



EDB

EDB Postgres Advanced Server JDBC Connector

リリース *42.5.0.1*

EnterpriseDB Corporation

2022 年 09 月 19 日

目次

第 1 章	Release notes	3
1.1	Version 42.5.0.1	3
1.2	Version 42.3.3.1	4
1.3	Version 42.3.2.1	4
1.4	Version 42.2.24.1	5
1.5	Version 42.2.19.1	5
1.6	Version 42.2.12.3	6
1.7	Version 42.2.9.1	6
1.8	Version 42.2.8.1	6
第 2 章	Product compatability	9
2.1	サポートされている JDK ディストリビューション	9
第 3 章	EDB Postgres Advanced Server JDBC Connector overview	11
3.1	JDBC ドライバーの種類	11
3.2	JDBC インターフェース	12
3.3	JDBC のクラスとインターフェース	12
3.4	JDBC DriverManager	12
3.5	EDB Postgres Advanced Server JDBC コネクタの互換性	12
第 4 章	Installing and configuring EDB JDBC Connector	15
4.1	Installing EDB JDBC Connector on Linux	15
4.1.1	Installing EDB JDBC Connector on Linux x86 (amd64)	16
4.1.2	Installing EDB JDBC Connector on IBM Power (ppc64le)	23
4.2	Using the graphical installer to install the connector	26
4.3	Configuring the EDB Postgres Advanced Server JDBC Connector	29
4.4	Upgrading a Linux installation	33
第 5 章	Using the EDB Postgres Advanced Server JDBC Connector with Java applications	35
5.1	Loading the EDB Postgres Advanced Server JDBC Connector	36
5.2	Connecting to the database	37
5.2.1	Additional connection properties	37
5.2.2	Preferring synchronous secondary database servers	39
5.3	Executing SQL statements through statement objects	46
5.3.1	CallableStatement オブジェクトで名前付け表記を使用する	46
5.4	Retrieving results from a ResultSet object	48
5.5	Freeing resources	49
5.6	Handling errors	49
第 6 章	Using advanced queueing	51
6.1	メッセージのエンキューまたはデキュー	51

6.2	サーバーサイドのセットアップ	51
6.2.1	EDB-PSQL の使用	51
6.3	クライアントサイドの例	53
6.3.1	メッセージのエンキューとデキュー	55
6.3.2	メッセージをエンキューする	55
6.3.3	メッセージをデキューする	55
6.4	完全なエンキューおよびデキュープログラム	56
第 7 章	Executing SQL commands with executeUpdate() or through PreparedStatement objects	61
7.1	executeUpdate() を使用してデータを挿入する	61
7.2	PreparedStatement を使用して SQL コマンドを送信する	62
第 8 章	Adding a graphical interface to a Java program	65
第 9 章	Advanced JDBC Connector functionality	69
9.1	Reducing client-side resource requirements	69
9.1.1	ステートメントオブジェクトのバッチサイズの変更	69
9.2	Using PreparedStatement to send SQL commands	70
9.3	Executing stored procedures	72
9.3.1	ストアドプロシージャの呼び出し	72
9.3.2	IN OUT パラメーターを使用したストアドプロシージャの実行	77
9.4	Using REF CURSORS with Java	79
9.4.1	REF CURSOR を使用して ResultSet を取得する	79
9.5	Using BYTEA data with Java	81
9.5.1	EDB Postgres Advanced Server への BYTEA データの挿入	82
9.5.2	EDB Postgres Advanced Server データベースから BYTEA データを取得する	84
9.6	Using object types and collections with Java	85
9.6.1	オブジェクトタイプの使用	86
9.6.2	コレクションの使用	89
9.7	Asynchronous notification handling with NoticeListener	91
第 10 章	Security and encryption	95
10.1	Using SSL	95
10.1.1	Configuring the server	95
10.1.2	Configuring the client	96
10.1.3	Testing the SSL JDBC connection	97
10.1.4	Using certificate authentication without a password	98
10.2	Scram compatibility	98
10.3	Support for GSSAPI-encrypted connection	99
10.3.1	GSSAPI/SSPI 認証	99
10.3.2	SSPI の使用 (Windows のみの環境)	99
10.3.3	GSSAPI の使用 (Linux のみの環境)	100
10.3.4	SSPI/GSSAPI の使用 (Windows および Linux 環境)	101
第 11 章	EDB Postgres Advanced Server JDBC Connector logging	103
11.1	logging.properties でロギングを有効にする	103
第 12 章	Reference - JDBC data types	105

EDB JDBC コネクタは、Java アプリケーションと EDB Postgres Advanced Server データベース間の接続を提供します。EDB JDBC Connector は Java で書かれ、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、

[JDBC ドライバーの種類](#) を参照してください

EDB JDBC Connector は、PostgreSQL コミュニティドライバのすべての機能をサポートしています。ドライバの特長と機能の詳細については、コミュニティ

[documentation](#) を参照してください。

[release_notes](#) [requirements_overview](#) [advanced_server_jdbc_connector_overview](#) [installa-](#)
[tion_and_configuring_the_jdbc_connector_using_the_advanced_server_jdbc_connector_with_java_applicationsunning_sql_command](#)

第1章 Release notes

EDB JDBC コネクタのドキュメントには、EDB JDBC コネクタの最新バージョンが説明されています。

これらのリリースノートでは、各リリースの新機能について説明します。マイナーリリースまたはパッチリリースで新機能が導入されると、コンテンツ内のインジケータによって、新機能が導入されたバージョンが識別されます。

Version	Release Date
<i>Version 42.5.0.1</i>	2022 Sep 1
<i>Version 42.3.3.1</i>	2022 Apr 20
<i>Version 42.3.2.1</i>	2022 Feb 15
<i>Version 42.2.24.1</i>	2021 Nov 5
<i>Version 42.2.19.1</i>	2021 Apr 15
<i>Version 42.2.12.3</i>	2020 Oct 22
<i>Version 42.2.9.1</i>	2020 May 18
<i>Version 42.2.8.1</i>	2019 Oct 21

1.1 Version 42.5.0.1

EDB JDBC コネクタは、Java アプリケーションと EDB Postgres Advanced Server データベース間の接続を提供します。

EDB JDBC Connector 42.5.0.1 の新機能、機能強化、バグ修正、およびその他の変更には次のものが含まれます。

Type	Description
Upstream Merge	アップストリームのコミュニティドライババージョン 42.5.0 と統合されました。詳細については、コミュニティ `JDBC documentation` < https://jdbc.postgresql.org/documentation/changelog.html#version_42.5.0 >`_`を参照してください。
Security Fix	CVE-2022-31197 - PgResultSet.refresh() で生成された SQL を修正して、SQL インジェクションを防ぐために列識別子をエスケープします。以前は、キー列とデータ列の両方の列名が、生成された SQL にそのままコピーされました。これにより、ステートメントターミネータを含む列名を持つ悪意のあるテーブルが、複数の個別のコマンドとして解析および実行されました。また、この修正により、この変更を確認するための新しいテストクラス ResultSetRefreshTest が追加されます。

次のページに続く

表 2 – 前のページからの続き

Type	Description
Change	ビルドを Gradle に移行しました。
Enhancement	新しい <i>changeServerName</i> 接続プロパティを追加しました。`changeServerName` の値が <code>true</code> に設定されている場合、`getServerName()` 呼び出しは `PostgreSQL` として値を返します。デフォルト値は `false` です。
Enhancement	新しい <i>forceBinaryTransfer</i> 接続プロパティを追加しました。値が `true` に設定されている場合、すべてのバイナリ型を PostgreSQL サーバーから JDBC ドライバーにバイナリ形式で転送します。デフォルト値は `false` です。

1.2 Version 42.3.3.1

EDB JDBC コネクタは、Java アプリケーションと EDB Postgres Advanced Server データベース間の接続を提供します。

EDB JDBC Connector 42.3.3.1 の新機能、機能強化、バグ修正、およびその他の変更には次のものが含まれます。

Type	Description
Upstream Merge	アップストリームのコミュニティドライババージョン 42.3.3 と統合されました。詳細については、コミュニティ JDBC documentation <https://jdbc.postgresql.org/documentation/changelog.html#version_42.3.3> を参照してください。
Security Fix	GHSA-673j-qm5f-xpv8 ：この修正の一部として、 <code>loggerFile</code> および <code>loggerLevel</code> 構成プロパティを削除しました。プロパティはまだ存在しますが、ドライバログの構成には使用できません。代わりに、`logging.properties` などの <code>java.util.logging</code> 構成メカニズムを使用してください。
Change	セキュリティ修正 GHSA-673j-qm5f-xpv の一部として、接続プロパティを使用してログを有効にする機能は、バージョン 42.3.3 で使用できなくなりました。

1.3 Version 42.3.2.1

EDB JDBC コネクタは、Java アプリケーションと EDB Postgres Advanced Server データベース間の接続を提供します。

EDB JDBC Connector 42.3.2.1 の新機能、機能強化、バグ修正、およびその他の変更には次のものが含まれます。

Type	Description
Upstream merge	Merged with the upstream community driver version 42.3.2. See the community JDBC documentation for details.

次のページに続く

表 4 – 前のページからの続き

Type	Description
New feature	<i>org.checkerframework.*</i> was previously packaged in the EDB JDBC jar file; causing conflicts with other applications utilizing <i>org.checkerframework.*</i> with different versions. New feature is packaging the checker framework under a custom namespace in the connector using the shade plugin. [Support Ticket:#74134]
New feature	JMS based API to interact with DBMS_AQ package seamlessly. This API has been made part of edb-jdbc code under com.edb.jms and com.edb.aq packages.
Enhancement	New property oidTimestamp used to change default behavior of driver when using setTimeStamp method for preparedStatement. If property oidTimestamp it is set to true, sets the oid to Oid.TIMESTAMP, otherwise uses default behavior.
Bug fix	Issue: Change in date format nls_date_format=' YYYY/MM/DD' in EDB*PLUS gives error. [Support Ticket:#75812]
Bug fix	Rounding differences between EDB and Oracle. [Support Ticket:#72708]
Security fix	CVE-2022-21724 as part of community merge with v42.3.2
Security fix	CVE-2021-36373 - Removed dependency for org.apache.ant
Security fix	CVE-2020-15250 - junit fix for temporary folder.

1.4 Version 42.2.24.1

EDB JDBC コネクタは、Java アプリケーションと Advanced Server データベース間の接続を提供します。

EDB JDBC Connector 42.2.24.1 の新機能、機能強化、バグ修正、およびその他の変更には次のものが含まれます。

Type	Description
Upstream merge	アップストリームのコミュニティドライババージョン 42.2.24 と統合されました。詳細については、コミュニティ `JDBC documentation` < https://jdbc.postgresql.org/documentation/head/index.html >`__`を参照してください。

1.5 Version 42.2.19.1

EDB JDBC コネクタは、Java アプリケーションと Advanced Server データベース間の接続を提供します。

EDB JDBC Connector 42.2.19.1 の新機能、機能強化、バグ修正、およびその他の変更には次のものが含まれます。

Type	Description
Upstream merge	アップストリームのコミュニティドライババージョン 42.2.19 と統合されました。詳細は `community JDBC documentation` < https://jdbc.postgresql.org/documentation/head/index.html >`__`を参照してください。

次のページに続く

表 6 – 前のページからの続き

Type	Description
Enhancement	EDB JDBC Connector は、GSSAPI 暗号化接続をサポートするようになりました。 :ref:`Support for GSSAPI Encrypted Connection <../09_security_and_encryption/03_support_for_gssapi_encrypted_connection>`を参照してください。

注釈: EDB JDBC コネクタ v42.2.12.3 は Java 1.6 および 1.7 をサポートしていません。 EDB JDBC Connector の以前のバージョンは、引き続き Java 1.6 および 1.7 をサポートしています。

1.6 Version 42.2.12.3

EDB JDBC コネクタは、Java アプリケーションと Advanced Server データベース間の接続を提供します。

EDB JDBC Connector 42.2.12.3 の新機能、機能強化、バグ修正、およびその他の変更には次のものが含まれます。

Type	Description
Enhancement	EDB JDBC コネクタは、EDB Postgres Advanced Server 13 をサポートするようになりました。

1.7 Version 42.2.9.1

EDB JDBC コネクタは、Java アプリケーションと Advanced Server データベース間の接続を提供します。

EDB JDBC Connector 42.2.9.1 の新機能、機能強化、バグ修正、およびその他の変更には次のものが含まれます。

Type	Description
Enhancement	EDB JDBC Connector が Red Hat Enterprise Linux および CentOS (x86_64) 8.x でサポートされるようになりました。

1.8 Version 42.2.8.1

EDB JDBC コネクタは、Java アプリケーションと Advanced Server データベース間の接続を提供します。

EDB JDBC Connector 42.2.8.1 の新機能、機能強化、バグ修正、およびその他の変更には次のものが含まれます。

Type	Description
Upstream merge	アップストリームのコミュニティドライバーバージョン 42.2.8 と統合されました。詳細については、コミュニティ `JDBC documentation` < https://jdbc.postgresql.org/documentation/head/index.html >`__`を参照してください。
Enhancement	EDB JDBC コネクタは、EDB Postgres Advanced Server 12 をサポートするようになりました。
Enhancement	EDB JDBC Connector が Windows Server 2019 プラットフォームでサポートされるようになりました。

第2章 Product compatability

この表は、最新の JDBC コネクタバージョンと、サポートされている対応する EDB Postgres Advanced Server (EPAS) バージョンをリストしています。JDBC コネクタは、EDB Postgres Advanced Server と同じプラットフォームでサポートされています。詳細は [Product Compatibility](#) を参照してください。

JDBC Connector	EPAS 14	EPAS 13	EPAS 12	EPAS 11	EPAS 10
<i>Version 42.5.0.1</i>	Y	Y	Y	Y	Y
<i>Version 42.3.3.1</i>	Y	Y	Y	Y	Y
<i>Version 42.3.2.1</i>	Y	Y	Y	Y	Y
<i>Version 42.2.24.1</i>	Y	Y	Y	Y	Y
<i>Version 42.2.19.1</i>	N	Y	Y	Y	Y
<i>Version 42.2.12.3</i>	N	N	Y	Y	Y
<i>Version 42.2.9.1</i>	N	N	Y	Y	Y
<i>Version 42.2.8.1</i>	N	N	Y	Y	Y

2.1 サポートされている JDK ディストリビューション

Java 仮想マシン (JVM) : Java SE 8 以降 (LTS バージョン)、Oracle JDK、OpenJDK、および IBM SDK (Java) ディストリビューション。

第3章 EDB Postgres Advanced Server JDBC Connector overview

Sun Microsystems は、Java アプリケーションをデータベースに接続するための Java Database Connectivity (JDBC) と呼ばれる標準化されたインタフェースを作成しました。EDB JDBC 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 ドライバーの種類

現在、4 種類の JDBC ドライバーがあり、それぞれに独自の実装、使用法、および制限があります。EDB JDBC コネクタはタイプ 4 ドライバーです。

第 1 種ドライバー

- 本ドライバー種別は、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。
- コンピューターに ODBC がインストールされている必要があります。
- コンピューターにインストールされている特定のデータベース用の ODBC ドライバーが必要です。
- 通常、ネイティブメソッド呼び出しのため、アプレット内では実行できません。

2 型ドライバー

- 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、アプレット内では実行できません。
- 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はありません。
- クライアントにプリ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様に、その場でダウンロードして構成できます。
- ミドルウェアサーバーと通信するための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。

4 型ドライバー

- 100 % Java ドライバー、ネイティブメソッドはありません。
- クライアントにプリ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様に、その場でダウンロードして構成できます。
- Type3 ドライバーとは異なり、DBMS サーバと直接対話します。
- 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イスは `java.sql` パッケージに実装されています。

3.3 JDBC の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イスで構成されています。これらのクラスとインターフェイスは、図に示すように連携します。

3.4 JDBC DriverManager

この図は、一般的な JDBC アプリケーションでの `DriverManager` クラスの役割を示しています。`DriverManager` は、Java アプリケーションとバックエンドデータベース間のブリッジとして機能し、ターゲットデータベースに使用する JDBC ドライバーを決定します。

