



EDB

jdbc_connector

リリース 42.3.3.1

EDB

2022 年 05 月 11 日

目次

第 1 章	Release notes	3
1.1	Version 42.3.3.1	3
1.2	Version 42.3.2.1	4
1.3	Version 42.2.24.1	4
1.4	Version 42.2.19.1	5
1.5	Version 42.2.12.3	5
1.6	Version 42.2.9.1	5
1.7	Version 42.2.8.1	6
第 2 章	Requirements overview	7
2.1	サポートされている JDK ディストリビューション	7
第 3 章	EDB Postgres Advanced Server JDBC Connector overview	9
3.1	JDBC ドライバのタイプ	9
3.2	JDBC インタフェース	10
3.3	JDBC クラスとインターフェース	10
3.4	JDBC DriverManager	10
3.5	EDB Postgres Advanced Server JDBC コネクタの互換性	10
第 4 章	Installing and configuring EDB JDBC Connector	13
4.1	Installing EDB JDBC Connector on Linux	13
4.1.1	Installing EDB JDBC Connector on IBM Power (ppc64le)	14
4.1.2	Installing EDB JDBC Connector on Linux x86 (amd64)	17
4.2	Using the graphical installer to install the connector	27
4.3	Configuring the EDB Postgres Advanced Server JDBC Connector	33
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	38
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	サーバー側のセットアップ	51

6.2.1	EDB-PSQL を使用する	51
6.3	クライアント側の例	53
6.3.1	メッセージをエンキューおよびデキューする	54
6.3.2	メッセージをキューに入れる	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	55
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	70
9.2	Using PreparedStatement to send SQL commands	71
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	80
9.5	Using BYTEA data with Java	82
9.5.1	BYTEA データを EDB Postgres Advanced Server に挿入する	83
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	86
9.6.1	オブジェクトタイプの使用	86
9.6.2	コレクションを使用する	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	99
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)	100
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	ロギング .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 コネクタは Java で記述されており、Sun の JDK アーキテクチャに準拠しています。詳細については、[JDBC ドライバのタイプ](#) を参照してください

EDB JDBC コネクタは、PostgreSQL コミュニティドライバのすべての機能に基づいて構築され、サポートされています。ドライバの機能の詳細については、[documentation](#) コミュニティをご覧ください。

release_notes 要件_advanced_server_jdbc_connector_overview_installing_and_configuring_the_jdbc_connector
using_advanced_server_jdbc_connector_with_java_applications execution_sql_commands_with_executeUpdate ()

</ div>

第1章 Release notes

EDB JDBC コネクタの文書には、EDB JDBC コネクタの最新バージョンが記載されています。

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ています。マイナーリリースまたはパッチリリースで新しい機能が導入されると、コンテンツ内のインジケータは、新しい機能を導入したバージョンを識別します。

Version	Release Date
[42.3.3.1](08_jdbc_42.3.3.1_rel_notes)	2022 Apr 20
[42.3.2.1](09_jdbc_42.3.2.1_rel_notes)	2022 Feb 15
[42.2.24.1](10_jdbc_42.2.24.1_rel_notes)	2021 Nov 5
[42.2.19.1](12_jdbc_42.2.19.1_rel_notes)	2021 Apr 15
[42.2.12.3](14_jdbc_42.2.12.3_rel_notes)	2020 Oct 22
[42.2.9.1](16_jdbc_42.2.9.1_rel_notes)	2020 May 18
[42.2.8.1](18_jdbc_42.2.8.1_rel_notes)	2019 Oct 21

1.1 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 コネクタ 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以下が含まれます。

Type	Description
Upstream Merge	アップストリーム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](https://github.com/advisories/GHSA-673j-qm5f-xpv8)：この修正のパートとして <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、ドライバのロギングの構成には使用できなくなります。代わりに <code>java.util.logging.properties</code> などのロギング構成メカニズム。
Change	セキュリティ修正 GHSA-673j-qm5f-xpv のパートとして、接続プロパティを使用してロギングを有効にする機能は、バージョン 42.3.3 から使用できなくなりました。

1.2 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 コネクタ 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以下が含まれます。

Type	Description
Upstream merge	アップストリームコミュニティドライババージョン 42.3.2 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_42.3.2) を参照してください。
New feature	<code>`org.checkerframework.*`</code> は、以前は EDB JDBC jar ファイルにパッケージ化されていました。異なるバージョンの <code>`org.checkerframework.*`</code> を使用する他のアプリケーションとの競合を引き起こします。新しい機能は、シェードプラグインを使用して、コネクタのカスタム名前空間の下にチェッカーフレームワークをパッケージ化します。[サポートチケット：# 74134]
New feature	DBMS_AQ パッケージとシームレスに対話するための JMS ベースの API。この API は、 <code>com.edb.jms</code> および <code>com.edb.aq</code> パッケージソフトの下の <code>edb-jdbc</code> コードのパートになっています。
Enhancement	<code>prepareStatement</code> に <code>setTimeStamp</code> メソッドを使用する場合、ドライバのデフォルトの動作を変更するために使用される新しいプロパティ <code>oidTimestamp</code> 。プロパティ <code>oidTimestamp</code> が <code>true</code> に設定されている場合、 <code>oid</code> を <code>Oid.TIMESTAMP</code> に設定します。それ以外の場合はデフォルトの動作を使用します。
Bug fix	問題：EDB * PLUS の日付フォーマット <code>nls_date_format = 'YYYY / MM / DD'</code> を変更するとエラーが発生します。[サポートチケット：# 75812]
Bug fix	EDB とオラクルの丸めの違い。[サポートチケット：# 72708]
Security fix	v42.3.2 とのコミュニティマージのパートとしての CVE-2022-21724
Security fix	CVE-2021-36373-org.apache.ant の依存関係を削除
Security fix	CVE-2020-15250-一時フォルダーの junit 修正。

1.3 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 コネクタ 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以下が含まれます。

