジェクトは、emp_obj_typ オブジェクトタイプの display_emp() メソッドに基づいて準備されます。

```

String commandText = "{call emp_obj_typ.display_emp(?)}";
CallableStatement stmt = conn.prepareCall(commandText);

```

createStruct() は、それぞれ address および emp という名前のオブジェクト型 addr_object_type および emp_obj_typ のインスタンスを初期化および作成します。

```

Struct address = conn.createStruct("addr_object_type",
    new Object[]{"123 MAIN STREET", "EDISON", "NJ", 8817});
Struct emp      = conn.createStruct("emp_obj_typ",
    new Object[]{9001, "JONES", address});

```

`registerOutParameter()` を呼び出すと、`emp_obj_typ` のパラメータータイプ (`Types.STRUCT`) が登録されます。

```
stmt.registerOutParameter(1, Types.STRUCT, "emp_obj_typ");
```

`setObject()` メソッドは、オブジェクトインスタンス `emp` を IN OUT プレースホルダーにバインドします。

```
stmt.setObject(1, emp);
```

`stmt.execute()` への呼び出しは `display_emp()` メソッドへの呼び出しを実行します:

```
stmt.execute();
```

`getObject()` は `emp_obj_typ` オブジェクトタイプを取得します。次に、`emp` および `address` オブジェクトインスタンスの属性が取得され、表示されます。

```
emp = (Struct) stmt.getObject(1);
Object[] attrEmp = emp.getAttributes();
System.out.println("empno: " + attrEmp[0]);
System.out.println("ename: " + attrEmp[1]);

address = (Struct) attrEmp[2];
Object[] attrAddress = address.getAttributes();
System.out.println("street: " + attrAddress[0]);
System.out.println("city: " + attrAddress[1]);
System.out.println("state: " + attrAddress[2]);
System.out.println("zip: " + attrAddress[3]);
```

8.6.2 コレクションを使用する

AdvancedServer データベースにコレクションタイプ NUMBER_ARRAY および CHAR_ARRAY を作成します。

```
CREATE OR REPLACE TYPE NUMBER_ARRAY AS TABLE OF NUMBER;
CREATE OR REPLACE TYPE CHAR_ARRAY AS TABLE OF VARCHAR(50);
```

リスト 1.17-a は、コレクションタイプ NUMBER_ARRAY および CHAR_ARRAY を IN パラメーターとして使用し、CHAR_ARRAY を OUT パラメーターとして使用する SPL 関数です。

この関数は、NUMBER_ARRAY IN パラメーターの従業員 ID と、CHAR_ARRAY IN パラメーターの対応する行の従業員名を連結します。結果の連結されたエントリは CHAR_ARRAY OUT パラメータで返されます。

```
CREATE OR REPLACE FUNCTION concatEmpIdName
(
    arrEmpIds      NUMBER_ARRAY,
    arrEmpNames    CHAR_ARRAY
) RETURN CHAR_ARRAY
AS
DECLARE
    i              INTEGER := 0;
    arrEmpIdNames  CHAR_ARRAY;
BEGIN
    arrEmpIdNames := CHAR_ARRAY(NULL, NULL);
    FOR i IN arrEmpIds.FIRST..arrEmpIds.LAST LOOP
        arrEmpIdNames(i) := arrEmpIds(i) || ' ' || arrEmpNames(i);
    END LOOP;
    RETURN arrEmpIdNames;
END;
```

リスト 1.17-b は、リスト 1.17-a 関数を呼び出し、コレクション型を渡して取得する Java メソッドです。

```
public static void testTableOfAsInOutParams() throws SQLException {
    Connection conn = null;
    try {
        conn = getEDBConnection();
        String commandText = "{? = call concatEmpIdName(?,?)}";
        CallableStatement stmt = conn.prepareCall(commandText);

        // create collections to specify employee id and name values
        Array empIdArray = conn.createArrayOf("integer",
            new Integer[]{7900, 7902});
        Array empNameArray = conn.createArrayOf("varchar",
            new String[]{"JAMES", "FORD"});

        // set TABLE OF VARCHAR as OUT param
        stmt.registerOutParameter(1, Types.ARRAY);
```