3.5 EDB Postgres Advanced Server JDBC コネクタの互換性

これはドライバーの現在のバージョンです。特別な要件（古いアプリケーションまたは JVM の実行）がない限り、これが使用する必要があります。このドライバーは PostgreSQL 10 以降のバージョンをサポートしており、Java 8 以降のバージョンが必要です。SSL と `javax.sql` パッケージのサポートが含まれています。

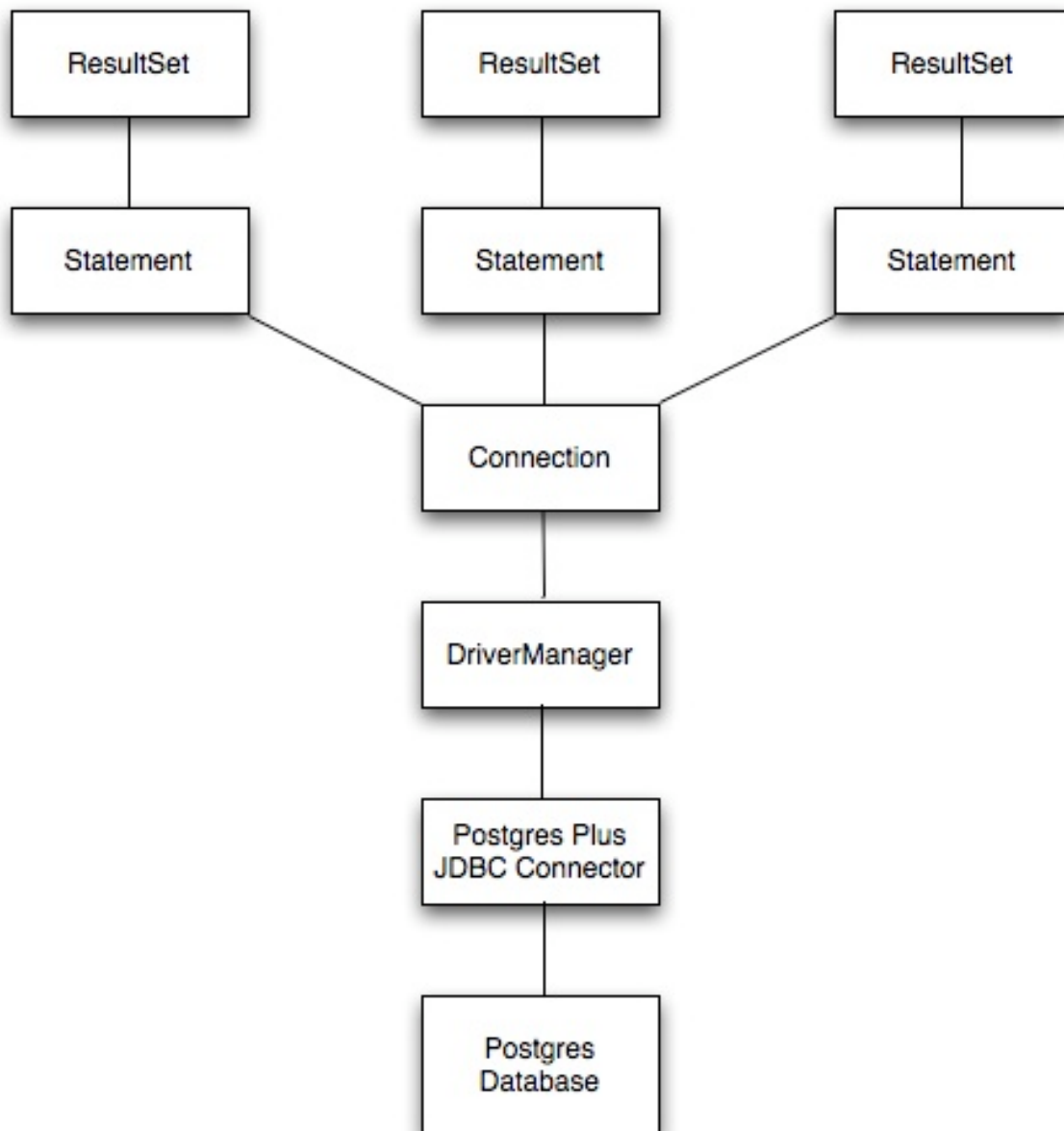


図 1: JDBC Class Relationships

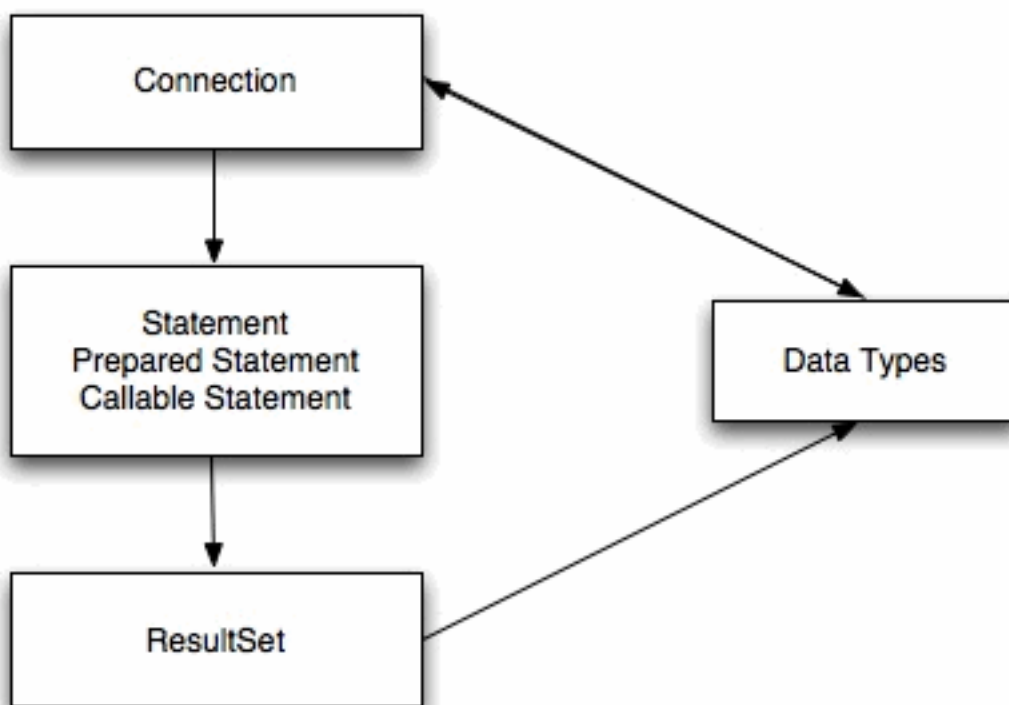


図 2: Core Classes and Interfaces

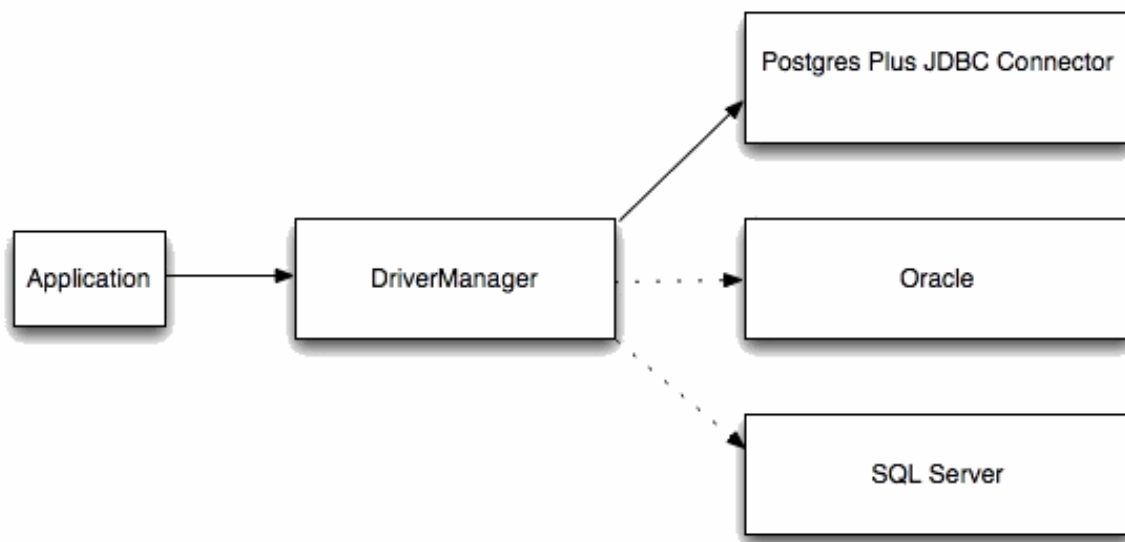


図 3: DriverManager/Drivers

第4章 Installing and configuring EDB JDBC Connector

EDB 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。Oracle Java Downloads [website](#) から、環境に合わせた 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 [Installation Instruction](#) リンクから入手できます。

EDB Postgres Advanced Server 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、インストールに EDB JDBC Connector を追加できます。

4.1 Installing EDB JDBC Connector on Linux

EDB JDBC Connector 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。EnterpriseDB リポジトリへのアクセスを許可する資格情報を要求するには、

[EDB Repository Access instructions](#) を参照してください。

プラットフォーム固有のインストール手順については、次を参照してください。

Linux x86-64 (amd64) :

- RHEL 8/OL 8
- Rocky Linux 8/AlmaLinux 8
- RHEL 7/OL 7
- *[Installing EDB JDBC Connector on CentOS 7 x86_64](#)*
- *[Installing EDB JDBC Connector on SLES 15 ppc64le](#)*
- *[Installing EDB JDBC Connector on SLES 12 ppc64le](#)*
- *[Installing EDB JDBC Connector on Ubuntu 20.04 x86_64](#)*
- *[Installing EDB JDBC Connector on Ubuntu 18.04 x86_64](#)*
- *[Installing EDB JDBC Connector on Debian 10 x86_64](#)*

Linux on IBM Power (ppc64le) :

- *[Installing EDB JDBC Connector on RHEL 8 ppc64le](#)*
- *[Installing EDB JDBC Connector on SLES 15 ppc64le](#)*
- *[Installing EDB JDBC Connector on SLES 12 ppc64le](#)*

4.1.1 Installing EDB JDBC Connector on Linux x86 (amd64)

オペレーティングシステム固有のインストール手順については、次を参照してください。

- RHEL 8/OL 8
- Rocky Linux 8/AlmaLinux 8
- RHEL 7/OL 7
- *Installing EDB JDBC Connector on CentOS 7 x86_64*
- *Installing EDB JDBC Connector on SLES 15 ppc64le*
- *Installing EDB JDBC Connector on SLES 12 ppc64le*
- *Installing EDB JDBC Connector on Ubuntu 20.04 x86_64*
- *Installing EDB JDBC Connector on Ubuntu 18.04 x86_64*
- *Installing EDB JDBC Connector on Debian 10 x86_64*

Installing EDB JDBC Connector on RHEL 8 or OL 8 x86_64

前提条件

インストールプロセスを開始する前に：

- 接続文字列を使用して接続できるホストに EDB Postgres Advanced Server をインストールします。同じホスト上にある必要はありません。

Installing EDB Postgres Advanced Server を参照してください。

- Supported JDBC distributions を確認します。
- リポジトリをセットアップする

リポジトリの設定は 1 回限りのタスクです。リポジトリを既に設定している場合、この手順を実行する必要はありません。

リポジトリをセットアップするには、

EDB repositories に移動し、そこに表示される指示に従います。

- その他の前提条件に対処する

```
# Install the EPEL repository:
sudo dnf -y install https://dl.fedoraproject.org/pub/epel/epel-release-latest-8.
↳noarch.rpm

# Enable additional repositories to resolve dependencies:
ARCH=$( /bin/arch ) subscription-manager repos --enable "codeready-builder-for-rhel-8-
↳${ARCH}-rpms"
```

(次のページに続く)

(前のページからの続き)

```
# Disable the built-in PostgreSQL module:
sudo dnf -qy module disable postgresql
```

パッケージをインストールする

```
sudo dnf -y install edb-jdbc
```

Installing EDB JDBC Connector on AlmaLinux 8 or Rocky Linux 8 x86_64

前提条件

インストールプロセスを開始する前に：

- 接続文字列を使用して接続できるホストに EDB Postgres Advanced Server をインストールします。同じホスト上にある必要はありません。

Installing EDB Postgres Advanced Server を参照してください。

- Supported JDBC distributions を確認します。
- リポジトリをセットアップする

リポジトリの設定は 1 回限りのタスクです。リポジトリを既に設定している場合、この手順を実行する必要はありません。

リポジトリをセットアップするには、

EDB repositories に移動し、そこに表示される指示に従います。

- その他の前提条件に対処する

```
# Install the EPEL repository:
sudo dnf -y install epel-release
# Enable additional repositories to resolve dependencies:
sudo dnf config-manager --set-enabled PowerTools
# Disable the built-in PostgreSQL module:
sudo dnf -qy module disable postgresql
```

パッケージをインストールする

```
sudo dnf -y install edb-jdbc
```

Installing EDB JDBC Connector on RHEL 7 or OL 7 x86_64

前提条件

インストールプロセスを開始する前に：

- 接続文字列を使用して接続できるホストに EDB Postgres Advanced Server をインストールします。同じホスト上にある必要はありません。

Installing EDB Postgres Advanced Server を参照してください。

- Supported JDBC distributions を確認します。
- リポジトリをセットアップする

リポジトリの設定は 1 回限りのタスクです。リポジトリを既に設定している場合、この手順を実行する必要はありません。

リポジトリをセットアップするには、

EDB repositories に移動し、そこに表示される指示に従います。

- その他の前提条件に対処する

```
# Install the EPEL repository:
sudo yum -y install https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.
↳noarch.rpm
# Enable additional repositories to resolve dependencies:
subscription-manager repos --enable "rhel-*-optional-rpms" --enable "rhel-*-extras-
↳rpms" --enable "rhel-ha-for-rhel-*-server-rpms"
```

パッケージをインストールする

```
sudo yum -y install edb-jdbc
```

Installing EDB JDBC Connector on CentOS 7 x86_64

前提条件

インストールプロセスを開始する前に：

- 接続文字列を使用して接続できるホストに EDB Postgres Advanced Server をインストールします。同じホスト上にある必要はありません。

Installing EDB Postgres Advanced Server を参照してください。

- Supported JDBC distributions を確認します。
- リポジトリをセットアップする

リポジトリの設定は 1 回限りのタスクです。リポジトリを既に設定している場合、この手順を実行する必要はありません。

リポジトリをセットアップするには、

EDB repositories に移動し、そこに表示される指示に従います。

- その他の前提条件に対処する

```
# Install the EPEL repository:
sudo yum -y install https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.
↪noarch.rpm
```

パッケージをインストールする

```
sudo yum -y install edb-jdbc
```

Installing EDB JDBC Connector on SLES 15 x86_64

前提条件

インストールプロセスを開始する前に：

- 接続文字列を使用して接続できるホストに EDB Postgres Advanced Server をインストールします。同じホスト上にある必要はありません。

Installing EDB Postgres Advanced Server を参照してください。

- Supported JDBC distributions を確認します。
- リポジトリをセットアップする

リポジトリの設定は 1 回限りのタスクです。リポジトリを既に設定している場合、この手順を実行する必要はありません。

リポジトリをセットアップするには、

EDB repositories に移動し、そこに表示される指示に従います。

- その他の前提条件に対処する

```
# Activate the required SUSE module
sudo SUSEConnect -p PackageHub/15.3/x86_64

# Refresh the metadata
sudo zypper refresh
```

パッケージをインストールする

```
sudo zypper -n install edb-jdbc
```

初期構成

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 バージョンに対応した適切な `java .jar` ファイルを `$java_home/jre/lib/ext` ディレクトリにコピーするか、`.jar` ファイルの場所を `CLASSPATH` 環境変数に追加する必要があります。

`jar` ファイルの場所を `CLASSPATH` 環境変数に追加する場合は、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-jdbc18.jar
```

Installing EDB JDBC Connector on SLES 12 x86_64

前提条件

インストールプロセスを開始する前に：

- 接続文字列を使用して接続できるホストに EDB Postgres Advanced Server をインストールします。同じホスト上にある必要はありません。

Installing EDB Postgres Advanced Server を参照してください。

- Supported JDBC distributions を確認します。
- リポジトリをセットアップする

リポジトリの設定は 1 回限りのタスクです。リポジトリを既に設定している場合、この手順を実行する必要はありません。

リポジトリをセットアップするには、

[EDB repositories](#) に移動し、そこに表示される指示に従います。

- その他の前提条件に対処する