Type	Description
Upstream merge	アップストリーム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。

1.4 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 コネクタ 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以下が含まれます。

Type	Description
Upstream merge	アップストリーム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。詳細については、[community 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、GSSAPI 暗号化接続をサポートするようになりました。 [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 Connector 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 コネクタ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。

1.5 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 コネクタ 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以下が含まれます。

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

1.6 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 コネクタ 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以下が含まれます。

Type	Description
Enhancement	EDB JDBC コネクタは、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。

1.7 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 コネクタ 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以下が含まれます。

Type	Description
Upstream merge	アップストリームコミュニティドライババージョン 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 コネクタが EDB Postgres Advanced Server 12 をサポートするようになりました。
Enhancement	EDB JDBC コネクタが 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 Requirements overview

</div>

この表には、最新の JDBC コネクタバージョンと、対応する EDB Postgres Advanced Server (EPAS) バージョンがリストされています。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。詳細については、[Product Compatibility](#) を参照してください。

JDBC Connector	EPAS 14	EPAS 13	EPAS 12	EPAS 11	EPAS 10
[42.3.3.1](01_jdbc_rel_notes/08_jdbc_42.3.3.1_rel_notes.mdx)	Y	Y	Y	Y	Y
[42.3.2.1](01_jdbc_rel_notes/09_jdbc_42.3.2.1_rel_notes.mdx)	Y	Y	Y	Y	Y
[42.2.24.1](01_jdbc_rel_notes/10_jdbc_42.2.24.1_rel_notes.mdx)	Y	Y	Y	Y	Y
[42.2.19.1](01_jdbc_rel_notes/12_jdbc_42.2.19.1_rel_notes.mdx)	N	Y	Y	Y	Y
[42.2.12.3](01_jdbc_rel_notes/12_jdbc_42.2.12.3_rel_notes.mdx)	N	N	Y	Y	Y
[42.2.9.1](01_jdbc_rel_notes/12_jdbc_42.2.9.1_rel_notes.mdx)	N	N	Y	Y	Y
[42.2.8.1](01_jdbc_rel_notes/12_jdbc_42.2.8.1_rel_notes.mdx)	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 以上 (LTS バージョン)、オラクル JDK、OpenJDK、および IBM SDK (Java) ディストリビューションを含む。

第3章 EDB Postgres Advanced Server JDBC Connector overview

</ div>

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。

</ div>

3.1 JDBC ドライバのタイプ

現在、4 種類の JDBC ドライバがあり、それぞれ独自の実装、使用、制限があります。EDB JDBC コネクタは、タイプ 4 ドライバです。

タイプ 1 ドライバ

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

タイプ 2 ドライバ

- これはネイティブデータベースライブラリドライバです。
- コンピュータ上のネイティブデータベースライブラリを使用してデータベースにアクセスします。
- 通常、ネイティブメソッド呼び出しのためにアプレット内で実行できません。
- クライアントにデータベースライブラリインストール済んでいる必要があります。

タイプ 3 ドライバ

- 100 % Java ドライバー、ネイティブメソッドなし。
- クライアントに事前インストールする必要はありません。
- Java クラスファイルと同じようにオンザフライでダウンロードおよび構成できます。
- ミドルウェアサーバーとの通信に独自のプロトコルを使用します。

- ミドルウェアサーバーは、独自の呼び出しから DBMS 固有の呼び出しに変換します。

タイプ 4 ドライバ

- 100 % Java ドライバ、ネイティブメソッドなし。
- クライアントに事前インストールする必要はありません。
- Java クラスファイルと同じようにオンザフライでダウンロードおよび構成できます。
- タイプ 3 ドライバとは異なり、DBMS サーバーと直接通信します。JDBC 呼び出しをデータベース固有の呼び出しに直接変換します。

3.2 JDBC インタフェース

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。これらのインタフェースは、`java.sql` パッケージに実装されています。

3.3 JDBC クラスとインタフェース

コア API は、クラスとインタフェースで構成されています。これらのクラスとインタフェースは、図に示すように連携して機能します。

3.4 JDBC DriverManager

この図は、典型的な JDBC アプリケーションでの `DriverManager` クラスのロールを示しています。`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、ドライバの現在のバージョンです。異常な要件（古いアプリケーションまたは JVM の実行）がない限り、これが使用するドライバです。このドライバは PostgreSQL 9.6 以降のバージョンをサポートし、Java 8 以降のバージョンが必要です。SSL および `javax.sql` パッケージのサポートが含まれています。