(次のページに続く)

(前のページからの続き)

```

// set TABLE OF INTEGER as IN param
stmt.setObject(2, empIdArray, Types.OTHER);

// set TABLE OF VARCHAR as IN param
stmt.setObject(3, empNameArray, Types.OTHER);
stmt.execute();
java.sql.Array empIdNameArray = stmt.getArray(1);
String[] emps = (String[]) empIdNameArray.getArray();

System.out.println("items length: " + emps.length);
System.out.println("items[0]: " + emps[0].toString());
System.out.println("items[1]: " + emps[1].toString());

} catch (ClassNotFoundException cnfe) {
    System.err.println("Error: " + cnfe.getMessage());
} finally {
    if (conn != null) {
        conn.close();
    }
}
}

```

CallableStatement オブジェクトは concatEmpIdName () 関数を呼び出すために準備されます:

```

String commandText = "{? = call concatEmpIdName(?,?)}";
CallableStatement stmt = conn.prepareCall(commandText);

```

createArrayOf () は empIdArray と empNameArray という名前のコレクションを初期化して作成します:

```

Array empIdArray = conn.createArrayOf("integer",
    new Integer[]{7900, 7902});
Array empNameArray = conn.createArrayOf("varchar",
    new String[]{"JAMES", "FORD"});

```

registerOutParameter () を呼び出すと、OUT パラメーターのパラメータータイプ (Types.ARRAY) が登録されます。

```

stmt.registerOutParameter(1, Types.ARRAY);

```

setObject () メソッドは、コレクション empIdArray および empNameArray を IN プレースホルダーにバインドします。

```

stmt.setObject(2, empIdArray, Types.OTHER);
stmt.setObject(3, empNameArray, Types.OTHER);

```

stmt.execute () への呼び出しは concatEmpIdName () 関数を呼び出します:

```
stmt.execute();
```

`getArray()` は、関数によって返されたコレクションを取得します。連結された従業員 ID と名前で構成される最初の 2 行が表示されます。

```
java.sql.Array empIdNameArray = stmt.getArray(1);
String[] emps = (String[]) empIdNameArray.getArray();
System.out.println("items length: " + emps.length);
System.out.println("items[0]: " + emps[0].toString());
System.out.println("items[1]: " + emps[1].toString());
```

8.7 NoticeListener による非同期通知処理

The Advanced Server JDBC Connector provides asynchronous notification handling functionality. A notification is a message generated by the server when an SPL (or PL/pgSQL) program executes a RAISE NOTICE statement. Each notification is sent from the server to the client application. To intercept a notification in a JDBC client, an application must create a NoticeListener object (or, more typically, an object derived from NoticeListener).

It is important to understand that a notification is sent to the client as a result of executing an SPL (or PL/pgSQL) program. To generate a notification, you must execute an SQL statement that invokes a stored procedure, function, or trigger: the notification is delivered to the client as the SQL statement executes. Notifications work with any type of statement object; CallableStatement objects, PreparedStatement objects, or simple Statement objects. A JDBC program intercepts a notification by associating a NoticeListener with a Statement object. When the Statement object executes an SQL statement that raises a notice, JDBC invokes the noticeReceived() method in the associated NoticeListener.