```
# Activate the required SUSE module
sudo SUSEConnect -p PackageHub/12.5/x86_64
sudo SUSEConnect -p sle-sdk/12.5/x86_64

# Refresh the metadata
sudo zypper refresh
```

パッケージをインストールする

```
sudo zypper -n install edb-jdbc
```

初期構成

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 バージョンに対応した適切な `java .jar` ファイルを `$java_home/jre/lib/ext` ディレクトリにコピーするか、`.jar` ファイルの場所を `CLASSPATH` 環境変数に追加する必要があります。

`jar` ファイルの場所を `CLASSPATH` 環境変数に追加する場合は、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-jdbc18.jar
```

Installing EDB JDBC Connector on Ubuntu 20.04 x86_64

前提条件

インストールプロセスを開始する前に：

- 接続文字列を使用して接続できるホストに EDB Postgres Advanced Server をインストールします。同じホスト上にある必要はありません。

Installing EDB Postgres Advanced Server を参照してください。

- Supported JDBC distributions を確認します。
- リポジトリをセットアップする

リポジトリの設定は 1 回限りのタスクです。リポジトリを既に設定している場合、この手順を実行する必要はありません。

リポジトリをセットアップするには、

[EDB repositories](#) に移動し、そこに表示される指示に従います。

パッケージをインストールする

```
sudo apt-get -y install edb-jdbc
```

Installing EDB JDBC Connector on Ubuntu 18.04 x86_64

前提条件

インストールプロセスを開始する前に：

- 接続文字列を使用して接続できるホストに EDB Postgres Advanced Server をインストールします。同じホスト上にある必要はありません。

Installing EDB Postgres Advanced Server を参照してください。

- Supported JDBC distributions を確認します。
- リポジトリをセットアップする

リポジトリの設定は 1 回限りのタスクです。リポジトリを既に設定している場合、この手順を実行する必要はありません。

リポジトリをセットアップするには、

EDB repositories に移動し、そこに表示される指示に従います。

パッケージをインストールする

```
sudo apt-get -y install edb-jdbc
```

Installing EDB JDBC Connector on Debian 10 x86_64

前提条件

インストールプロセスを開始する前に：

- 接続文字列を使用して接続できるホストに EDB Postgres Advanced Server をインストールします。同じホスト上にある必要はありません。

Installing EDB Postgres Advanced Server を参照してください。

- Supported JDBC distributions を確認します。
- リポジトリをセットアップする

リポジトリの設定は 1 回限りのタスクです。リポジトリを既に設定している場合、この手順を実行する必要はありません。

リポジトリをセットアップするには、

EDB repositories に移動し、そこに表示される指示に従います。

パッケージをインストールする

```
sudo apt-get -y install edb-jdbc
```

4.1.2 Installing EDB JDBC Connector on IBM Power (ppc64le)

オペレーティングシステム固有のインストール手順については、次を参照してください。

- *Installing EDB JDBC Connector on RHEL 8 ppc64le*
- *Installing EDB JDBC Connector on SLES 15 ppc64le*
- *Installing EDB JDBC Connector on SLES 12 ppc64le*

Installing EDB JDBC Connector on RHEL 8 ppc64le

前提条件

インストールプロセスを開始する前に：

- 接続文字列を使用して接続できるホストに EDB Postgres Advanced Server をインストールします。同じホスト上にある必要はありません。

Installing EDB Postgres Advanced Server を参照してください。

- Supported JDBC distributions を確認します。
- リポジトリをセットアップする

リポジトリの設定は 1 回限りのタスクです。リポジトリを既に設定している場合、この手順を実行する必要はありません。

リポジトリをセットアップするには、

EDB repositories に移動し、そこに表示される指示に従います。

- その他の前提条件に対処する

```
# Install the EPEL repository:
sudo dnf -y install https://dl.fedoraproject.org/pub/epel/epel-release-latest-8.
→noarch.rpm

# Refresh the cache:
sudo dnf makecache

# Enable additional repositories to resolve dependencies:
ARCH=$( /bin/arch ) subscription-manager repos --enable "codeready-builder-for-rhel-8-
→${ARCH}-rpms"
```

(次のページに続く)

(前のページからの続き)

```
# Disable the built-in PostgreSQL module:
sudo dnf -qy module disable postgresql
```

パッケージをインストールする

```
sudo dnf -y install edb-jdbc
```

Installing EDB JDBC Connector on SLES 15 ppc64le

前提条件

インストールプロセスを開始する前に：

- 接続文字列を使用して接続できるホストに EDB Postgres Advanced Server をインストールします。同じホスト上にある必要はありません。

Installing EDB Postgres Advanced Server を参照してください。

- Supported JDBC distributions を確認します。
- リポジトリをセットアップする

リポジトリの設定は 1 回限りのタスクです。リポジトリを既に設定している場合、この手順を実行する必要はありません。

リポジトリをセットアップするには、

EDB repositories に移動し、そこに表示される指示に従います。

- その他の前提条件に対処する

```
# Activate the required SUSE module
sudo SUSEConnect -p PackageHub/15.3/ppc64le

# Refresh the metadata
sudo zypper refresh
```

パッケージをインストールする

```
sudo zypper -n install edb-jdbc
```

初期構成

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 バージョンに対応した適切な `java .jar` ファイルを `$java_home/jre/lib/ext` ディレクトリにコピーするか、`.jar` ファイルの場所を `CLASSPATH` 環境変数に追加する必要があります。

`jar` ファイルの場所を `CLASSPATH` 環境変数に追加する場合は、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-jdbc18.jar
```

Installing EDB JDBC Connector on SLES 12 ppc64le

前提条件

インストールプロセスを開始する前に：

- 接続文字列を使用して接続できるホストに EDB Postgres Advanced Server をインストールします。同じホスト上にある必要はありません。

Installing EDB Postgres Advanced Server を参照してください。

- Supported JDBC distributions を確認します。
- リポジトリをセットアップする

リポジトリの設定は 1 回限りのタスクです。リポジトリを既に設定している場合、この手順を実行する必要はありません。

リポジトリをセットアップするには、

[EDB repositories](#) に移動し、そこに表示される指示に従います。

- その他の前提条件に対処する

```
# Activate the required SUSE module
sudo SUSEConnect -p PackageHub/12.5/ppc64le
sudo SUSEConnect -p sle-sdk/12.5/ppc64le