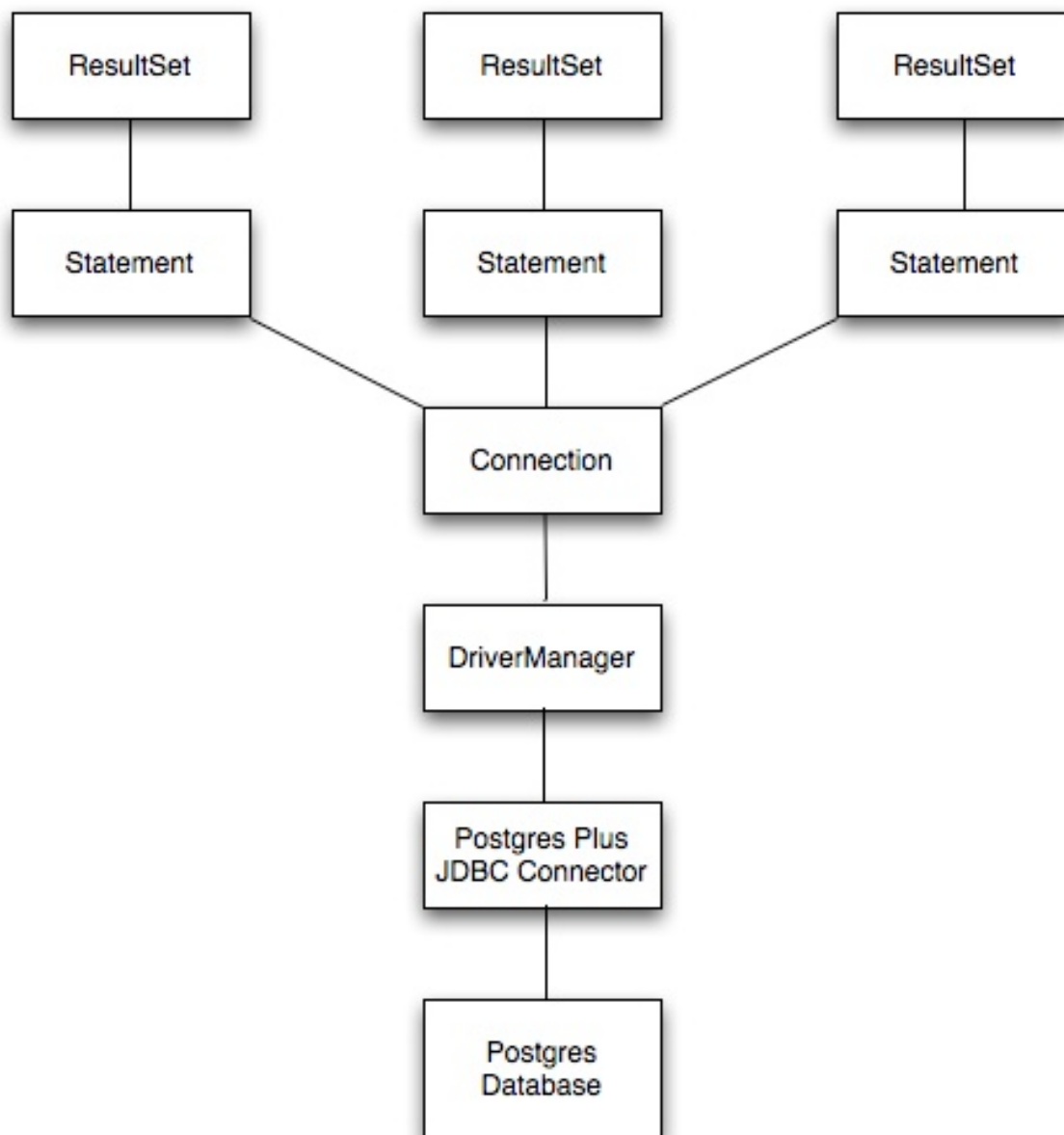


図 1: JDBC Class Relationships

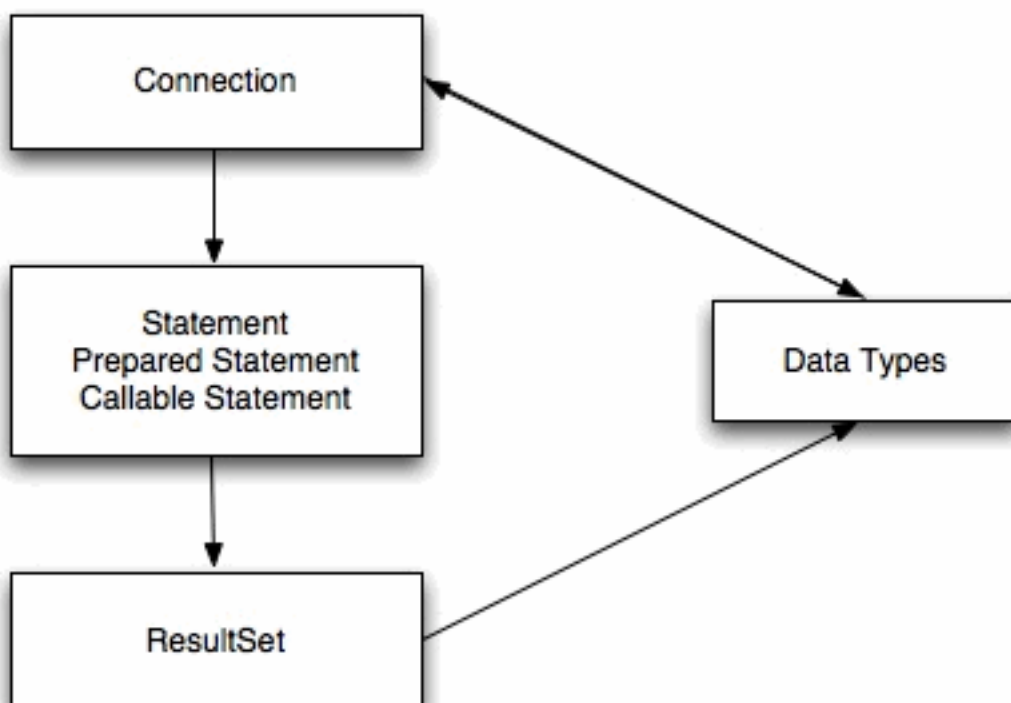


図 2: Core Classes and Interfaces

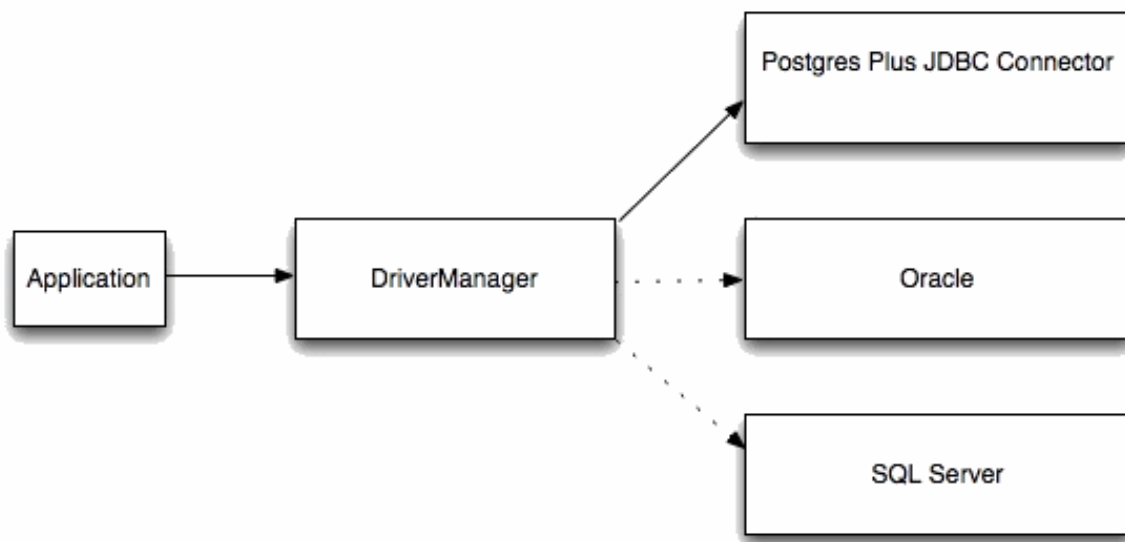


図 3: DriverManager/Drivers

第4章 Installing and configuring EDB JDBC Connector

</ div>

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。オラクル Java ダウンロード [website](#) から、環境に合った Java インストーラ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または RPM パッケージを使用して、インストールに EDB JDBC コネクタを追加できます。

4.1 Installing EDB JDBC Connector on Linux

EDB JDBC コネクタ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、以下を参照してください。

- Linux x86-64 (amd64) :
 - *Installing EDB JDBC Connector on RHEL 8/OL 8 x86*
 - *Installing EDB JDBC Connector on Rocky Linux 8/AlmaLinux 8 x86*
 - *Installing EDB JDBC Connector on RHEL 7/OL 7 x86*
 - *Installing EDB JDBC Connector on CentOS 7 x86*
 - *Installing EDB JDBC Connector on SLES 15 IBM Power (ppc64le)*
 - *Installing EDB JDBC Connector on SLES 12 IBM Power (ppc64le)*
 - *Installing EDB JDBC Connector on Ubuntu 20.04/Debian 10 x86 - Installing EDB JDBC Connector on Ubuntu 18.04/Debian 9 x86*
- Linux on IBM Power (ppc64le) :
 - *Installing EDB JDBC Connector on RHEL 8 IBM Power (ppc64le)*
 - *Installing EDB JDBC Connector on SLES 15 IBM Power (ppc64le) - Installing EDB JDBC Connector on SLES 12 IBM Power (ppc64le)*

4.1.1 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、以下を参照してください。

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

Installing EDB JDBC Connector on RHEL 8 IBM Power (ppc64le)

インストールを完了するには2つの手順があります。

- リポジトリのセットアップ
- パッケージのインストール

各ステップで、スーパーユーザとしてログインする必要があります。

スーパーユーザとしてログするには：

```
sudo su -
```

リポジトリのセットアップ

1. EDB に登録して EDB リポジトリの資格情報を受け取るには、[Repository Access Request](#) にアクセスしてください。
2. EDB リポジトリをセットアップします。

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

これにより、/etc/yum.repos.d/edb.repo 構成ファイルが作成されます。

1. EDB 認証情報を edb.repo ファイルに追加します。

```