リスト 1.18-a は emp テーブルをループして各従業員に 10 % の昇給を与える SPL プロシージャを示しています。各従業員が処理されると、adjustSalary は RAISE NOTICE ステートメントを実行します（この場合、通知に含まれるメッセージはクライアントアプリケーションに進行状況を報告します）。

```
CREATE OR REPLACE PROCEDURE adjustSalary
IS
  v_empno      NUMBER(4);
  v_ename      VARCHAR2(10);
  CURSOR emp_cur IS SELECT empno, ename FROM emp;
BEGIN
  OPEN emp_cur;
  LOOP
    FETCH emp_cur INTO v_empno, v_ename;
    EXIT WHEN emp_cur%NOTFOUND;

    UPDATE emp SET sal = sal * 1.10 WHERE empno = v_empno;
    RAISE NOTICE 'Salary increased for %', v_ename;
  END LOOP;
  CLOSE emp_cur;
END;
```

リスト 1.18-b は、JDBC アプリケーションで通知をインターセプトする NoticeListener を作成する方法を示しています。

```
public void NoticeExample(Connection con)
{
  CallableStatement stmt;
  try
  {
```

(次のページに続く)

(前のページからの続き)

```

    stmt = con.prepareCall("{call adjustSalary()}");

    MyNoticeListener listener = new MyNoticeListener();
    ((BaseStatement)stmt).addNoticeListener(listener);
    stmt.execute();
    System.out.println("Finished");
}
catch (SQLException e)
{
    System.out.println("An error has occurred.");
    System.out.println("See full details below.");
    e.printStackTrace();
}
}
class MyNoticeListener implements NoticeListener
{
    public MyNoticeListener()
    {
    }

    public void noticeReceived(SQLWarning warn)
    {
        System.out.println("NOTICE: " + warn.getMessage());
    }
}

```

NoticeExample() メソッドは簡単です。呼び出し元からの単一の引数、Connection オブジェクトを期待します:

```
public void NoticeExample(Connection con)
```

NoticeExample() begins by preparing a call to the adjustSalary procedure shown in example 1.10-a. As you would expect, con.prepareCall() returns a CallableStatement object. Before executing the CallableStatement, you must create an object that implements the NoticeListener interface and add that object to the list of NoticeListeners associated with the CallableStatement:

```

CallableStatement stmt = con.prepareCall("{call adjustSalary()}");
MyNoticeListener listener = new MyNoticeListener();
((BaseStatement)stmt).addNoticeListener(listener);

```

NoticeListener が配置されると、NoticeExample メソッドは CallableStatement を実行し（サーバーで adjustSalary プロシージャを呼び出します）、ユーザーにメッセージを表示します。

```

stmt.execute();
System.out.println("Finished");

```

adjustSalary プロシージャが RAISE NOTICE ステートメントを実行するたびに、サーバーはメッセージのテキスト (/"Salary increased for .../") をクライアントアプリケーションの Statement (または派生) オブジェクトに送信します。JDBC は noticeReceived() メソッドを (おそらく何度も) 呼び出します before stmt.execute() への呼び出しは完了します。

```
class MyNoticeListener implements NoticeListener
{
    public MyNoticeListener()
    {
    }

    public void noticeReceived(SQLWarning warn)
    {
        System.out.println("NOTICE: "+ warn.getMessage());
    }
}
```

JDBC は noticeReceived() メソッドを呼び出すと、サーバー上の RAISE NOTICE ステートメントによって生成されたメッセージのテキストを含む SQLWarning オブジェクトを作成します。

Notice that each Statement object keeps a list of NoticeListeners`. When the JDBC driver receives a notification from the server, it consults the list maintained by the Statement object. If the list is empty, the notification is saved in the Statement object (you can retrieve the notifications by calling stmt.getWarnings() once the call to execute() completes). If the list is not empty, the JDBC driver delivers an SQLWarning to each listener, in the order in which the listeners were added to the Statement.