# Refresh the metadata
sudo zypper refresh
```

パッケージをインストールする

```
sudo zypper -n install edb-jdbc
```

初期構成

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 バージョンに対応した適切な `java .jar` ファイルを `$java_home/jre/lib/ext` ディレクトリにコピーするか、`.jar` ファイルの場所を `CLASSPATH` 環境変数に追加する必要があります。

`jar` ファイルの場所を `CLASSPATH` 環境変数に追加する場合は、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-jdbc18.jar`

4.2 Using the graphical installer to install the connector

EDB Connectors Installation ウィザードを使用して、JDBC コネクタをシステムに追加できます。ウィザードは

[EDB website](#) で入手できます。

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。Installation Wizard を開くには、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、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。

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。

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。

インストールの準備完了ダイアログ

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は、インストールウィザードの進行状況を確認します。

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を完了したことを通知したら、**Finish** ボタンをクリックしてダイアログを終了します。

StackBuilder Plus を使用して、既存の Advanced Server インストールにコネクタを追加または更新することもできます。StackBuilder Plus を開くには、Windows の Apps メニューから StackBuilder Plus を選択します。

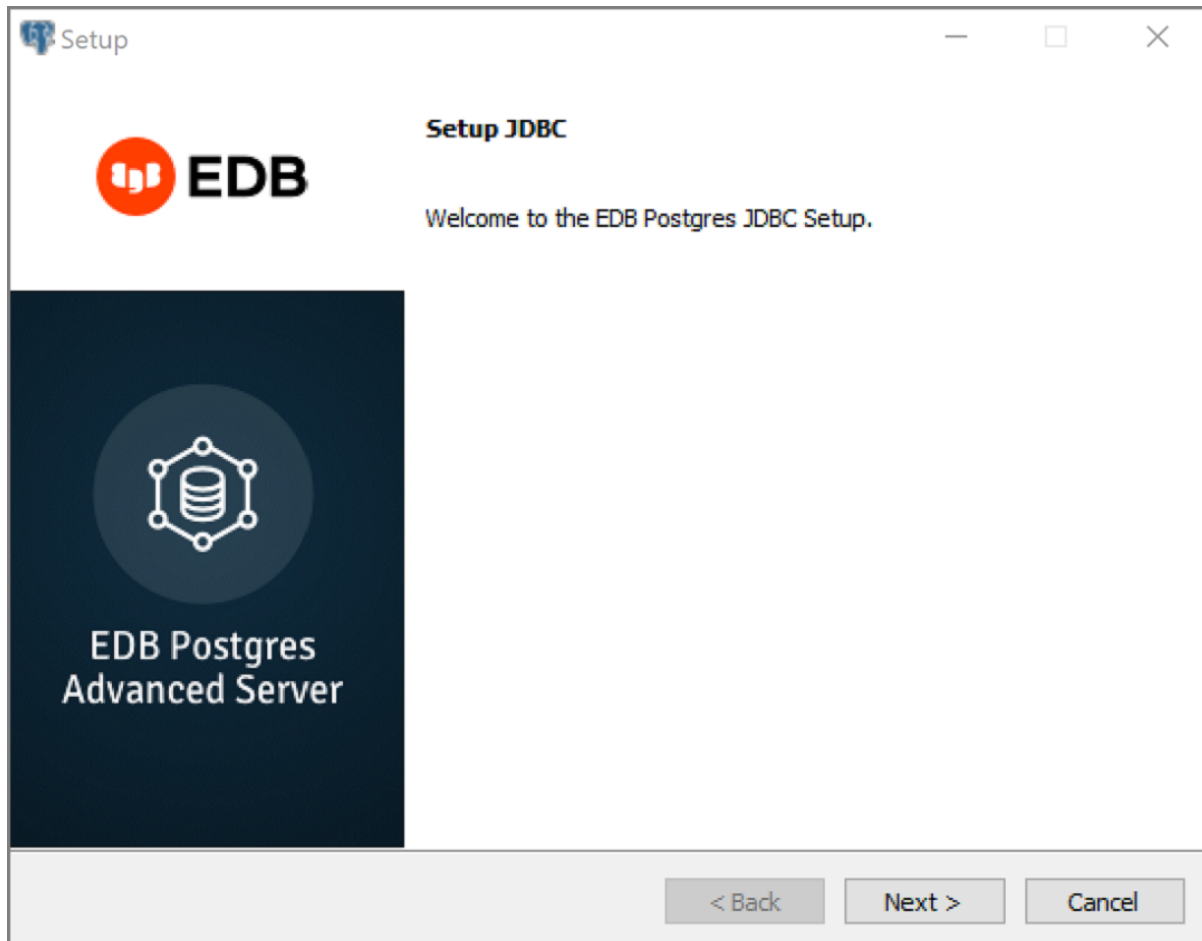


図 1: The JDBC Connector Installation wizard

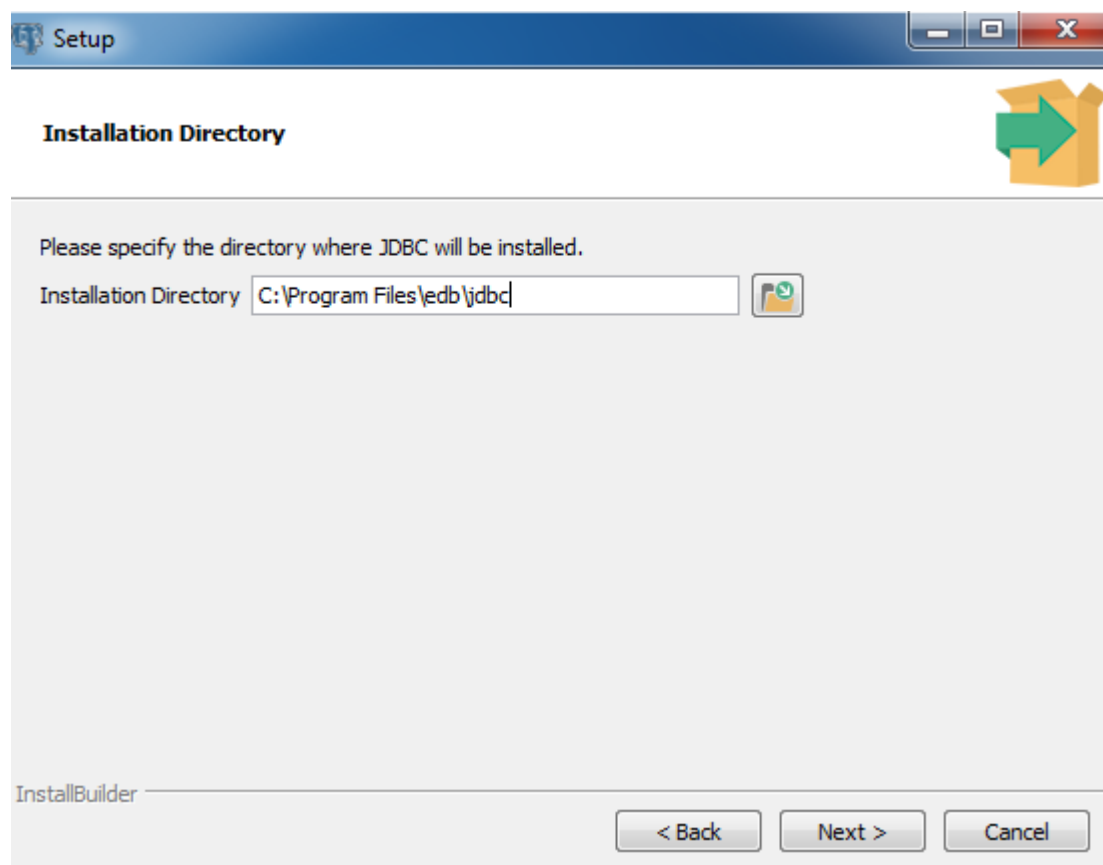


図 2: The Installation dialog

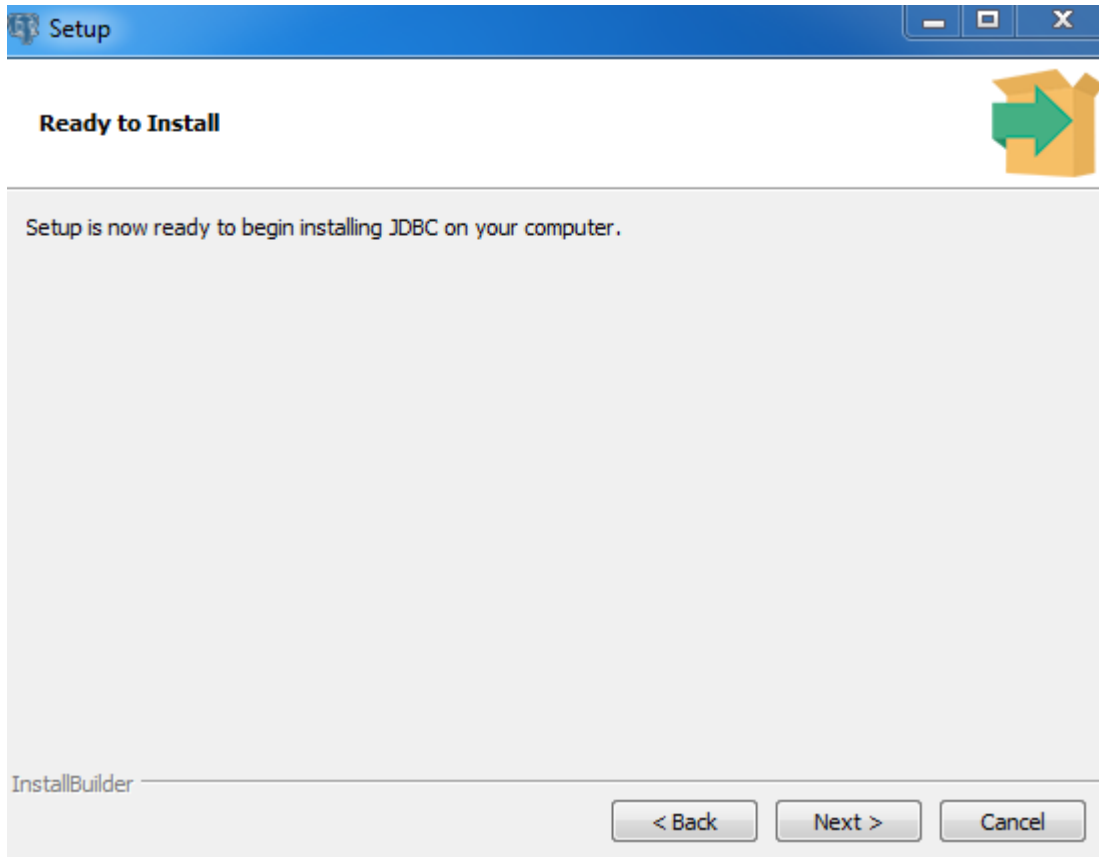


図 3: The Ready to Install dialog

StackBuilder Plus の起動

StackBuilder Plus 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。

Connectors インストーラーの選択

画面上のウィザードの指示に従って、EDB Connectors のインストールを追加/更新します。

4.3 Configuring the EDB Postgres Advanced Server JDBC Connector

edb-jdbc18.jar は JDBC バージョン 4.2 をサポートしています。

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 バージョンに対応した適切な java .jar ファイルを \$java_home/jre/lib/ext ディレクトリにコピーするか、.jar ファイルの場所を CLASSPATH 環境変数に追加する必要があります。

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ります。

/usr/edb/jdbc/edb-jdbc18.jar

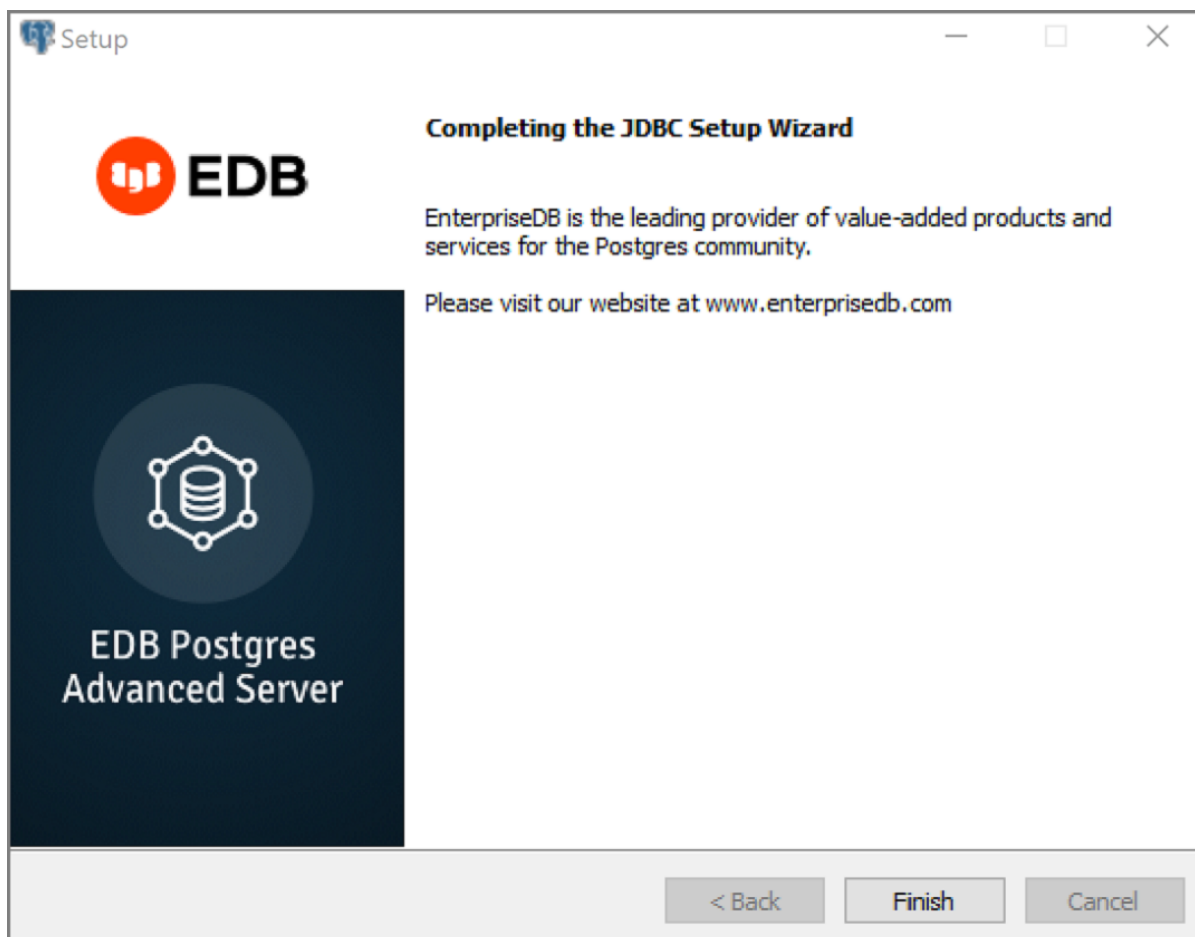


図 4: The installation is complete

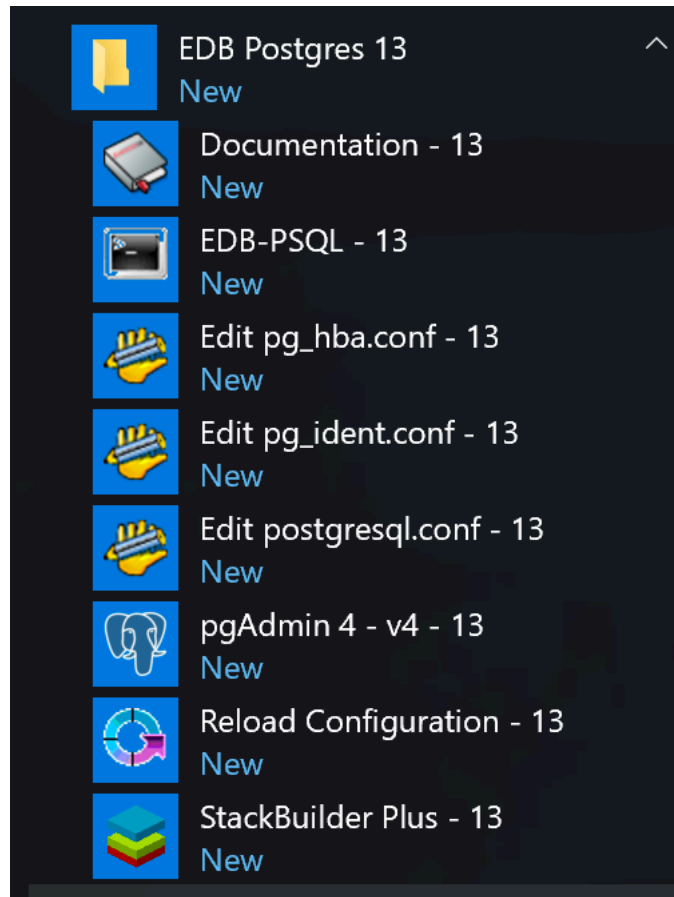


図 5: Starting StackBuilder Plus

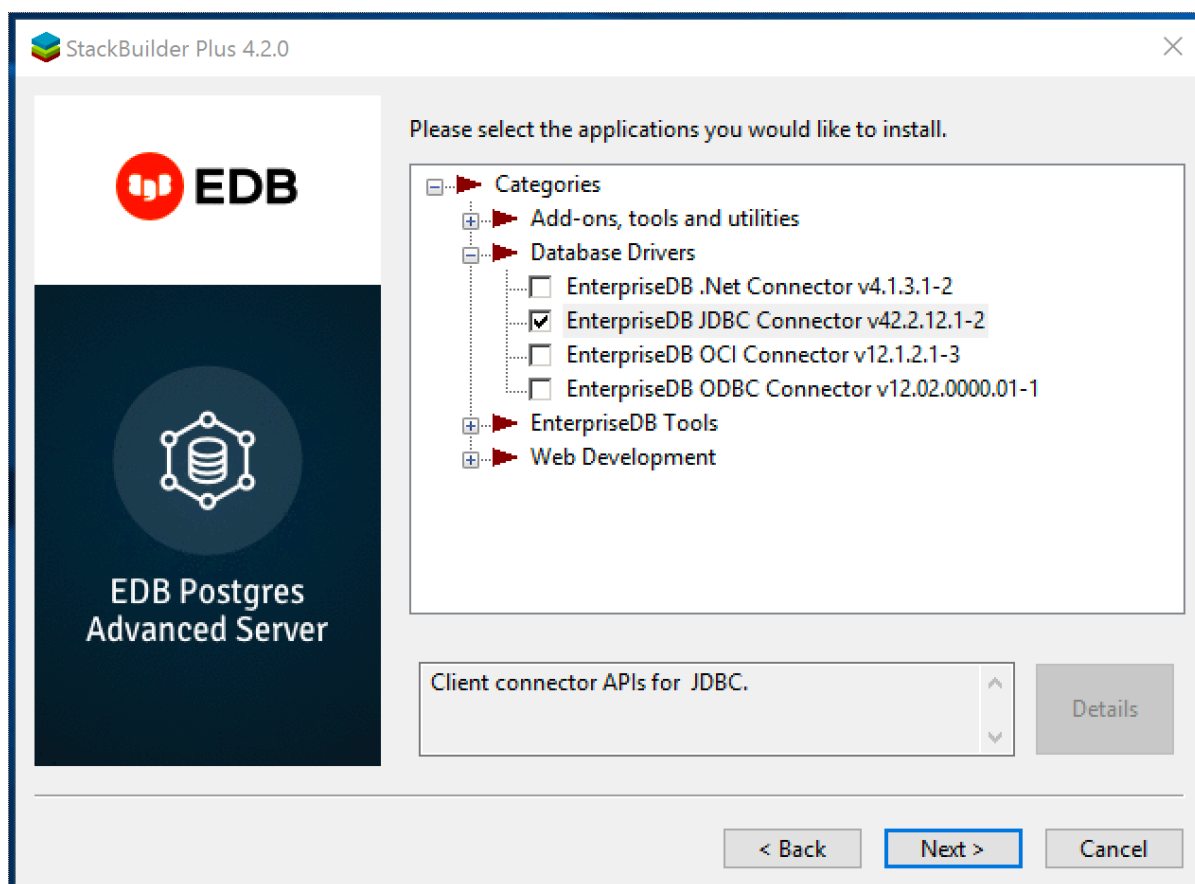


図 6: Selecting the Connectors installer

4.4 Upgrading a Linux installation

既存の JDBC Connector RPM インストールがある場合は、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 CentOS 7 の場合：

```
yum upgrade edb-repo
```

- RHEL または Rocky Linux または AlmaLinux 8 の場合：

```
dnf upgrade edb-repo
```

yum または dnf は edb.repo ファイルを更新して、edb.repo ファイルで指定された資格情報で接続するように構成された現在の EDB リポジトリへのアクセスを有効にします。次に、yum または dnf を使用して、インストールされているパッケージをアップグレードできます。

- RHEL または CentOS 7 の場合：

```
yum upgrade edb-jdbc
```

- RHEL または Rocky Linux または AlmaLinux 8 の場合：

```
dnf upgrade edb-jdbc
```


第5章 Using the EDB Postgres Advanced Server JDBC Connector with Java applications

Java と EDB JDBC Connector を配置すると、Java アプリケーションは EDB Postgres Advanced Server データベースにアクセスできます。この例では、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。

```
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();
            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());
        }
    }
}
```

(次のページに続く)

(前のページからの続き)

```
        System.out.println("Vendor Error:  " + exp.getErrorCode());
    }
}
}
```

この例は簡単ですが、Java アプリケーションから EDB Postgres Advanced Server データベースを操作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します。
- 接続プロパティを構築します。
- データベースサーバーに接続します。
- SQL を実行します。
- 結果セットを処理します。
- 片付けます。
- 発生したエラーを処理します。

Loading_the_advanced_server_jdbc_connector connection_to_the_database unning_sql_statements_through_statement_objects
retrving_results_from_a_resultset_object freeing_resources Handling_errors

5.1 Loading the EDB Postgres Advanced Server JDBC Connector

EDB Postgres Advanced Server JDBC ドライバーは Java で記述され、コンパイルされた Java Archive (JAR) ファイルの形式で配布されます。Class.forName() メソッドを使用してドライバーをロードします。forName() メソッドは、実行時に Java クラスを動的にロードします。アプリケーションが forName() メソッドを呼び出すと、Java 仮想マシン (JVM) は要求されたクラスを実装するコンパイル済みフォーム (バイトコード) を見つけようとします。

EDB Postgres Advanced Server JDBC 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ロードした後、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 ドライバーを管理する別の JDBC クラス (DriverManager という名前) に自分自身を登録します。

JVM が指定されたドライバーを見つけれられない場合、ClassNotFoundException 例外を報告します (プログラムの最後近くにある catch ブロックでインターセプトされます)。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。

5.2 Connecting to the database

ドライバーがロードされ、自分自身を `DriverManager` に登録した後、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);
```

すべての JDBC 接続は `DriverManager` で始まります。`DriverManager` クラスは、データベースへの接続を作成する `getConnection()` と呼ばれる静的メソッドを提供します。`getConnection()` メソッドを呼び出すと、`DriverManager` はデータベースへの接続に使用する JDBC ドライバーを決定する必要があります。この決定は、`getConnection()` に渡す URL に基づいています。

JDBC URL は次の一般的な形式をとります。

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

JDBC URL の最初のコンポーネントは常に `jdbc` です。EDB Postgres Advanced Server JDBC コネクタを使用する場合、2 番目のコンポーネント（ドライバー）は `edb` です。

Advanced Server JDBC URL は次のいずれかの形式です。

```
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。

Name	Description
host	サーバーのホスト名。デフォルトは localhost です。
port	サーバーがリスンしているポート番号。デフォルトは EDB Postgres Advanced Server の標準ポート番号（5444）です。
database	データベース名。

`additional_connection_properties` `preferring_synchronous_secondary_database_servers`

5.2.1 Additional connection properties

標準の接続パラメーターに加えて、EDB Postgres Advanced Server JDBC ドライバーは、EDB に固有の動作を制御する接続プロパティをサポートしています。これらのプロパティは、接続 URL で、または `DriverManager.getConnection()` に渡される `Properties` オブジェクトパラメーターとして指定できます。この例は、`Properties` オブジェクトを使用して追加の接続プロパティを指定する方法を示しています。

```
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: デフォルトでは、`SSL=true` と接続 URL パラメーター `sslfactory=org.postgresql.ssl.NonValidatingFactory` を設定すると、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` の作成の詳細については、[the PostgreSQL JDBC driver documentation](#) を参照してください。

URL で追加の接続プロパティを指定するには、疑問符とアンパサンドで区切られたキーワードと値のペアの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。

Name	Type	Description
user	String	接続が作成されるデータベースユーザー。
password	String	データベースユーザーのパスワード。
ssl	Boolean	認証された暗号化された SSL 接続を要求します。
charSet	String	<code>charSet</code> 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	Integer	<code>prepareThreshold</code> の値は、サーバーサイドの準備されたステートメントに切り替える前に必要な <code>PreparedStatement</code> の実行数を決定します。デフォルトは 5 です。
loadBalanceHosts	Boolean	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。有効にすると、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。