sed -i "s<username>:<password>@USERNAME:PASSWORD@" /etc/yum.repos.d/edb.repo
```

USERNAME:PASSWORD は、[EDB account](#) から入手できるユーザー名とパスワードです。

1. EPEL リポジトリをインストールし、キャッシュをリフレッシュします。

```
dnf -y install https://dl.fedoraproject.org/pub/epel/epel-release-latest-8.noarch.rpm
dnf makecache
```

1. EPEL パッケージソフトはそこからのパッケージソフトに依存する可能性があるため、codeready-builder-for-rhel-8-[*-rpms](#) リポジトリを有効にします。

```
ARCH=$( /bin/arch )
subscription-manager repos --enable "codeready-builder-for-rhel-8-ARCH-rpms"
```

1. ビルトインの PostgreSQL モジュールを無効にします：edb_notranlate_5

パッケージのインストール

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

Installing EDB JDBC Connector on SLES 15 IBM Power (ppc64le)

インストールを完了するには2つの手順があります。

- リポジトリのセットアップ
- パッケージのインストール

各ステップで、スーパーユーザとしてログインする必要があります。

```
# To log in as a superuser:
sudo su -
```

リポジトリを設定する前に、EDB に登録する必要があります。EDB リポジトリの資格情報を受信するには、[Repository Access Request](#) にアクセスしてください。

リポジトリのセットアップ

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

```
# Install the repository configuration and enter your EDB repository
# credentials when prompted
zypper addrepo https://zypp.enterprisedb.com/suse/edb-sles.repo

# Install SUSEConnect to register the host with SUSE, allowing access to
# SUSE repositories
zypper install SUSEConnect

# Register the host with SUSE, allowing access to SUSE repositories
# Replace REGISTRATION_CODE and EMAIL with your SUSE registration
# information
SUSEConnect -r REGISTRATION_CODE -e EMAIL

# Activate the required SUSE module
```

(次のページに続く)

(前のページからの続き)

```
SUSEConnect -p PackageHub/15.3/ppc64le

# Refresh the metadata
zypper refresh
```

パッケージのインストール

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

Installing EDB JDBC Connector on SLES 12 IBM Power (ppc64le)

インストールを完了するには2つの手順があります。

- リポジトリのセットアップ
- パッケージのインストール

各ステップで、スーパーユーザとしてログインする必要があります。

```
# To log in as a superuser:
sudo su -
```

リポジトリを設定する前に、EDB に登録する必要があります。EDB リポジトリの資格情報を受信するには、[Repository Access Request](#) にアクセスしてください。