第 9

セキュリティと暗号化

9.1 SSL の使用

このセクションでは、以下について学習します。

- サーバーの構成
- クライアントの構成
- SSLJDBC 接続のテスト
- 証明書検証なしで SSL を使用する
- 証明書認証の使用（パスワードなし）

9.1.1 サーバーの設定

SSL 用の PostgreSQL または AdvancedServer の構成については、次を参照してください。

<https://www.enterprisedb.com/edb-docs/d/postgresql/reference/manual/12.3/ssl-tcp.html>

注釈: Java から SSL 対応サーバーにアクセスする前に、`edb-psql` を介してサーバーにログインできることを確認してください。SSL 接続を確立した場合、サンプル出力は次のようになります。

```
$ ./bin/edb-psql -U enterprisedb -d edb
psql.bin (12.0.1)
SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384, bits: 256,
↪compression: off)
Type "help" for help.

edb=#
```

9.1.2 クライアントの設定

クライアントを SSL 用に構成するための接続パラメーターがいくつかあります。SSL 接続パラメーターと追加の接続プロパティの詳細については、```セクション 5.2<# connecting-to-the-database>```を参照してください。

このセクションでは、異なる値で渡された場合の ssl 接続パラメーターの動作について詳しく説明します。接続パラメーター `ssl=true` をドライバーに渡すと、ドライバーは SSL 証明書を検証し、ホスト名を検証します。この動作とは反対に、`libpq` を使用すると、デフォルトでは検証されない SSL 接続になります。

`sslmode` 接続パラメーターを使用すると、SSL 接続をより適切に制御できます。このパラメーターは `libpq` `sslmode` パラメーターと同じで、既存の SSL は次の `sslmode` 接続パラメーターを実装します。

sslmode 接続パラメーター

sslmode=require

このモードでは暗号化が必須になり、暗号化できない場合は接続が失敗する必要があります。サーバーは、このホスト/IP アドレスの SSL 接続を受け入れるように構成されており、サーバーがクライアント証明書を認識します。

注釈: このモードでは、JDBC ドライバーはすべてのサーバー証明書を受け入れます。

sslmode=verify-ca

`sslmode=verify-ca` の場合、サーバーは、クライアントに保存されているルート証明書までの証明書チェーンをチェックすることにより検証されます。

sslmode=verify-full

`sslmode=verify-full` の場合、サーバーのホスト名が検証され、サーバー証明書に保存されている名前と一致することが確認されます。サーバー証明書を検証できない場合、SSL 接続は失敗します。このモードは、ほとんどのセキュリティが重要な環境で推奨されます。

証明書の検証に失敗した場合、`sslcert=` を試すことができ、`LibPQFactory` はクライアント証明書を送信しません。サーバーが証明書を使用して認証するように構成されていない場合は、接続する必要があります。

クライアント証明書、クライアントキー、ルート証明書の場所は、それぞれ `sslcert`、`sslkey`、`sslrootcert` の設定で上書きできます。これらのデフォルトはそれぞれ `/defaultdir/postgresql.crt`、`/defaultdir/postgresql.pk8`、および `/defaultdir/root.crt` で、`defaultdir` は UNIX システムでは `${user.home}/.postgresql/`、Windows では `%appdata%/postgresql/` です。

このモードでは、SSL 接続を確立するときに JDBC ドライバーがサーバーの ID を検証し、```中間者``` 攻撃を防ぎます。これは、サーバー証明書が信頼できる機関によって署名されていること、および接続先のホストが証明書のホスト名と同じであることを確認することでこれを行います。

9.1.3 SSLJDBC 接続のテスト

Java のデフォルトメカニズム（LibPQFactory ではない）を使用して SSL 接続を作成している場合、サーバー証明書を Java で使用可能にする必要があります。これは、以下に示す手順を実装することで実現できます。

1. Java プログラムで以下のプロパティを設定します。

```
props.setProperty( "ssl" , "true" );
```

または、接続 URL でプロパティを設定できます。

文字列 url= “jdbc:edb://localhost/test? user=fred & password=secret & ssl=true” ;

2. サーバー証明書を Java 形式に変換します。

```
$ openssl x509 -in server.crt -out server.crt.der -outform der
```

3. この証明書を Java のシステムトラストストアにインポートします。

```
$ keytool -keystore $JAVA_HOME/lib/security/cacerts -alias postgresql  
-import -file server.crt.der
```

注釈: cacerts キーストアのデフォルトのパスワードは **changeit** です。postgresql のエイリアスは重要ではなく、任意の名前を選択できます。