次のページに続く

表 2 – 前のページからの続き

Name	Type	Description
targetServerType	String	必要な状態のサーバーのみへの接続を開くことを許可します。許可される値は、 <i>any</i> 、 <i>primary</i> 、 <i>secondary</i> 、 <i>preferSecondary</i> 、および <i>preferSyncSecondary</i> です。プライマリ/セカンダリの区別は、現在、サーバーが書き込みを許可するかどうかを監視することによって行われます。値 <i>preferSecondary</i> は、使用可能な場合はセカンダリへの接続を試み、それ以外の場合はプライマリへの接続を許可します。EDB Postgres Advanced Server JDBC コネクタは <i>preferSyncSecondary</i> をサポートしています。これは、同期セカンダリまたはアクティブな同期セカンダリがない場合はプライマリへの接続を許可します。
skipQuotesOnReturning	Boolean	<i>true</i> に設定すると、 <i>RETURNING</i> 句の列名は引用符で囲まれません。これにより、列名の大文字と小文字を区別する比較がなくなります。 <i>false</i> （デフォルト設定）に設定すると、列名が引用符で囲まれます。
changeServerName	Boolean	<i>PgConnection.java</i> の <i>getServerName()</i> 呼び出しは <i>EnterpriseDB</i> を返します。 <i>changeServerName</i> が <i>true</i> に設定されている場合、値を <i>PostgreSQL</i> として返します。デフォルト値は <i>false</i> です。
forceBinaryTransfer	Boolean	値が <i>true</i> に設定されている場合、すべてのバイナリ型を PostgreSQL サーバーから JDBC ドライバーにバイナリ形式で転送します。デフォルト値は <i>false</i> です。

5.2.2 Preferring synchronous secondary database servers

EDB Postgres Advanced Server JDBC コネクタは、`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。

`preferSyncSecondary` オプションは、フェイルオーバー接続に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://primary:port,secondary_1:port_1,secondary_2:port_2,.../
database?targetServerType=preferSyncSecondary
```

パラメーター

primary:port

プライマリデータベースサーバに割り当てられた IP アドレスまたは名前とそれに続くポート番号。primary 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 アドレスを使用して指定する必要があります。

注釈: リスト内の任意の場所でプライマリデータベースサーバーを指定できます。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、セカンダリデータベースサーバに割り当てられた IP アドレスまたは名前とそれに続くポート番号。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 アドレスを使用して指定する必要があります。

database

接続するデータベースの名前。

次に、接続 URL の例を示します。

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

preferSyncSecondary オプションには、次の特性が適用されます。

- 接続リストの任意の場所にプライマリデータベースサーバーを指定します。
- Java プログラムが使用するデータベースにアクセスするための接続は、最初に同期セカンダリで試行されます。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。
- 接続が試行される順序は、loadBalanceHosts 接続プロパティによって決まります。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序で接続が試行されます。有効にすると、接続がランダムに試行されます。
- 同期セカンダリに接続できない場合、プライマリデータベースサーバーへの接続が使用されます。プライマリデータベースサーバーがアクティブでない場合、接続は失敗します。

preferSyncSecondary オプションに使用する同期セカンダリは、ホットスタンバイを使用するように構成する必要があります。

プライマリ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構成の概要

プライマリ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。

`preferSyncSecondary` オプションに必要なホットスタンバイの使用に関する一般情報については、

[the PostgreSQL core documentation](#) を参照してください。

プライマリからセカンダリデータベースサーバ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について説明します) またはセクション 25.3.3, [セクション 25.3 の低レベル API](#) [The PostgreSQL Core Documentation](#) の連続アーカイブとポイントインタイムリカバリ (PITR) *。

ホットスタンバイを使用するために設定する構成パラメーターについては、

[Section 19.6, Replication](#) を参照してください。

例：プライマリ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

次の例では、

- プライマリ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 にあります。
- セカンダリデータベースサーバは `secondary2` という名前で、ホスト 192.162.2.24、ポート 5446 (プライマリと同じホスト) にあります。

プライマリ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` は、192.168.2.22 上の `secondary1` データベースサーバーと、プライマリに対してローカルな `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。

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

`postgresql.conf` ファイルとその `pg_hba.conf` ファイルでプライマリデータベースサーバを構成した後、ホスト 192.168.2.22 で `secondary1` の次のコマンドを呼び出すことにより、データベースサーバ `secondary1` が作成されます。

```
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、データベースクラスターに `recovery.conf` ファイルが生成されます。次の例では、`primary_conninfo` 文字列の一部として `application_name=secondary1` 設定を追加し、`pg_basebackup` によって自動的に生成された他の不要なオプションのいくつかを削除します。`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、プライマリ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 (`secondary2`) は、プライマリが使用するのと同じホスト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
- -----
0/60000028
(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
- -----
0/60000130
(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。`application_name=secondary2` 設定は、次の例に示すように、`primary_conninfo` 文字列の一部です。`standby_mode = 'on'` パラメーターも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、プライマリ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: EDB Postgres Advanced Server v12 以降、`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した。その結果、*primary_conninfo* は *recovery.conf* ファイルから *postgresql.conf* ファイルに移動されます。 *standby.signal* ファイルが存在すると、クラスターがスタンバイモードで実行されます。 *recovery.conf* ファイルを手動で作成して *data* ディレクトリの下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報告します。

パラメーター *standby_mode=on* も EDB Postgres Advanced Server v12 から削除され、 *trigger_file* パラメーター名が *promote_trigger_file* に変更されました。

次の表に、セカンダリデータベースサーバーと比較したプライマリ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リストします。

Parameter	Primary	Secondary	Description
<i>archive_mode</i>	on	off	アーカイブストレージに送信される完了した WAL セグメント
<i>archive_command</i>	cp %p /archive_dir/%f	n/a	完了した WAL セグメントをアーカイブする

<i>wal_level</i> (10 or later)	replica	minimal	Information written to WAL segment
<i>wal_keep_segments</i>	<i>n</i> (positive integer)	0	Minimum number of past log segments to keep for standby servers
<i>syn-chronous_standby_names</i>	<i>n</i> *(<i>*secondary1</i> , <i>secondary2</i> ,...)	n/a	List of standby servers for synchronous replication. Must be present to enable synchronous replication. These are obtained from the <i>application_name</i> option of the <i>primary_conninfo</i> parameter in the <i>recovery.conf</i> file of each standby server.
<i>hot_standby</i>	off	on	Client application can connect and run queries on the secondary server in standby mode

セカンダリデータベースサーバ (*secondary1*) が起動されます。

```
- 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 (*secondary2*) が起動されます。

```
- 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設定されていることを確認するには、プライマリ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 *sync* としてリストされています。

```
edb=# SELECT username, application_name, client_addr, client_port, sync_state FROM pg_
↳stat_replication;
__OUTPUT__
  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://primary: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ています。

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

この例では、優先される同期セカンダリ接続オプションは、次のプログラムに示すように、secondary1、次に secondary1 がアクティブでない場合は secondary2、そして secondary1 と secondary2 の両方がアクティブでない場合はプライマリで試行されます。プログラムは、接続されたデータベースサーバの IP アドレスとポートを表示します。

```
import java.sql.*;
public class InetServer
{
    public static void main(String[] args)
    {
        try
        {
            Class.forName("com.edb.Driver");
            String url =
"jdbc:edb://primary: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()");
            rs.next();
            System.out.println(rs.getString(1));
```

(次のページに続く)

(前のページからの続き)

```

        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 で 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 で 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 もシャットダウンすると、192.168.2.24 ポート 5444 で primary 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 Executing SQL statements through statement objects

EDB Postgres Advanced Server JDBC コネクタドライバをロードして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 オブジェクトをビルドし、SQL クエリを実行して結果を表示します。

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。それぞれは、特定の種類の SQL ステートメントを送信するように特化されています。

- Statement オブジェクトを使用して、パラメーターなしの単純な SQL ステートメントを実行します。
- 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への呼び出しを実行するために使用されます。

SQL ステートメントを実行する前に、Statement オブジェクトを構築する必要があります。Statement オブジェクトは、SQL ステートメントをサーバーに送信する（および結果セットにアクセスする）方法を提供します。各 Statement オブジェクトは、Connection に属します。createStatement() メソッドを使用して、Connection に Statement オブジェクトを作成するように依頼します。

Statement オブジェクトは、さまざまな種類の 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");
```

executeQuery() メソッドは単一の引数、つまり実行する SQL ステートメントを想定しています。executeQuery() は、クエリからのデータを ResultSet オブジェクトに返します。提供された SQL ステートメントの実行中にサーバーでエラーが発生した場合、サーバーは SQLException を返しますが、ResultSet を返します。

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

JDBC コネクタ（EDB Postgres Advanced Server バージョン 10 以降）は、CallableStatement オブジェクトをインスタンス化するときの名前付けパラメーターの使用をサポートしています。この構文は、JDBC でサポートされている構文の拡張であり、JDBC 標準に準拠していません。

CallableStatement オブジェクトを使用して、パラメーター値をストアードプロシージャに渡すことができます。CallableStatement オブジェクトを使用して、IN、OUT、および INOUT パラメーターに値を割り当てることができます。

CallableStatement クラスを使用する場合、序数表記または名前付け表記を使用して実引数の値を指定できます。ステートメント内の IN または INOUT パラメーターマーカーごとに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します。たとえば、STRING 値を渡す場合は、setString setter メソッドを使用します。

ステートメント (?) の各パラメーターマーカーは、序数値を表します。序数パラメーターを使用する場合、プロシージャ定義で仮引数が指定された順序で実際のパラメーター値をステートメントに渡します。

CallableStatement オブジェクトの引数値を指定するときに、名前付け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し、受け入れ可能なデフォルト値を持つパラメーターを省略できます。また、名前付けパラメーターは任意の順序で指定できます。

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 に対応している必要があります。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、=> トークンを使用します。

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ーを提供する方法を示しています。

```
CREATE OR REPLACE PROCEDURE hire_emp (ename VARCHAR2
empno NUMBER,
job VARCHAR2,
sal NUMBER,
hiredate DATE DEFAULT now(),
mgr NUMBER DEFAULT 7100,
deptno NUMBER
)
IS
BEGIN
    INSERT INTO emp VALUES (empno, ename, job, mgr, hiredate, sal, deptno);
END;
```

次の例では、序数表記を使用してパラメーターを提供します。

```
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, Date.valueOf("2016-06-01"));
cstmt.setInt(6, 7566);
cstmt.setInt(7, 30);
```

次の例では、名前付け表記を使用してパラメーターを提供します。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たりできます。

```
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 Retrieving results from a ResultSet object

ResultSet オブジェクトは、SQL ステートメントによって返されるデータの主要なストレージメカニズムです。各 ResultSet オブジェクトには、データとメタデータの両方が **ResultSetMetaData** オブジェクトの形式で含まれています。ResultSetMetaData には、SQL コマンドが返す結果に関する有用な情報（列名、列数、行数、列長など）が含まれています。

ResultSet オブジェクトに格納されている行データにアクセスするには、アプリケーションは 1 つ以上の **getter** メソッドを呼び出します。getter メソッドは、現在の行の特定の列の値を取得します。多くの異なる getter メソッドがあります。各メソッドは特定の型の値を返します。たとえば、**getString()** メソッドは **STRING** 型を返し、**getDate()** メソッドは **Date** を返し、**getInt()** メソッドは **INT** 型を返します。アプリケーションが getter メソッドを呼び出すと、JDBC は値を要求された型に変換しようとします。

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 を返す場合、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（または最初）の行に進むには、ResultSet' s **next()** メソッドを呼び出します。ResultSet.next() はブールメソッドです。ResultSet に別の行がある場合は **TRUE** を返し、最後の行を過ぎて移動した場合は **FALSE** を返します。

サンプルアプリケーションは、ポインターを 1 行目に移動した後、**getString()** getter メソッドを使用して 1 列目の値を取得し、その値を出力します。ListEmployees は rs.next() と rs.getString() をループで呼び出すため、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 インターフェイスを使用する場合：

- 値を読み取る前に `next()` を呼び出す必要があります。 `next()` は、別の行が使用可能な場合に `true` を返し、行を処理のために準備します。
- 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。このルールに従うのが最も安全ですが、現在、EDB Postgres Advanced Server JDBC ドライバーではフィールドに何度でもアクセスできます。
- `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 Freeing resources

すべての JDBC オブジェクトは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が含まれる場合があります。通常、アプリケーションで必要なくなったときにこ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た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と、そのオブジェクトはエラーを返します。

5.6 Handling errors

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 EDB Postgres Advanced Server JDBC Connector は両方とも、さまざまな種類のエラー処理を提供します。

ListEmployees class example 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。

JDBC オブジェクト（`SQLException` 型または `SQLException` から派生した型のオブジェクト）がエラーを返すと、`SQLException` オブジェクトは 3 つの異なるエラー情報を公開します。

- 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、エラーが発生した場合、コードは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章 Using advanced queueing

EDB Postgres Advanced Server アドバンスドキューイングは、EDB Postgres Advanced Server データベースのメッセージキューとメッセージ処理を提供します。ユーザー定義メッセージはキューに格納され、キューのコレクションはキュー表に格納されます。キュー表に依存するキューを作成する前に、キュー表を作成する必要があります。

サーバー側では、DBMS_AQADM パッケージのプロシージャによってメッセージキューとキューテーブルが作成および管理されます。DBMS_AQ パッケージを使用して、キューにメッセージを追加または削除したり、PL/SQL コールバックプロシージャを登録または登録解除します。DBMS_AQ および DBMS_AQADM の詳細については、EDB Postgres Advanced Server documentation を参照してください。

クライアント側では、アプリケーションは EDB-JDBC ドライバーの JMS API を使用してメッセージをエンキューおよびデキューします。

6.1 メッセージのエンキューまたはデキュー

EDB Postgres Advanced Server の高度なキューイング機能の使用の詳細については、Database Compatibility for Oracle Developers Built-in Package Guide を参照してください。

6.2 サーバーサイドのセットアップ

JMS ベースの Java アプリケーションでアドバンスドキュー機能を使用するには、最初にユーザー定義型、キューテーブル、およびキューを作成します。次に、データベースサーバーでキューを開始します。Java アプリケーションでは、EDB-PSQL または EDB-JDBC JMS API を使用できます。

6.2.1 EDB-PSQL の使用

EDB-PSQL を呼び出し、EDB Postgres Advanced Server ホストデータベースに接続します。コマンドラインで次の SPL コマンドを使用します。

ユーザー定義型の作成

RAW データ型を指定するには、ユーザー定義型を作成します。この例では、mytype という名前のユーザー定義型を作成します。

```
CREATE TYPE mytype AS (code int, project TEXT);
```

キューテーブルを作成する

キュー表は、同じペイロードタイプの複数のキューを保持できます。この例では、MSG_QUEUE_TABLE という名前のテーブルを作成します。

```
EXEC DBMS_AQADM.CREATE_QUEUE_TABLE
    (queue_table => MSG_QUEUE_TABLE,
     queue_payload_type => mytype,
     comment => Message queue table);
END;
```

キューを作成する

この例では、テーブル MSG_QUEUE_TABLE に MSG_QUEUE という名前のキューを作成します。

```
EXEC DBMS_AQADM.CREATE_QUEUE ( queue_name => MSG_QUEUE, queue_table => MSG_QUEUE_
→TABLE, comment => This queue contains pending messages.);
```

キューを開始する

キューが作成されたら、コマンドラインで次の SPL コードを呼び出して、EDB データベースでキューを開始します。

```
EXEC DBMS_AQADM.START_QUEUE(queue_name => MSG_QUEUE);
commit;
``` ### EDB-JDBC JMS API の使用
```

次の JMS API 呼び出しは、EDB-PSQL を使用して実行したのと同じ手順を実行します。

- EDB Postgres Advanced Server データベースに接続します
- ユーザー定義型を作成する
- キューテーブルとキューを作成する
- キューを開始する

```
```java
edbJmsFact = new EDBJmsConnectionFactory("localhost", 5444, "edb", "edb", "edb");

conn = (EDBJmsQueueConnection) edbJmsFact.createQueueConnection();

session = (EDBJmsQueueSession) conn.createQueueSession(true, Session.CLIENT_
→ACKNOWLEDGE);

String sql = "CREATE TYPE mytype AS (code int, project TEXT);";
UDTType udtType = new UDTType(conn.getConn(), sql, "mytype");
Operation operation = new UDTTypeOperation(udtType);
```

(次のページに続く)

(前のページからの続き)

```

operation.execute();

queueTable = session.createQueueTable(conn.getConn(), "MSG_QUEUE_TABLE", "mytype",
    ↪ "Message queue table");

Queue queue1 = new Queue(conn.getConn(), "MSG_QUEUE", "MSG_QUEUE_TABLE", "Message_
    ↪ Queue");
operation = new QueueOperation(queue1);
operation.execute();

queue = (EDBJmsQueue) session.createQueue("MSG_QUEUE");
queue.setEdbQueueTbl(queueTable);

queue.start();