リポジトリのセットアップ

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

```
# Install the repository configuration and enter your EDB repository
# credentials when prompted
zypper addrepo https://zypp.enterprisedb.com/suse/edb-sles.repo

# Install SUSEConnect to register the host with SUSE, allowing access to
# SUSE repositories
zypper install SUSEConnect

# Register the host with SUSE, allowing access to SUSE repositories
# Replace REGISTRATION_CODE and EMAIL with your SUSE registration
# information
SUSEConnect -r REGISTRATION_CODE -e EMAIL
```

(次のページに続く)

(前のページからの続き)

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

# Refresh the metadata
zypper refresh

# Install OpenJDK (version 1.8)
zypper -n install java-1_8_0-openjdk
```

パッケージのインストール

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

4.1.2 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、以下を参照してください。

- *Installing EDB JDBC Connector on RHEL 8/OL 8 x86*
- *Installing EDB JDBC Connector on Rocky Linux 8/AlmaLinux 8 x86*
- *Installing EDB JDBC Connector on RHEL 7/OL 7 x86*
- *Installing EDB JDBC Connector on CentOS 7 x86*
- *Installing EDB JDBC Connector on SLES 15 IBM Power (ppc64le)*
- *Installing EDB JDBC Connector on SLES 12 IBM Power (ppc64le)*
- *Installing EDB JDBC Connector on Ubuntu 20.04/Debian 10 x86*
- *Installing EDB JDBC Connector on Ubuntu 18.04/Debian 9 x86*

Installing EDB JDBC Connector on RHEL 8/OL 8 x86

JDBC コネクタをインストールする前に、以下の前提条件パッケージソフトをインストールし、EDB から資格情報を要求する必要があります。

epel-release パッケージをインストールします。

```
dnf -y install https://dl.fedoraproject.org/pub/epel/epel-release-latest-8.noarch.rpm
```

codeready-builder-for-rhel-8-**-rpms* リポジトリを有効にします。

```
ARCH=$( /bin/arch )
subscription-manager repos --enable "codeready-builder-for-rhel-8-${ARCH}-rpms"
```

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

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

リポジトリの資格情報を受け取った後、次のことができます。

1. リポジトリ設定ファイルを作成します。ファイルを変更し、ユーザ名とパスワードを入力します。
edb-jdbc をインストールします。

リポジトリ設定ファイルの作成

リポジトリ構成ファイルを作成するには、スーパーユーザ特権を想定して、次のコマンドを呼び出します。

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

リポジトリ構成ファイルの名前付けは `edb.repo` です。ファイルは `/etc/yum.repos.d` にあります。

ファイルの変更、ユーザ名とパスワードの提供

`edb.repo` ファイルを作成した後、エディタの選択を使用して、`enabled` パラメータの値が 1 であることを保証し、`baseurl` 仕様の `username` および `password` プレースホルダーを登録済み EDB ユーザの名前とパスワードに置き換えます。

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

JDBC コネクタのインストール

構成ファイルへの変更を保存した後、以下のコマンドを使用して JDBC コネクタをインストールします。

```
dnf install edb-jdbc
```

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

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

Installing EDB JDBC Connector on Rocky Linux 8/AlmaLinux 8 x86

JDBC コネクタをインストールする前に、以下の前提条件パッケージソフトをインストールし、EDB から資格情報を要求する必要があります。

epel-release パッケージをインストールします。

```
dnf -y install https://dl.fedoraproject.org/pub/epel/epel-release-latest-8.noarch.rpm
```

PowerTools リポジトリを有効にします。

```
dnf config-manager --set-enabled PowerTools
```

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

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

リポジトリの資格情報を受け取った後、次のことができます。

1. リポジトリ設定ファイルを作成します。ファイルを変更し、ユーザ名前とパスワードを入力します。edb-jdbc をインストールします。

リポジトリ設定ファイルの作成

リポジトリ構成ファイルを作成するには、スーパーユーザ特権を想定して、次のコマンドを呼び出します。

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

リポジトリ構成ファイルの名前付けは `edb.repo` です。ファイルは `/etc/yum.repos.d` にあります。

ファイルの変更、ユーザ名前とパスワードの提供

`edb.repo` ファイルを作成した後、エディタの選択を使用して、`enabled` パラメータの値が 1 であることを保証し、`baseurl` 仕様の `username` および `password` プレースホルダーを登録済み EDB ユーザの名前とパスワードに置き換えます。

```
[edb]
name=EnterpriseDB RPMs $releasever - $basearch
baseurl=https://<username>:<password>@yum.enterprisedb.com/edb/redhat/rhel-
→$releasever-$basearch
enabled=1
gpgcheck=1
```

(次のページに続く)

(前のページからの続き)

```
repo_gpgcheck=1
gpgkey=file:///etc/pki/rpm-gpg/ENTERPRISEDB-GPG-KEY
```

JDBC コネクタのインストール

構成ファイルへの変更を保存した後、次のコマンドを使用して JDBC コネクタをインストールします。

```
dnf install edb-jdbc
```

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

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

Installing EDB JDBC Connector on RHEL 7/OL 7 x86

JDBC コネクタをインストールする前に、以下の前提条件パッケージソフトをインストールし、EDB から資格情報を要求する必要があります。

epel-release パッケージをインストールします。

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

オプション、エクストラ、HA リポジトリを有効にします。

```
subscription-manager repos --enable "rhel-*-optional-rpms" --enable "rhel-*-extras-  
rpms" --enable "rhel-ha-for-rhel-*-server-rpms"
```

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

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

リポジトリの資格情報を受け取った後、次のことができます。

1. リポジトリ設定ファイルを作成します。ファイルを変更し、ユーザ名前とパスワードを入力します。edb-jdbc をインストールします。

リポジトリ設定ファイルの作成

リポジトリ構成ファイルを作成するには、スーパーユーザ特権を想定して、次のコマンドを呼び出します。

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

リポジトリ構成ファイルの名前付けは `edb.repo` です。ファイルは `/etc/yum.repos.d` にあります。

ファイルの変更、ユーザ名前とパスワードの提供

`edb.repo` ファイルを作成した後、エディタの選択を使用して、`enabled` パラメータの値が 1 であることを保証し、`baseurl` 仕様の `username` および `password` プレースホルダーを登録済み EDB ユーザの名前とパスワードに置き換えます。

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

JDBC コネクタのインストール

構成ファイルへの変更を保存した後、次のコマンドを使用して JDBC コネクタをインストールします。