4. システム cacerts トラストストアにアクセスできない場合は、次のように独自のトラストストアを作成します。

```
$ keytool -keystore mystore -alias postgresql -import -file server.crt.der
```

5. Java アプリケーションを起動し、プログラムをテストします。

```
$ java -Djavax.net.ssl.trustStore=mystore com.mycompany.MyApp
```

例:

```
$ java -classpath ./usr/edb/jdbc/edb-jdbc18.jar-  
Djavax.net.ssl.trustStore=mystore pg_test2 public
```

注釈: 接続の問題のトラブルシューティングを行うには、java コマンドに **-Djavax.net.debug=ssl** を追加します。

証明書検証なしで **SSL** を使用する

デフォルトでは、`SSL=true` と接続 URL パラメーター `sslfactory=com.edb.ssl.NonValidatingFactory` の設定により、接続が暗号化されますが、**SSL** 証明書は検証されません。証明書の検証を強制するには、Custom `SSLSocketFactory` を使用する必要があります。

Custom `SSLSocketFactory` の書き込みの詳細については、以下を参照してください:

<https://jdbc.postgresql.org/documentation/head/ssl-factory.html>

9.1.4 証明書認証の使用（パスワードなし）

証明書認証（パスワードなし）を使用するには、次の手順に従います。

1. クライアント証明書を DER 形式に変換します。

```
$ openssl x509 -in postgresql.crt -out postgresql.crt.der -outform der
```

2. クライアントキーを DER 形式に変換します。

```
$ openssl pkcs8 -topk8 -outform DER -in postgresql.key -out postgresql.key.pk8 -nocrypt
```

3. クライアントファイル（postgresql.crt.der、postgresql.key.pk8）とルート証明書をクライアントマシンにコピーし、Java プログラムで次のプロパティを使用してテストします。

```
String url = "jdbc:edb://snvm001:5444/edbstore";
Properties props = new Properties();
props.setProperty("user", "enterprisedb");
props.setProperty("ssl", "true");
props.setProperty("sslmode", "verify-full");
props.setProperty("sslcert", "postgresql.crt.der");
props.setProperty("sslkey", "postgresql.key.pk8");
props.setProperty("sslrootcert", "root.crt");
```

4. Java プログラムをコンパイルしてテストします。

```
$ java -Djavax.net.ssl.trustStore=mystore -classpath
../edb-jdbc18.jar pg_ssl public
```

9.2 ScramCompatibility

EDBJDBC ドライバーは、AdvancedServer バージョン 10、11、および 12 の SCRAM-SHA-256 サポートを提供します。JRE/JDK バージョン 1.8 の場合、このサポートは EDBJDBC コネクタリリース 42.2.2.1 以降で利用可能です。JRE/JDK バージョン 1.7 の場合、このサポートは EDBJDBC コネクタリリース 42.2.5 以降で利用できます。

第 10

AdvancedServerJDBCConnectorLogging

The Advanced Server JDBC Connector supports the use of logging to help resolve issues with the JDBC Connector when used in your application. The JDBC Connector uses the logging APIs of `java.util.logging` that was part of Java since JDK 1.4. For information on `java.util.logging`, see [The PostgreSQL JDBC Driver](#).

注釈: `AdvancedServerJDBCConnector` の以前のバージョンは、ロギングを有効にするためにカスタムメカニズムを使用していましたが、コミュニティバージョン 42.1.4.1 から前進するバージョンでは `java.util.logging` の使用に置き換えられています。古いメカニズムは使用できなくなりました。

10.1 接続プロパティを使用したロギングの有効化（静的）

JDBC コネクタの接続プロパティ内でロギングを直接構成できます。ロギングに関連する接続プロパティは、`loggerLevel` および `loggerFile` です。