```

6.3 クライアントサイドの例

ユーザー定義型を作成し、続いてキュー表とキューを作成したら、キューを開始します。次に、EDB-JDBC ドライバーの JMS API を使用してメッセージをエンキューまたはデキューできます。

Java プロジェクトを作成し、edb-jdbc インストールディレクトリからそのライブラリに edb-jdbc18.jar を追加します。

作成した種類に対応した Java Bean を作成します。

```

package mypackage;

import java.util.ArrayList;
import com.edb.aq.UDTType;

public class MyType extends UDTType {

    private int code;
    private String project;

    public MyType() {}

    /**
     * @return the code
     */
    public int getCode() {
        return code;
    }
}

```

(次のページに続く)

```

/**
 * @param code the code to set
 */
public void setCode(int code) {
    this.code = code;
}

/**
 * @return the project
 */
public String getProject() {
    return project;
}

/**
 * @param project the project to set
 */
public void setProject(String project) {
    this.project = project;
}

/**
 * Override this method and call getter methods in the same order as in CREATE_
→TYPE statement.
 * CREATE TYPE mytype AS (code int, project TEXT);
 * @return
 */
@Override
public Object[] getParamValues() {
    ArrayList<Object> params = new ArrayList<Object>();
    params.add(getCode());
    params.add(getProject());
    return params.toArray(); //To change body of generated methods, choose Tools_
→| Templates.
}
}

```

6.3.1 メッセージのエンキューとデキュー

メッセージをエンキューおよびデキューするには、次の手順を実行します。

1. JMS 接続ファクトリーを作成し、キュー接続を作成します。
2. キュー セッションを作成します。

```
edbJmsFact = new EDBJmsConnectionFactory("localhost", 5445, "edb", "edb", "edb");

conn = (EDBJmsQueueConnection) edbJmsFact.createQueueConnection();

session = (EDBJmsQueueSession) conn.createQueueSession(true, Session.CLIENT_
→ACKNOWLEDGE);

queue = new EDBJmsQueue("MSG_QUEUE");
```

6.3.2 メッセージをエンキューする

メッセージをエンキューするには:

1. セッションから EDBJmsMessageProducer を作成します。
2. エンキューメッセージを作成します。
3. EDBJmsMessageProducer.send メソッドを呼び出します。

```
messageProducer = (EDBJmsMessageProducer) session.createProducer(queue);

MyType udtType1 = new MyType();
udtType1.setProject("Test Omega");
udtType1.setCode(321);

udtType1.setName("mytype");

messageProducer.send(udtType1);
```

6.3.3 メッセージをデキューする

メッセージをデキューするには、次の手順を実行します。

1. セッションから EDBJmsMessageConsumer を作成します。
2. EDBJmsMessageConsumer.Receive メソッドを呼び出します。

```

messageConsumer = (EDBJmsMessageConsumer) session.createConsumer(queue);

queue.setDequeue_mode(DequeueMode.BROWSE);
queue.setType_name("mytype");

Message message = messageConsumer.receive();

```

6.4 完全なエンキューおよびデキュープログラム

この例は、エンキューとデキューを示しています。EDB-PSQL を使用してユーザー定義型、キューテーブル、およびキューを作成し、キューを起動します。

```

package mypackage;

import com.edb.aq.DequeueMode;
import com.edb.aq.operations.*;
import com.edb.jms.client.EDBJmsQueueConnection;
import com.edb.jms.client.EDBJmsConnectionFactory;
import com.edb.jms.client.EDBJmsMessageConsumer;
import com.edb.jms.client.EDBJmsMessageProducer;
import com.edb.jms.client.EDBJmsQueue;
import com.edb.jms.client.EDBJmsQueueSession;
import com.edb.jms.client.EDBQueueTable;
import java.sql.Connection;
import java.sql.DriverManager;
import javax.jms.JMSEException;
import javax.jms.Message;
import javax.jms.Session;

public class JMSClient {

    public static void main(String args[]) throws JMSEException {

        EDBJmsConnectionFactory edbJmsFact = null;
        EDBJmsQueueConnection conn = null;
        EDBJmsQueueSession session = null;
        EDBQueueTable queueTable = null;
        EDBJmsQueue queue = null;
        EDBJmsMessageProducer messageProducer = null;
        EDBJmsMessageConsumer messageConsumer = null;

        try {

            edbJmsFact = new EDBJmsConnectionFactory("localhost", 5444, "edb", "edb",
↪ "edb");

```

(次のページに続く)

(前のページからの続き)

```

        conn = (EDBJmsQueueConnection) edbJmsFact.createQueueConnection();

        session = (EDBJmsQueueSession) conn.createQueueSession(true, Session.
→CLIENT_ACKNOWLEDGE);

        queue = (EDBJmsQueue) session.createQueue("MSG_QUEUE");

        messageProducer = (EDBJmsMessageProducer) session.createProducer(queue);

        MyType udtType1 = new MyType();
        udtType1.setProject("Test Omega");
        udtType1.setCode(321);

        udtType1.setName("mytype");

        messageProducer.send(udtType1);

        messageConsumer = (EDBJmsMessageConsumer) session.createConsumer(queue);

        queue.setDequeue_mode(DequeueMode.BROWSE);
        queue.setTypeNames("mytype");

        Message message = messageConsumer.receive();
        System.out.println("Received: " + message);

        message = messageConsumer.receive();

        System.out.println("Received: " + message);
    } catch (JMSEException jmsEx) {
        System.out.println(jmsEx.getMessage());
    } finally {
        if(conn != null) {
            conn.close();
        }
    }
}
}

```

この例では、ユーザー定義型、キュー表、およびキューのエンキュー、デキュー、および作成を示しています。また、キューを開始します。

```

package mypackage;

import com.edb.aq.DequeueMode;

```

(次のページに続く)

(前のページからの続き)

```

import com.edb.aq.operations.*;
import com.edb.jms.client.EDBJmsQueueConnection;
import com.edb.jms.client.EDBJmsConnectionFactory;
import com.edb.jms.client.EDBJmsMessageConsumer;
import com.edb.jms.client.EDBJmsMessageProducer;
import com.edb.jms.client.EDBJmsQueue;
import com.edb.jms.client.EDBJmsQueueSession;
import com.edb.jms.client.EDBQueueTable;
import java.sql.Connection;
import java.sql.DriverManager;
import javax.jms.JMSEException;
import javax.jms.Message;
import javax.jms.Session;

public class JMSClient {

    public static void main(String args[]) throws JMSEException {

        EDBJmsConnectionFactory edbJmsFact = null;
        EDBJmsQueueConnection conn = null;
        EDBJmsQueueSession session = null;
        EDBQueueTable queueTable = null;
        EDBJmsQueue queue = null;
        EDBJmsMessageProducer messageProducer = null;
        EDBJmsMessageConsumer messageConsumer = null;

        try {

            edbJmsFact = new EDBJmsConnectionFactory("localhost", 5444, "edb", "edb",
↪ "edb");

            conn = (EDBJmsQueueConnection) edbJmsFact.createQueueConnection();

            session = (EDBJmsQueueSession) conn.createQueueSession(true, Session.
↪ CLIENT_ACKNOWLEDGE);

            String sql = "CREATE TYPE mytype AS (code int, project TEXT);";
            UDTType udtType = new UDTType(conn.getConn(), sql, "mytype");
            Operation operation = new UDTTypeOperation(udtType);
            operation.execute();

            queueTable = session.createQueueTable(conn.getConn(), "MSG_QUEUE_TABLE",
↪ "mytype", "Message queue table");
            Queue queue1 = new Queue(conn.getConn(), "MSG_QUEUE", "MSG_QUEUE_TABLE",
↪ "Message Queue");

```

(次のページに続く)

(前のページからの続き)

```
operation = new QueueOperation(queue1);
operation.execute();
queue = (EDBjmsQueue) session.createQueue("MSG_QUEUE");
queue.setEdbQueueTbl(queueTable);
queue.start();

messageProducer = (EDBjmsMessageProducer) session.createProducer(queue);

MyType udtType1 = new MyType();
udtType1.setProject("Test Omega");
udtType1.setCode(321);

udtType1.setName("mytype");

messageProducer.send(udtType1);

messageConsumer = (EDBjmsMessageConsumer) session.createConsumer(queue);

queue.setDequeue_mode(DequeueMode.BROWSE);
queue.setTypeNames("mytype");

Message message = messageConsumer.receive();
System.out.println("Received: " + message);

message = messageConsumer.receive();

System.out.println("Received: " + message);
} catch (JMSEException jmsEx) {
    System.out.println(jmsEx.getMessage());
} finally {
    if(conn != null) {
        conn.close();
    }
}
}
```


第7章 Executing SQL commands with executeUpdate() or through PreparedStatement objects

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、他のタイプのコマンド (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。executeUpdate() メソッドは、行のコレクションを返す代わりに、実行する SQL コマンドの影響を受けた行数を返します。

executeUpdate() メソッドの署名は次のとおりです。

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

このメソッドに、実行する 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。

7.1 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。

注釈: 次の例は完全なアプリケーションではなく、メソッドです。これらのコードサンプルには、*Connection* のセットアップと解体に必要なコードが含まれていません。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。

```
public void updateEmployee(Connection con)
{
    try
    {
        int rowcount = stmt.executeUpdate("INSERT INTO "
            + "emp(empno, ename) VALUES(6000,Jones)");
        System.out.println("");
        System.out.println("Success - "+rowcount+
            " rows affected.");
    } catch(Exception err) {
        System.out.println("An error has occurred.");
    }
}
```

(次のページに続く)

(前のページからの続き)

```
System.out.println("See full details below.");
err.printStackTrace();
}
}
```

`updateEmployee()` メソッドは、呼び出し元から単一の引数、EDB Postgres Advanced Server データベースに接続する必要がある `Connection` オブジェクトを想定しています。

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

`executeUpdate()` メソッドは、SQL ステートメントの影響を受けた行数を返します（通常、`INSERT` は 1 つの行に影響しますが、`UPDATE` または `DELETE` ステートメントはさらに影響を与えます）。

```
int rowcount = stmt.executeUpdate("INSERT INTO emp(empno, ename)"
+" VALUES(6000,Jones)");
```

`executeUpdate()` がエラーなしで返った場合、`System.out.println` の呼び出しは、影響を受ける行数を示すメッセージをユーザーに表示します。

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

プログラムで例外が発生した場合、`catch` ブロックはユーザーに適切なエラーメッセージを表示します。

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

`executeUpdate()` は、結果セットを返さない SQL コマンドで使用できます。ただし、クエリをパラメータ化できる場合は、おそらく `PreparedStatement` を使用します。

7.2 `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、反復ごと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、サーバーは毎回ステートメントを解析、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、別の Statement 派生物である `PreparedStatement` を提供します。

次に、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` を示します。

```

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、プレースホルダーを挿入して、反復ごとに変更する値を表します。この例は、2つのプレースホルダー (それぞれが疑問符で表されます) 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(?,?)";
```

パラメーター化された SQL ステートメントを手に入れると、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、PreparedStatement オブジェクトを返すことができます。

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

この時点で、PreparedStatement は INSERT ステートメントを解析して計画しましたが、テーブルに追加する値がわかりません。PreparedStatement を実行する前に、setter メソッドを呼び出して、各プレースホルダーに値を指定する必要があります。setObject() は 2つの引数を必要とします。

- パラメータ番号。パラメーター番号 1 は最初の疑問符に対応し、パラメーター番号 2 は 2 番目の疑問符などに対応します。
- 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 と名前の入力を求め、ユーザーが指定した値を使用して setObject() を呼び出します。

```

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

```

次に、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と、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よってエラーメッセージが表示されます。

SQL コマンドを送信する `PreparedStatement` を使用した簡単な構文例を次に示します。

UPDATE コマンドを使用して行を更新するには：

```
String command = " UPDATE emp SET ename=? WHERE empno=?";
PreparedStatement stmt = con.prepareStatement(command);
stmt.setObject(1, c.readLine("Name:"));
stmt.setObject(2, new Integer(c.readLine("ID:")));
stmt.execute();
```

DROP TABLE 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:

```
String command = "DROP TABLE tableName";
PreparedStatement stmt = con.prepareStatement(command);
stmt.execute();
```

CREATE TABLE コマンド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：

```
String command = ("CREATE TABLE tablename (fieldname NUMBER(4,2), fieldname2_
→VARCHAR2(30))");
PreparedStatement stmt = con.prepareStatement(command);
stmt.execute();
```

ALTER TABLE コマンドを使用してテーブルの属性を変更するには：

```
String command = "ALTER TABLE tablename ADD COLUMN colname BOOLEAN ";
PreparedStatement stmt = con.prepareStatement(command);
stmt.execute();
```

第8章 Adding a graphical interface to a Java program

少しの作業で、プログラムにグラフィカルユーザーインターフェイスを追加できます。次の例は、JTable（スプレッドシートのようなグラフィカルオブジェクト）を作成し、クエリによって返されたデータをそのJTable にコピーする方法を示しています。

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。

```
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();
        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);
```

(次のページに続く)

(前のページからの続き)

```

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、JDK が提供するいくつかの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、呼び出し元によって 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 EDB Postgres Advanced Server 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、結果セットの形状（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）を説明するメタデータが含まれています。getMetaData() メソッドを呼び出すことにより、ResultSet からメタデータを抽出できます。

```

ResultSetMetaData rsmd = rs.getMetaData();

```

次に、showEmployees() は、列ヘッダーを保持するベクター（1 次元配列）を作成し、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));
}
```

列ヘッダーを配置したら、`showEmployees()` は `ResultSet` から各行を抽出し、新しいベクター（`rows` という名前）にコピーします。`rows` ベクトルは実際にはベクトルのベクトルです。`rows` ベクトルの各エンタリには、その行のデータ値を含むベクトルが含まれています。この組み合わせにより、`JTable` を構築するために必要な 2 次元配列が形成されます。`rows` ベクターを作成した後、プログラムは `ResultSet` の各行を読み取ります（`rs.next()` を呼び出すことにより）。各行の各列について、getter メソッドはその行/列の値を抽出し、その値を `rowValues` ベクトルに追加します。最後に、`showEmployee()` は各 `rowValues` ベクトルを `rows` ベクトルに追加します。

```
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);
}
```