```
yum install edb-jdbc
```

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

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

Installing EDB JDBC Connector on CentOS 7 x86

JDBC コネクタをインストールする前に、以下の前提条件パッケージソフトをインストールし、EDB から資格情報を要求する必要があります。

epel-release パッケージをインストールします。

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

注釈: *CentOS-Base.repo* ファイル（*/etc/yum.repos.d* にある）で *[extras]* リポジトリ定義を有効にする必要がある場合があります。

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

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

リポジトリの資格情報を受け取った後、次のことができます。

1. リポジトリ設定ファイルを作成します。ファイルを変更し、ユーザ名とパスワードを入力します。
edb-jdbc をインストールします。

リポジトリ設定ファイルの作成

リポジトリ構成ファイルを作成するには、スーパーユーザ特権を想定して、次のコマンドを呼び出します。

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

リポジトリ構成ファイルの名前付けは *edb.repo* です。ファイルは */etc/yum.repos.d* にあります。

ファイルの変更、ユーザ名とパスワードの提供

edb.repo ファイルを作成した後、エディタの選択を使用して、*enabled* パラメータの値が 1 であることを保証し、*baseurl* 仕様の *username* および *password* プレースホルダーを登録済み EDB ユーザの名前とパスワードに置き換えます。

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

JDBC コネクタのインストール

構成ファイルへの変更を保存した後、次のコマンドを使用して JDBC コネクタをインストールします。

```
yum install edb-jdbc
```

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

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

Installing EDB JDBC Connector on SLES 15 x86

インストールを完了するには 2 つの手順があります。

- リポジトリのセットアップ
- パッケージのインストール

各ステップで、スーパーユーザとしてログインする必要があります。

```
# To log in as a superuser:  
sudo su -
```

リポジトリを設定する前に、EDB に登録する必要があります。EDB リポジトリの資格情報を受信するには、[Repository Access Request](#) にアクセスしてください。

リポジトリのセットアップ

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

```
# Install the repository configuration and enter your EDB repository  
# credentials when prompted  
zypper addrepo https://zypp.enterprisedb.com/suse/edb-sles.repo  
  
# Install SUSEConnect to register the host with SUSE, allowing access to  
# SUSE repositories  
zypper install SUSEConnect  
  
# Register the host with SUSE, allowing access to SUSE repositories  
# Replace REGISTRATION_CODE and EMAIL with your SUSE registration  
# information  
SUSEConnect -r REGISTRATION_CODE -e EMAIL
```

(次のページに続く)

(前のページからの続き)

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

# Refresh the metadata
zypper refresh
```

パッケージのインストール

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

Installing EDB JDBC Connector on SLES 12 x86

コネクタをインストールするには、EnterpriseDB リポジトリへのアクセスを許可する資格情報が必要です。リポジトリの資格証明をリクエストするには、次の EnterpriseDB ウェブサイトにアクセスしてください。

<https://www.enterprisedb.com/repository-access-request>

zypper パッケージマネージャを使用して、SLES 12 ホストにコネクタをインストールできます。zypper は、パッケージをインストールするときにパッケージの依存関係を満たそうとしますが、EnterpriseDB でホストされていない特定のリポジトリへのアクセスが必要です。

1. コネクタをインストールする前に、スーパーユーザ権限を引き受け、ファイアウォールを停止する必要があります。次に、以下のコマンドを使用して、EnterpriseDB リポジトリをシステムに追加します。

```
zypper addrepo https://zypp.enterprisedb.com/suse/edb-sles.repo
```

2. コマンドは、`/etc/zypp/repos.d` ディレクトリにリポジトリ構成ファイルを作成します。次に、次のコマンドを使用して SLES ホストのメタデータをリフレッシュし、EnterpriseDB リポジトリを含めます。

```
zypper refresh
```

プロンプトが表示されたら、リポジトリの資格情報を提供し、提供されたキーを常に信頼するように指定し、EnterpriseDB リポジトリを含むようにメタデータを更新します。

3. また、SUSE Connect と SUSE Package Hub 拡張を SLES ホストに追加し、ホストを SUSE に登録して、SUSE リポジトリへのアクセスを許可する必要があります。「REGISTRATION_CODE」と「EMAIL」を SUSE 登録情報に置き換えます。次のコマンドを使用します。

```
zypper install SUSEConnect
SUSEConnect -r REGISTRATION_CODE -e EMAIL
SUSEConnect -p PackageHub/12.4/x86_64
SUSEConnect -p sle-sdk/12.4/x86_64
```

4. 依存関係を解決するために次のリポジトリをインストールします。

```
zypper addrepo https://download.opensuse.org/repositories/Apache:/Modules/SLE_12_SP4/
↳ Apache:Modules.repo
```

5. SLES ホストのメタデータを更新します：edb_notranlate_4

6. OpenJDK (バージョン 1.8) をインストールします。

```
zypper -n install java-1_8_0-openjdk
```

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

```
zypper install edb-jdbc
```

Installing EDB JDBC Connector on Ubuntu 20.04/Debian 10 x86

Debian または Ubuntu ホストに DEB パッケージをインストールするには、EDB リポジトリへのアクセスを許可する資格情報が必要です。リポジトリの資格情報を要求するには、[the EDB website](#) にアクセスしてください。

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

1. スーパーユーザ特権を想定します。

```
sudo su -
```

2. EDB リポジトリを構成します。

3. EDB リポジトリをセットアップします。

```
sh -c echo "deb [arch=amd64] https://apt.enterprisedb.com/$(lsb_release -cs)-edb/
↳ $(lsb_release -cs) main" > /etc/apt/sources.list.d/edb-$(lsb_release -cs).list
```