`loggerLevel`

ドライバーのロガーレベル。許可される値は、OFF、DEBUG、または TRACE です。このオプションにより、ドライバーの `java.util.logging.Logger` レベルが次の `AdvancedServerJDBC` レベルに対応するようになります。

loggerLevel	java.util.logging
オフ	オフ
デバッグ	ファイン
トレース	最高

loggerFile

ロガーのファイル名出力。デフォルトは `java.util.logging.ConsoleHandler` です。次の例では、ログレベルを `TRACE` (`FINEST`) に設定し、ログファイルを `to EDB-JDBC.LOG` に設定します。

```
jdbc:edb://myhost:5444/mydb?loggerLevel=TRACE&loggerFile=EDB-JDBC.LOG
```

10.2 logging.properties によるロギングの有効化（動的）

ロギングプロパティを使用してドライバーを動的に構成できます（たとえば、Tomcat、JBoss、WildFly などのアプリケーションサーバーで JDBC コネクタを使用する場合）。これにより、実行時にロギングを簡単に有効/無効にできます。次の例は、ロギングを動的に構成する方法を示しています。

```
handlers = java.util.logging.FileHandler
//logging level
.level = OFF
```

デフォルトのファイル出力は、ユーザーのホームディレクトリにあります。

```
java.util.logging.FileHandler.pattern = %h/EDB-JDBC%u.log
java.util.logging.FileHandler.limit = 5000000
java.util.logging.FileHandler.count = 20
java.util.logging.FileHandler.formatter = java.util.logging.SimpleFormatter
java.util.logging.FileHandler.level = FINEST
java.util.logging.SimpleFormatter.format=%1$tY-%1
    $tm-%1$td %1$tH:%1$tM:%1$tS %4$s %2$s %5$s%6$s%n
```

次のコマンドを使用して、JDBC コネクタのログレベルを **FINEST** に設定します（`loggerLevel` にマップ）。

```
com.edb.level=FINEST
```

次に、ロギング構成でアプリケーションを実行します。

```
java -jar -Djava.util.logging.config.file=logging.properties run.jar
```

リファレンス-JDBC データ型

次の表に、AdvancedServer および JDBC コネクターでサポートされる JDBC データ型を示します。setObject() メソッドを使用して AdvancedServer タイプ（中央の列に表示）にバインドしている場合は、左の列に表示されているタイプの JDBC 値を指定します。データを取得すると、getObject() メソッドは右端の列にリストされたオブジェクトタイプを返します:

JDBC タイプ	高度なサーバータイプ	getObject () が返ります
整数	INT4	java.lang.Integer
タンニン、サリン	INT2	java.lang.Integer
ビッグ	INT8	java.lang.Long
実	FLOAT4	java.lang.Float
ダブル、フロート	FLOAT8	java.lang.Double (float は double と同じです)
10 進数、数値	数値	java.math.BigDecimal
CHAR	BPCHAR	java.lang.String
VARCHAR、LONGVAR-CHAR	VARCHAR	java.lang.String
DATE	DATE	java.sql.Date
時間	TIME、TIMETZ	java.sql.Timestamp
タイムスタンプ	タイムスタンプ、タイムスタンプ	java.sql.Timestamp
バイナリ	バイティア	byte[] (プリミティブ)
プール、ビット	プール	java.lang.Boolean
Types.REF	離席者	java.sql.ResultSet
タイプ.REF_CURSOR	離席者	java.sql.ResultSet
タイプ。その他	離席者	java.sql.ResultSet
タイプ。その他	UUID	java.util.UUID
Types.SQLXML	XML	java.sql.SQLXML

注釈: Types.REF_CURSOR は JRE4.2 でのみサポートされます。

Types.OTHER は UUID に使用されるだけでなく、特定のタイプを指定せず、サーバーまたは JDBC ドライバーがタイプを決定できる場合にも使用されます。パラメーターが java.util.UUID のインスタンスである場合、ドライバは適切な内部タイプを決定し、サーバーに送信します。