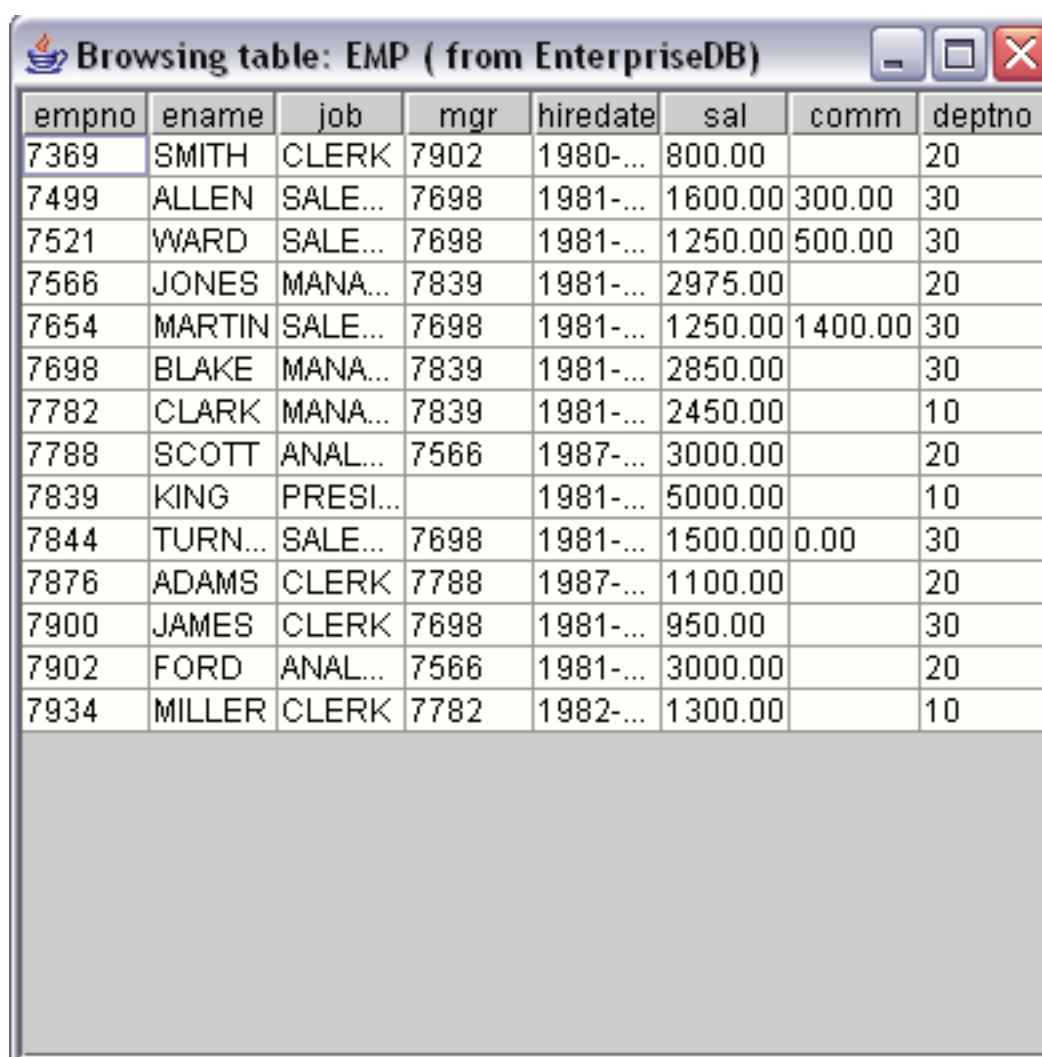
この時点で、ベクター（`labels`）には列ヘッダーが含まれ、2 番目の 2 次元ベクター（`rows`）にはテーブルのデータが含まれます。これで、ベクトルから `JTable` を作成し、`JTable` を保持することができます。

```
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、発生したエラーをインターセプトし、ユーザーに適切なメッセージを表示する `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: The showEmployees Window

第9章 Advanced JDBC Connector functionality

これらの例は、EDB Postgres Advanced Server JDBC コネクタのいくつかの高度な機能を示しています。

reduce_client-side_resource_requirements using_preparedstatements_to_send_sql_commandsunning_stored_procedures
using_ref_cursors_with_java using_bytea_data_with_java using_object_types_and_collections_with_java asyn-
chronous_notification_handling_with_noticelistener

9.1 Reducing client-side resource requirements

EDB Postgres Advanced Server JDBC ドライバーは、SQL クエリの結果を **ResultSet** オブジェクトとして取得します。バッチ化された **ResultSet** を使用して、クエリが多数の行を返す場合：

- 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削減します。

ResultSet オブジェクトのフェッチサイズを減らすと、ドライバーはネットワーク全体で（サーバーからクライアントへ）、**ResultSet** 全体をコピーしません。代わりに、ドライバーは一度に少数の行を要求します。クライアントアプリケーションが結果セットを移動すると、ドライバーはサーバーから次の行のバッチを取得します。

すべての状況でバッチ処理された結果セットを使用できるわけではありません。次の制限に従わないと、ドライバーはサイレントに **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。
- **TYPE_FORWARD_ONLY** タイプ（デフォルト）の **ResultSet** タイプで **Statement** オブジェクトを作成する必要があります。**TYPE_FORWARD_ONLY** 結果セットは、**ResultSet** を進むことのみできます。
- クエリは単一の SQL ステートメントで構成されている必要があります。

9.1.1 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、データの取得を高速化し、クライアント側アプリケーションが必要とするリソースを削減できます。次のコードは、5 行に制限されたバッチサイズで **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` を閉じません。 `Connection` を準備したら、 `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ついて、`println()` を呼び出すと `a row was returned` が出力されます。

```
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 行のみフェッチされます。クライアントの観点から、`batched` 結果セットと `unbatched` 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ことができることです。

9.2 Using PreparedStatements to send SQL commands

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、反復ごと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、サーバーは毎回ステートメントを解析、計画、および最適化する必要があります。JDBC は、このシナリオで必要な作業量を削減するために、別の `Statement` 派生物である `PreparedStatement` を提供します。

次のコードは、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` を呼び出します。

```

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、プレースホルダーを挿入して、反復ごとに変化する値を表します。次に、2つのプレースホルダー (それぞれ疑問符で表されます) を含む INSERT ステートメントを示します。

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

パラメーター化された SQL ステートメントを手に入れると、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、PreparedStatement オブジェクトを返すことができます。

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

この時点で、PreparedStatement は INSERT ステートメントを解析して計画しましたが、テーブルに追加する値がわかりません。PreparedStatement を実行する前に、setter メソッドを呼び出して、各プレースホルダーに値を指定する必要があります。setObject() は2つの引数を必要とします。

- パラメータ番号。パラメーター番号 1 は最初の疑問符に対応し、パラメーター番号 2 は 2 番目の疑問符などに対応します。
- 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 と名前の入力を求め、ユーザーが指定した値を使用して setObject() を呼び出します。

```

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

```

次に、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と、`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よってエラーメッセージが表示されます。

9.3 Executing stored procedures

ストアドプロシージャは、EDB の SPL で記述され、データベースに保存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ための入力パラメーターと、プロシージャからデータを返すための出力パラメーターを定義できます。ストアドプロシージャはサーバーで実行され、データベースから取得したデータを操作するデータベースアクセスコマンド (SQL)、制御ステートメント、およびデータ構造で構成されています。

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。また、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のも効率的です。

9.3.1 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ーを示します。こ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([?, ?, ...])}
```

各疑問符は、パラメーターのプレースホルダーとして機能します。ストアドプロシージャは、プレースホルダーが IN、OUT、または IN OUT パラメーターを表し、Java コードが一致する必要があるかどうかを判断します。

単純なストアドプロシージャの実行

以下は、各従業員の給与を 10 % 増やすストア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想定せず、情報を返しません。

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

次に、increaseSalary プロシージャを呼び出す方法を示します。

```
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 オブジェクトに作成を依頼することで CallableStatement オブジェクトを取得します。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()}");
```

名前が示すように、prepareCall() メソッドはステートメントを準備しますが、実行はしません。

IN パラメーターを使用したストアドプロシージャの実行 が示すように、アプリケーションは通常、prepareCall()

への呼び出しと execute() への呼び出しの間でパラメーター値をバインドします。サーバーでストアドプロシージャを呼び出すには、execute() メソッドを呼び出します。

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

このストアドプロシージャ (increaseSalary) は IN パラメーターを想定せず、呼び出し元に情報を返しませんでした (OUT パラメーターを使用)。

IN パラメーターを使用したストアードプロシージャの実行

次の例のコードでは、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。

以下は、EDB Postgres Advanced Server 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;
```

次に、Java からストアードプロシージャを呼び出す方法を示します。

```
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)
    {
    }
```

(次のページに続く)

(前のページからの続き)

```

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

```

コマンド (commandText) 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);

```

setObject() メソッドは、値を IN または IN OUT プレースホルダーにバインドします。setObject() への各呼び出しは、パラメーター番号とそのパラメーターにバインドする値を指定します。

```

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 パラメーターを使用したストアードプロシージャの実行

次の例では、deptSelect という SPL ストアドプロシージャを作成して呼び出します。このプロシージャは、1つの IN パラメーター（部門番号）を必要とし、部門番号に対応する2つの OUT パラメーター（部門の名前と場所）を返します。

```

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

```

(次のページに続く)

(前のページからの続き)

```
CLOSE deptCursor;
END;
```

deptSelect ストアドプロシージャを呼び出すために必要な Java コードを次に示します。

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

setObject() メソッドは、値を IN または IN OUT プレースホルダーにバインドします。setObject() を呼び出すときは、プレースホルダーを (その序数で) 識別し、そのプレースホルダーの代わりに値を指定する必要があります。

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

CallableStatement オブジェクトを実行する前に、各 OUT パラメーターの JDBC 型を登録します。JDBC 型の登録は registerOutParameter() メソッドで行います。

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

ステートメントの実行後、CallableStatement 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() getter 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));
```

この例では、GetDeptInfo() は 2 つの OUT パラメーターを登録し、(ストアドプロシージャの実行後に) OUT パラメーターで返された値を取得します。OUT パラメーターは両方とも VARCHAR 値として定義されているため、GetDeptInfo() は getString() メソッドを使用して OUT パラメーターを取得します。

9.3.2 IN OUT パラメーターを使用したストアドプロシージャの実行

次の例のコードは、1 つの IN パラメーター (p_deptno)、2 つの IN OUT パラメーター (p_empno および p_ename)、および 3 つの OUT パラメーター (p_job、empQuery_9、および empQuery は、2 つの IN OUT パラメーターと 3 つの OUT パラメーターで従業員に関する情報を返します。

このコードは、empQuery という名前のストア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;
```

次のコードは、empQuery プロシージャの呼び出し、IN パラメーターの値の指定、および OUT および 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 empQuery(?,?,?,?)}";
        CallableStatement stmt = con.prepareCall(commandText);
```

(次のページに続く)

(前のページからの続き)

```

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));
System.out.println("Employee Name: " + stmt.getString(3));
System.out.println("Job : " + stmt.getString(4));
System.out.println("Hired date : " + 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) 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 empQuery(?,?,?,?,?,?)}";
CallableStatement stmt = con.prepareCall(commandText);

```

setInt() メソッドは、Integer 値を IN または IN OUT プレースホルダーにバインドするタイプ固有の setter メソッドです。setInt() の呼び出しでは、パラメーター番号を指定し、そのプレースホルダーの代わりに値を提供します。

```

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 プ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);

```

(次のページに続く)

(前のページからの続き)

```
stmt.registerOutParameter(5, Types.TIMESTAMP);
stmt.registerOutParameter(6, Types.NUMERIC);
```

IN パラメーターを使用してプロシージャを呼び出す前に、セッターメソッドを使用してそのパラメーターに値を割り当てる必要があります。OUT パラメーターを使用してプロシージャを呼び出す前に、そのパラメーターの型を登録します。次に、ゲッターメソッドを呼び出して返された値を取得できます。IN OUT パラメーターを定義するプロシージャを呼び出すときは、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。
- パラメーターの種類を登録する。
- 戻り値を getter メソッドで取得する。

9.4 Using REF CURSORS with Java

REF CURSOR は、OPEN ステートメントによって返されるクエリ結果セットへのポインターを含むカーソル変数です。静的カーソルとは異なり、REF CURSOR は特定のクエリに関連付けられていません。異なるクエリを含む OPEN ステートメントで、同じ REF CURSOR 変数を何度でも開くことができます。毎回、そのクエリに対して新しい結果セットが作成され、カーソル変数を介して利用できるようになります。REF CURSOR は、あるプロシージャから別のプロシージャに結果セットを渡すこともできます。

EDB Postgres Advanced Server は、強い型指定と弱い型指定の両方の REF CURSOR 変数の宣言をサポートしています。厳密に型指定されたカーソルは、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。宣言された列を返すクエリでは、厳密に型指定されたカーソルのみを使用できます。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、サーバーは例外を返します。一方、弱く型付けされたカーソルは、任意の形状の結果セットを操作できます。

厳密に型指定された REF CURSOR を宣言するには：

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

弱く型付けされた REF CURSOR を宣言するには：

```
name SYS_REFCURSOR;
```

9.4.1 REF CURSOR を使用して ResultSet を取得する

次の (getEmpNames) に示すストアードプロシージャは、サーバー上に2つの REF CURSOR 変数をビルドします。最初の REF CURSOR には、emp テーブルの委託従業員のリストが含まれています。2番目の REF CURSOR には、emp テーブルの給与所得者のリストが含まれています。

```
CREATE OR REPLACE PROCEDURE getEmpNames
(
  commissioned OUT SYS_REFCURSOR,
  salaried 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;
```

次に示す RefCursorSample() メソッドは、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る名前を表示します。

```
public void RefCursorSample(Connection con)
{
  try
  {
    con.setAutoCommit(false);
    String commandText = "{call getEmpNames(?,?)}";
    CallableStatement stmt = con.prepareCall(commandText);
    stmt.registerOutParameter(1, 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、それぞれの REF CURSOR (commissioned および 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 OUT パラメーターとして返されます。

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

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

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

registerOutParameter() をもう 1 回呼び出すと、2 番目の REF CURSOR (salaried) の 2 番目のパラメータータイプ (Types.REF) が登録されます。

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

stmt.execute() を呼び出すと、次のステートメントが実行されます。

```
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、委託された各従業員の名前を出力します。

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

同じゲッターメソッドが 2 番目のパラメーターから ResultSet を取得し、RefCursorExample はそのカーソルを反復処理して、各サラリーマンの名前を出力します。

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

9.5 Using BYTEA data with Java

BYTEA データ型は、バイナリ文字列をバイトシーケンスで格納します。多くの場合、デジタル画像と音声ファイルはバイナリデータとして保存されます。EDB Postgres Advanced Server は、BYTEA データ型を介してバイナリデータを保存および取得できます。

次の Java サンプルは、EDB Postgres Advanced Server データベースに BYTEA データを保存し、そのデータを取得する方法を示しています。

まず、以下は BYTEA のデータを格納するテーブル (emp_detail) を作成します。emp_detail には 2 つの列が含まれています。

- 1 列目には社員の ID 番号 (INT 型) が格納され、テーブルの主キーになります。
- 2 列目は、社員の写真を BYTEA 形式で格納します。

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

以下は、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;
```

次に、次は特定の従業員の写真を返すファンクション (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;
```

9.5.1 EDB Postgres Advanced Server への BYTEA データの挿入

以下は、ADD_PIC プロシージャを呼び出して写真を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())
    {
```

(次のページに続く)

(前のページからの続き)

```

        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、ユーザーに従業員番号と画像ファイルの名前の入力求めます。

```

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 手続きを呼び出す CallableStatement オブジェクト (stmt) を用意します。最初のプレースホルダー (?) は、ADD_PIC (p_empno) が想定す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 に値を提供します。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ると、InsertPic() は CallableStatement を実行します。

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

すべてがうまくいくと、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();
}
```

9.5.2 EDB Postgres Advanced Server データベースから BYTEA データを取得する

Java アプリケーションから BYTEA データを挿入する方法を理解したので、以下にサーバーから BYTEA データを取得する方法を示します。

```
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 パラメーターを持つ 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 のデータなので、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に値を提供します。

```
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の入力ユーザーに求めます。

```
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+"\"");
```

9.6 Using object types and collections with Java

SQL CREATE TYPE コマンドを使用して、EDB Postgres Advanced Server データベースに保存されるユーザー定義の object type を作成します。CREATE TYPE コマンドは、一般的に配列と呼ばれるコレクションを作成するためにも使用されます。これも EDB Postgres Advanced Server データベースに保存されます。

これらのユーザー定義型は、SPL プロシージャ、SPL ファンクション、および Java プログラムで参照できます。

基本的なオブジェクトタイプは、CREATE TYPE AS OBJECT コマンドとオプションで CREATE TYPE BODY コマンドを使用して作成します。

CREATE TYPE AS TABLE OF コマンドを使用して、ネストした表型のコレクションを作成します。VARRAY 型のコレクションは、CREATE TYPE VARRAY コマンドで作成されます。

以下は、EDB Postgres Advanced Server データベースへの接続を確立するために、今後の両方の例で使用される 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;
}
```

9.6.1 オブジェクトタイプの使用

EDB Postgres Advanced Server 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つはオブジェクトタイプ `ADDR_OBJECT_TYPE` です。オブジェクトタイプの本文には、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);
```

(次のページに続く)

(前のページからの続き)

```

DBMS_OUTPUT.PUT_LINE(City/State/Zip: || SELF.addr.city || , ||
    SELF.addr.state || || LPAD(SELF.addr.zip,5,0));
END;
END;

```

以下は、これらのユーザー定義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
        // 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();
        }
    }
}

```

(次のページに続く)

(前のページからの続き)

```

    }
}

```

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して作成します。

```

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()` への呼び出しは、`emp_obj_typ` のパラメータータイプ (`Types.STRUCT`) を登録します。

```

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()` を呼び出すと、`display_emp()` メソッドの呼び出しが実行されます。