2. 次のコマンドで、**username** および **password** を EDB 資格情報に置き換えます。

```
sh -c echo "machine apt.enterprisedb.com login <username> password <password>" > /etc/
↳ apt/auth.conf.d/edb.conf
```

3. 安全な APT リポジトリのサポートをシステムに追加します。

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

4. EDB 署名キーを追加します。

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

5. リポジトリメタデータを更新します。

```
apt-get update
```

6. DEB パッケージをインストールします。

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

Installing EDB JDBC Connector on Ubuntu 18.04/Debian 9 x86

Debian または Ubuntu ホストに DEB パッケージをインストールするには、EDB リポジトリへのアクセスを許可する資格情報が必要です。リポジトリの資格情報を要求するには、[the EDB website](#) にアクセスしてください。

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

1. スーパーユーザ特権を想定します。

```
sudo su -
```

2. EDB リポジトリを構成する

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

3. 安全な APT リポジトリのサポートをシステムに追加します。

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

4. EDB 署名キーを追加します。

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

5. リポジトリメタデータを更新します。

```
apt-get update
```

6. DEB パッケージをインストールします。

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

注釈: デフォルトでは、Debian 9x および Ubuntu 18.04 プラットフォームは Java バージョン 10 をインストールします。システムに Java バージョン 8 をインストールして、EDB Java ベースのコンポーネントを実行してください。