結論

EDBPostgresAdvancedServerJDBC コネクタガイド

Copyright©2007-2020EnterpriseDBCorporation。

無断複写・転載を禁じます。

EnterpriseDB®Corporation

34CrosbyDrive、Suite201、Bedford、MA01730、アメリカ

T+17813573390F+19784671307E

info@enterprisedb.com

www.enterprisedb.com

- EDB は、存在する場合、EDBPostgres 製品のログオン UI の設計、コーディングのベストプラクティスの確立、レビュー、および入力検証の検証を行います。EDB は追加の入力コンポーネントに対して同じアプローチに従いますが、製品の性質上、限定された検証が可能な信頼できるユーザーによって入力および送信される自由形式の SQL、WMI またはその他の文字列を受け入れる必要があります。このような場合、ユーザーが誤った入力やその他の危険な入力を防ぐことはできません。
- EDB は、将来、認証された信頼できるユーザーからのフリーフォーム SQL、WMI またはその他の潜在的に危険な入力を受け入れる製品に機能を追加する権利を留保しますが、そのような機能はすべて、可能な限り最小のリスクを提供するように設計およびテストされます可能であれば、スーパーユーザーまたは同等の特権が必要です。

- EDB は、すべての潜在的な脅威を予測できることを保証しません。したがって、すべての潜在的な脆弱性が対処または検討されたことをプロセスが完全に保証することはできません。

索引

Advanced Server JDBC Connector Compatibility, 10
 AdvancedServerJDBCCConnectorLogging, 93
 AdvancedServerJDBCCConnectorOverview, 5
 AdvancedServerJDBCCConnector の設定, 24
 AdvancedServerJDBC コネクタの読み込み, 27

Debian または Ubuntu ホストへの DEB パッケージのインストール, 18

enable logging with connection properties, 93
 enable logging with logging.properties, 95
 executeupdate() syntax examles, 48
 executeUpdate () による SQL コマンドの実行, 45
 executing a simple stored procedure, 60
 executing stored procedures with in out parameters, 66
 executing stored procedures with in parameters, 62
 executing stored procedures with out parameters, 64

inserting bytea data into advanced server database, 73
 invoking stored procedures, 59

Java アプリケーションでの AdvancedServerJDBCCConnector の使用, 25
 Java で BYTEA データを使用する, 72
 Java で REF カーソルを使用する, 69
 Java でのオブジェクトタイプとコレクションの使用, 77
 jdbc classes, 8
 jdbc driver types, 5
 jdbc drivermanager, 9
 jdbc interface, 7
 JDBC コネクタのインストールと設定, 12

modifying batch size of statement object, 55

NoticeListener による非同期通知処理, 84

PreparedStatement を使用して SQL コマンドを送信する, 56

ResultSet オブジェクトからの結果の取得, 42
 retrieving bytea data from advanced server database, 75
 RPM パッケージを使用したコネクタのインストール, 13

ScramCompatibility, 92
 SLES12 ホストへのコネクタのインストール, 16

SSLJDBC 接続のテスト, 90
 sslmode connection parameters, 89
 SSL の使用, 87
 supported platforms, 3
 supported server versions, 3

using collection, 81
 using executeupdate() to insert data, 46
 using named notation with a callablestatement object, 40
 using object type, 78
 using ref cursor to retrieve a resultset, 69
 using ssl without certificate validation, 91

エラーの処理, 44

クライアントの設定, 89
 クライアント側のリソース要件の削減, 54
 グラフィカルインストーラを使用してコネクタをインストールする, 19
 グラフィカルインターフェイスを Java プログラムに追加する, 49

サーバーの設定, 88

ステートメントオブジェクトによる SQL ステートメントの実行, 39
 ストアドプロシージャの実行, 58

セキュリティと暗号化, 87

データベースへの接続, 28

リソースの解放, 43
 リファレンス-JDBC データ型, 96

結論, 98
 高度な JDBC コネクタ機能, 53
 証明書認証の使用 (パスワードなし), 92
 新機能, 2
 追加の接続プロパティ, 29
 同期セカンダリデータベースサーバーの設定, 32
 要件の概要, 3