```

stmt.execute();

```

`getObject()` は、`emp_obj_typ` オブジェクト型を取得します。`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]);

```

9.6.2 コレクションの使用

EDB Postgres Advanced Server 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);
```

以下は、コレクション型 `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 ファンクションです。

このファンクションは、`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` パラメーターで返されます。

```
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;
```

以下は、前の関数を呼び出し、コレクション型を渡したり取得したりする 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"});
```

(次のページに続く)

(前のページからの続き)

```

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

// 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() ファンクションを呼び出す準備ができています。

```

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

```

createArrayOf() は、empIdArray および 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() への呼び出しは、OUT パラメーターのパラメータータイプ (Types.ARRAY) を登録します。

```

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() を呼び出すと、concatEmpIdName() 関数が呼び出されます。

```
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());
```

9.7 Asynchronous notification handling with NoticeListener

EDB Postgres Advanced Server JDBC コネクターは、非同期通知処理機能を提供します。通知は、SPL（または PL/pgSQL）プログラムが RAISE NOTICE ステートメントを実行したときにサーバーによって生成されるメッセージです。各通知はサーバーからクライアントアプリケーションに送信されます。JDBC クライアントで通知をインターセプトするには、アプリケーションで NoticeListener オブジェクト（または、より一般的には NoticeListener から派生したオブジェクト）を作成する必要があります。

SPL（または PL/pgSQL）プログラムを実行した結果、クライアントに通知が送信されることを理解することが重要です。通知を生成するには、ストアードプロシージャ、ファンクション、またはトリガーを呼び出す SQL ステートメントを実行する必要があります。SQL ステートメントが実行されると、通知がクライアントに配信されます。通知は、任意のタイプのステートメントオブジェクトで機能します: CallableStatement オブジェクト、PreparedStatement オブジェクト、または単純な Statement オブジェクト。JDBC プログラムは、NoticeListener を Statement オブジェクトに関連付けることにより、通知をインターセプトします。Statement オブジェクトが通知を発生させる SQL ステートメントを実行すると、JDBC は関連する NoticeListener の noticeReceived() メソッドを呼び出します。

以下は、emp テーブルをループし、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
```

(次のページに続く)

(前のページからの続き)

```
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;
```

次に、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ます。

```
public void NoticeExample(Connection con)
{
    CallableStatement stmt;
    try
    {
        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 オブジェクトを想定しています。

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

NoticeExample() は、前に示した adjustSalary プロシージャの呼び出しを準備することから始まります。ご想像のとおり、con.prepareCall() は CallableStatement オブジェクトを返します。CallableStatement を実行する前に、NoticeListener インターフェイスを実装するオブジェクトを作成し、そのオブジェクトを CallableStatement に関連付けられた NoticeListeners のリストに追加する必要があります。

```
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 ("Salary increased for ...") のテキスト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、stmt.execute() への呼び出しが完了する前に、noticeReceived() メソッドを（おそらく何度も）呼び出します。

```
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。

各 Statement オブジェクトは、NoticeListeners のリストを保持します。JDBC ドライバーは、サーバーから通知を受信すると、Statement オブジェクトが保持するリストを参照します。リストが空の場合、通知は Statement オブジェクトに保存されます。（execute() への呼び出しが完了したら、stmt.getWarnings() を呼び出すことで通知を取得できます。）リストが空でない場合、JDBC ドライバーは、リスナーが Statement に追加された順序で各リスナーに SQLWarning を配信します。

第10章 Security and encryption

PostgreSQL は、セキュリティを強化するために SSL 接続を使用してクライアント/サーバー通信を暗号化することをネイティブでサポートしています。これには、OpenSSL がクライアントシステムとサーバーシステムの両方にインストールされていること、および PostgreSQL でのサポートがビルド時に有効になっている必要があります。

using_ssl scram_compatibility support_for_gssapi_encrypted_connection

10.1 Using SSL

SSL を使用する場合は、次のことを考慮してください。

- サーバーの構成
- クライアントの構成
- SSL JDBC 接続のテスト
- 証明書の検証なしで SSL を使用する
- パスワードなしで証明書認証を使用する

configuration_the_server configuration_the_client testing_the_ssl_jdbc_connection using_certificate_authentication_without_a_password

10.1.1 Configuring the server

SSL 用に PostgreSQL または EDB Postgres Advanced Server を構成する方法については、[PostgreSQL documentation](#) を参照してください。

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 接続を確立している場合、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.
```

(次のページに続く)

edb=#

10.1.2 Configuring the client

SSL 用にクライアントを構成するために多くの接続パラメーターを使用できます。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、

Connecting to the database を参照してください。

異なる値が渡された場合、SSL 接続パラメーターの動作は異なります。接続パラメーター `ssl=true` をドライバに渡すと、ドライバは SSL 証明書を検証し、ホスト名を検証します。逆に、`libpq` を使用すると、デフォルトで非検証の SSL 接続が使用されます。

`sslmode` 接続パラメーターを使用して、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あり、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します。サーバーは、このホスト/IP アドレスの SSL 接続を受け入れ、サーバーがクライアント証明書を認識するように構成されています。

注釈: このモードでは、JDBC ドライバ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確認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、サーバー証明書に格納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。

証明書の検証が失敗する場合は、`sslcert=` を試すことができます。`LibPQFactory` はクライアント証明書を送信しません。証明書を使用して認証するようにサーバーが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、およびルート証明書の場所は、それぞれ `sslcert`、`sslkey`、および `sslrootcert` 設定でオーバーライドできます。これらはデフォルトでそれぞれ `/defaultdir/postgresql.crt`、`/defaultdir/postgresql.pk8`、および `/defaultdir/root.crt` です。 `defaultdir` は、Unix システムでは `${user.home}/.postgresql/`、Windows%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。これを行うには、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、および接続しているホストが証明書のホスト名と同じであることを確認します。

10.1.3 Testing the SSL JDBC connection

Java のデフォルトのメカニズム（`LibPQFactory` ではありません）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 で使用できるようにする必要があります。

1. Java プログラムで次のプロパティを設定します。

```
String 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
```

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、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： 接続の問題をトラブルシューティングするには、Java コマンドに `-Djavax.net.debug=ssl` を追加します。

証明書の検証なしで 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` を設定すると、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 の作成の詳細については、[PostgreSQL documentation](#) を参照してください。

10.1.4 Using certificate authentication without a password

パスワードなしで証明書認証を使用するには：

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
```

10.2 Scram compatibility

EDB JDBC ドライバーは、EDB Postgres Advanced Server 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 EDB JDBC Connector リリース 42.2.2.1 以降で利用できます。JRE / JDK バージョン 1.7 の場合、このサポートは EDB JDBC Connector リリース 42.2.5 以降で利用できます。

10.3 Support for GSSAPI-encrypted connection

EDB JDBC ドライバーは、EDB Postgres Advanced Server 12 以降の GSSAPI 暗号化接続をサポートしています。

`gssEncMode` パラメーターは、GSSAPI で暗号化された接続を制御します。パラメーターには、次のいずれかの値を指定できます。

- **Disable**。GSS 暗号化モードを使用した接続の試行を無効にします。
- **Allow**。プレーンテキストで接続しようとします。次に、サーバーが要求すると、暗号化モードに切り替わります。
- **Prefer**。暗号化モードで接続を試行し、暗号化された接続の取得に失敗した場合はプレーンテキストにフォールバックします。
- **Require**。暗号化モードで接続しようとしませんが、それが不可能な場合は接続に失敗します。

10.3.1 GSSAPI/SSPI 認証

Windows および Linux プラットフォームでの GSSAPI/SSPI 認証のデフォルトの動作は次のとおりです。

- Windows では、EDB JDBC ドライバーは SSPI を使用して接続しようとします。
- Linux では、EDB JDBC ドライバーは GSSAPI を使用して接続しようとします。

このデフォルトの動作は、次のいずれかの値を取る `gsslib` 接続パラメーターを使用して制御されます。

- **auto**。サーバーが要求すると、ドライバーは SSPI 認証を試みます。EDB JDBC クライアントは Windows で実行されており、SSPI に必要なワッフルライブラリは `CLASSPATH` にあります。それ以外の場合は、JSSE を介した Kerberos/GSSAPI 認証を選択します。`libpq` とは異なり、EDB JDBC ドライバーは Kerberos (GSSAPI) 要求に Windows SSPI ライブラリを使用しません。
- **gssapi**。このオプションは、SSPI が使用可能な場合でも、JSSE の GSSAPI 認証を強制します。
- **sspi**。このオプションは、SSPI 認証を強制します。この認証は、Linux または SSPI を使用できない場合に失敗します。

10.3.2 SSPI の使用 (Windows のみの環境)

EDB Postgres Advanced Server と JDBC クライアントが両方とも Windows の場合、JDBC ドライバーは次の接続文字列のいずれかを使用して SSPI 認証で接続します。

```
Class.forName("com.edb.Driver");
con = DriverManager.getConnection("jdbc:edb://localhost:5444/edb");
OR
con = DriverManager.getConnection("jdbc:edb://localhost:5444/edb?gsslib=sspi");
```

注釈:

- サーバーは SSPI 認証を必要とするため、`gsslib=sspi` はオプションです。- ユーザー名とパスワードを指定する必要はありません。ログインしたユーザーの資格情報は、ユーザーの認証に使用されます。
-

例

この例では、Windows マシンで SSPI 認証が構成されていることを前提としています。`edb-jdbc18.jar` のパスが<PATH_DRIVER> で、ワッフルライブラリのパスが<PATH_WAFFLE> であるとします。`CLASSPATH` を設定して JEdb サンプルを実行する方法は次のとおりです。

```
set CLASSPATH=<PATH_DRIVER>\edb-jdbc18.jar;<PATH_WAFFLE>\*;
javac JEdb.java
java JEdb
```

10.3.3 GSSAPI の使用 (Linux のみの環境)

EDB Postgres Advanced Server と JDBC クライアントが両方とも Linux 上にある場合、JDBC ドライバーは次の接続文字列を使用して GSSAPI 認証で接続します。

```
Class.forName("com.edb.Driver");
Properties connectionProps = new Properties();
connectionProps.setProperty("user", "postgres/myrealm.example@MYREALM.EXAMPLE");
String databaseUrl = "jdbc:edb://myrealm.example:5444/edb";
con = DriverManager.getConnection(databaseUrl, connectionProps);
```

注釈: サーバーは GSSAPI 認証を必要とするため、`gsslib=gssapi` はオプションです。

例

この例では、Linux マシンで GSS 認証が設定されていることを前提としています。

次の内容で `pgjdbc.conf` という名前のファイルを作成します。

```
pgjdbc {
com.sun.security.auth.module.Krb5LoginModule required
doNotPrompt=true
useTicketCache=true
renewTGT=true
debug=true;
};
```

`pgjdbc.conf` が `/etc/pgjdbc.conf` に配置されているとします。JEdb サンプルを実行する方法は次のとおりです。

```
javac JEdb.java
java -Djava.security.auth.login.config=/etc/pgjdbc.conf -cp .:edb-jdbc18.jar JEdb
```

10.3.4 SSPI/GSSAPI の使用 (Windows および Linux 環境)

EDB Postgres Advanced Server が Linux 上にあり、認証が GSSAPI として構成されており、JDBC クライアントが Windows にある場合、EDB JDBC は SSPI または GSSAPI 認証を使用して接続します。

gsslib=gssapi または gsslib=auto の場合、EDB JDBC は SSPI を使用します。gsslib=gssapi の場合、GSSAPI 認証を使用します。

例

この例では、Windows Active Directory と Linux マシンの間で GSS 認証が設定されていることを前提としています。

SSPI

このシナリオでは、JDBC は SSPI 認証を使用しています。次のコードを使用して接続を作成します。

edb_notranlate_5 この場合の EDB JDBC ベースのアプリの実行は、
SSPI の使用 (Windows のみの環境) で説明したものと同じです。

GSSAPI

このシナリオでは、JDBC は GSSAPI 認証を使用しています。次のコードを使用して接続を作成します。

```
Class.forName("com.edb.Driver");
Properties connectionProps = new Properties();
connectionProps.setProperty("user", "david@MYREALM.EXAMPLE");
String databaseUrl = "jdbc:edb://pg.myrealm.example:5444/edb?gsslib=gssapi";
con = DriverManager.getConnection(databaseUrl, connectionProps);
```

Kerberos 資格情報キャッシュファイルを設定し、チケットを取得します。

新しいディレクトリ、たとえば `c:\temp` とシステム環境変数 `KRB5CCNAME` を作成します。変数値フィールドに、`c:\temp\krb5cache` と入力します。

注釈: `krb5cache` は、Kerberos ソフトウェアによって管理されるファイルです。

MIT Kerberos Ticket Manager を使用するか、`ktpass` ユーティリティを使用して `keytab` ファイルを使用して、Kerberos プリンシパルのチケットを取得します。

GSSAPI の使用 (Linux のみの環境) で説明したのと同じ内容で `pgjdbc.conf`

ファイルを作成します。

pgjdbc.conf が c:\pgjdbc.conf に配置されているとします。JEdb サンプルを実行する方法は次のとおりです。

```
set CLASSPATH=C:\Program Files\edb\jdbc\edb-jdbc18.jar;  
java -Djava.security.auth.login.config=c:\pgjdbc.conf JEdb
```

第11章 EDB Postgres Advanced Server JDBC Connector logging

EDB Postgres Advanced Server JDBC コネクタは、アプリケーションで使用するときに JDBC コネクタの問題を解決するのに役立つロギングの使用をサポートしています。JDBC コネクタは、JDK 1.4 以降の Java の一部である `java.util.logging` のロギング API を使用します。 `java.util.logging` の詳細については、

[The PostgreSQL JDBC Driver](#) を参照してください。

注釈: EDB Postgres Advanced Server JDBC コネクタの以前のバージョンでは、カスタムメカニズムを使用してロギングを有効にしました。コミュニティバージョン 42.1.4.1 以降のバージョンでは、`java.util.logging` の使用に置き換えられました。古いメカニズムは使用できなくなりました。

Advanced Server JDBC コネクタの以前のバージョンでは、接続プロパティを使用してロギングを有効にできますが、42.3.3 以降は使用できません。

11.1 logging.properties でロギングを有効にする

デフォルトの Java ロギングフレームワークは、その構成を `logging.properties` というファイルに保存します。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、実行時のロギングの有効化/無効化が簡単になります。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` (`loggerLevel` にマップ) に設定します。

```
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
```

第12章 Reference - JDBC data types

次の表に、EDB Postgres Advanced Server と JDBC コネクタでサポートされている JDBC データ型をリストします。setObject() メソッドを使用して EDB Postgres Advanced Server 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ているオブジェクト型を返します。

JDBC Type	Advanced Server Type	getObject() returns
INTEGER	INT4	java.lang.Integer
TINYINT, SMALL-INT	INT2	java.lang.Integer
BIGINT	INT8	java.lang.Long
REAL	FLOAT4	java.lang.Float
DOUBLE, FLOAT	FLOAT8	java.lang.Double (Float is same as double)
DECIMAL, NUMERIC	NUMERIC	java.math.BigDecimal
CHAR	BPCHAR	java.lang.String
VARCHAR, LONG-VARCHAR	VARCHAR	java.lang.String
DATE	DATE	java.sql.Date
TIME	TIME, TIMETZ	java.sql.Timestamp
TIMESTAMP	TIMESTAMP, TIMESTAMPTZ	java.sql.Timestamp
BINARY	BYTEA	byte Release notes
BOOLEAN, BIT	BOOL	java.lang.Boolean
Types.REF	REFCURSOR	java.sql.ResultSet
Types.REF_CURSOR	REFCURSOR	java.sql.ResultSet
Types.OTHER	REFCURSOR	java.sql.ResultSet
Types.OTHER	UUID	java.util.UUID
Types.SQLXML	XML	java.sql.SQLXML

注釈: *Types.REF_CURSOR* は JRE 4.2 でのみサポートされています。

Types.OTHER は UUID に使用されるだけでなく、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判断できるようにする場合にも使用されます。パラメーターが *java.util.UUID* のインスタンスの場合、ドライバーは適切な内部型を判断し、サーバーに送信します。