4.2 Using the graphical installer to install the connector

</ div>

EDB コネクタインストールウィザードを使用して、システムに JDBC コネクタを追加できます。ウィザードは [EDB website](#) で利用可能です。

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

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

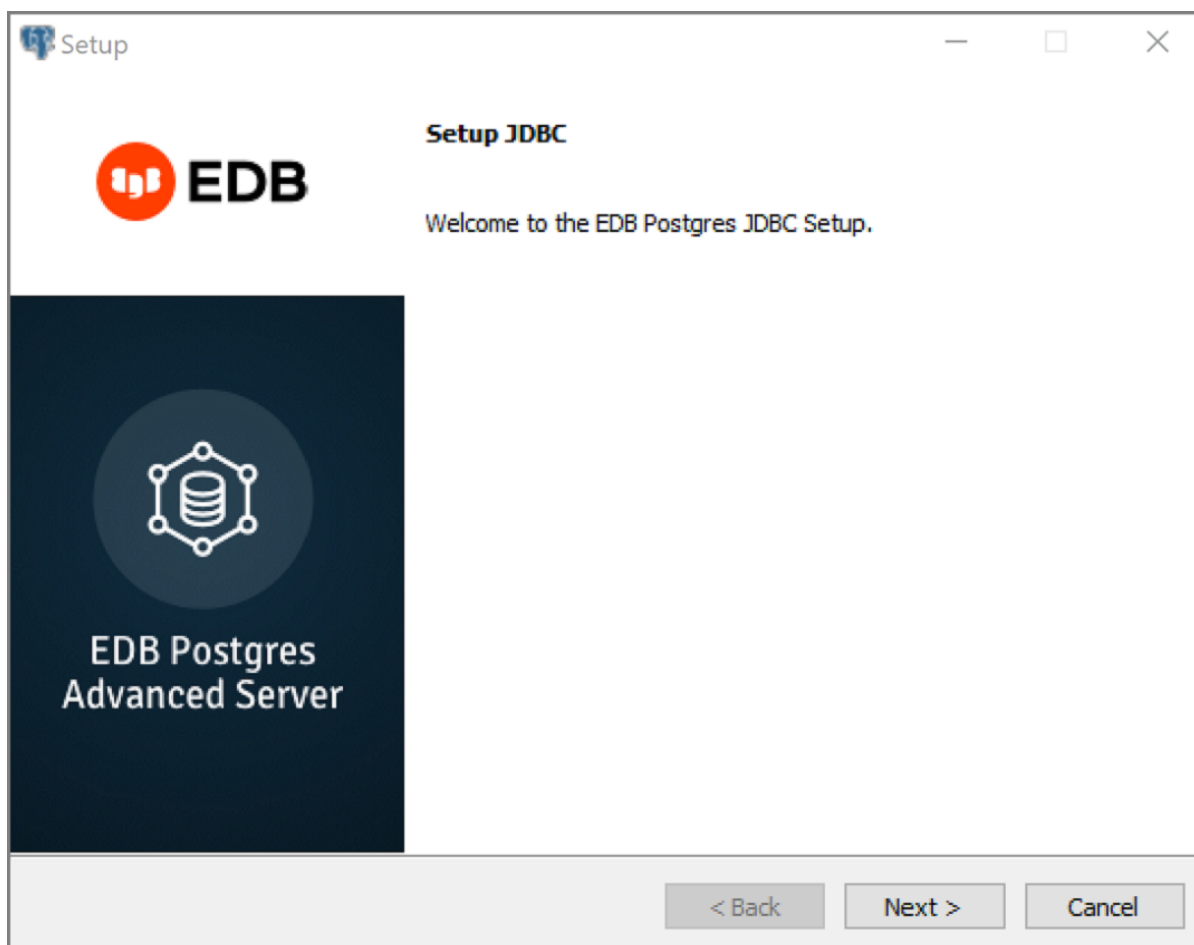


図 1: The JDBC Connector Installation wizard

JDBC コネクタの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

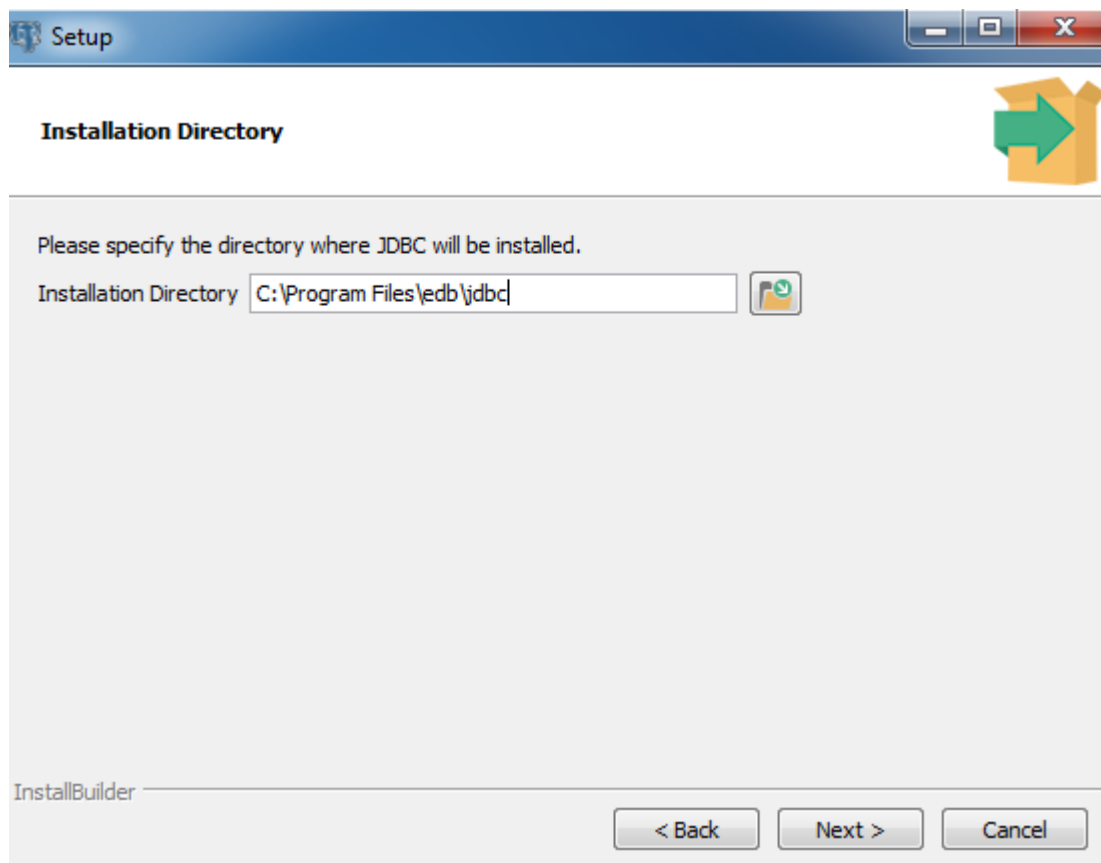


図 2: The Installation dialog

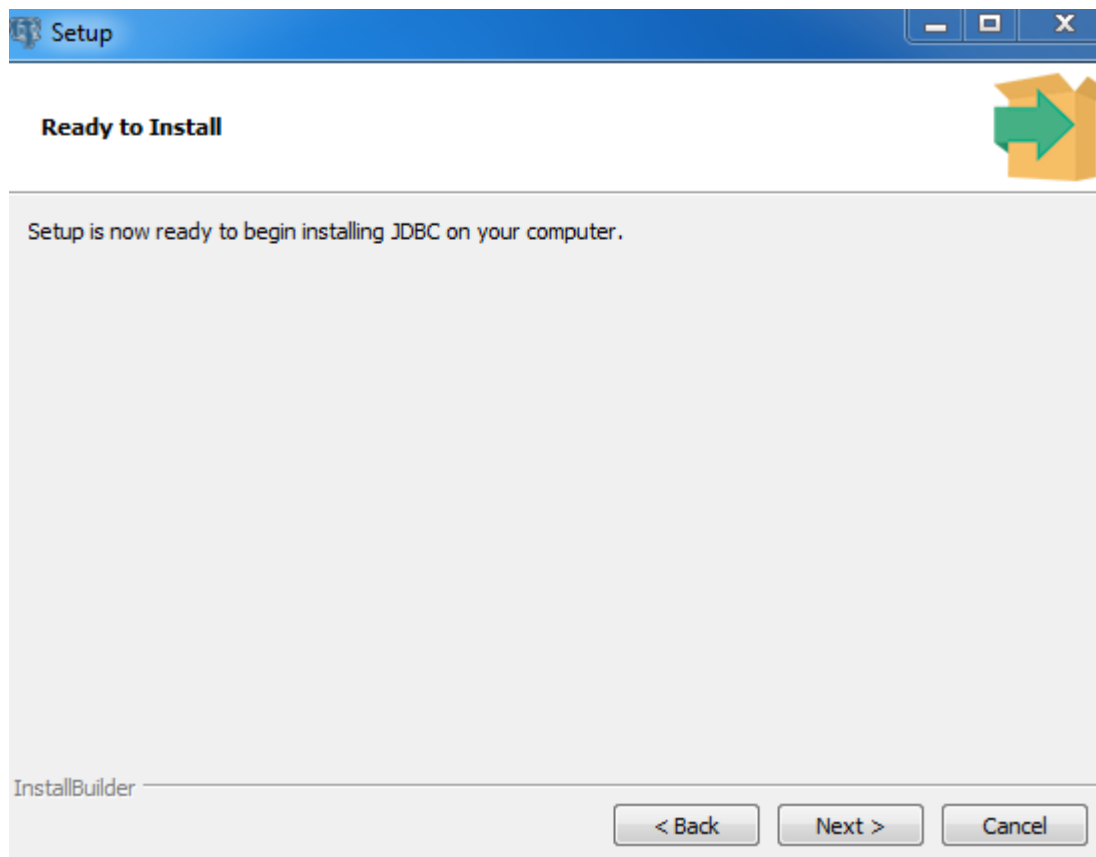


図 3: The Ready to Install dialog

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

インストールが完了しました

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

StackBuilder Plus を使用して、既存の Advanced Server インストールで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。

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。

コネクタインストーラの選択

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

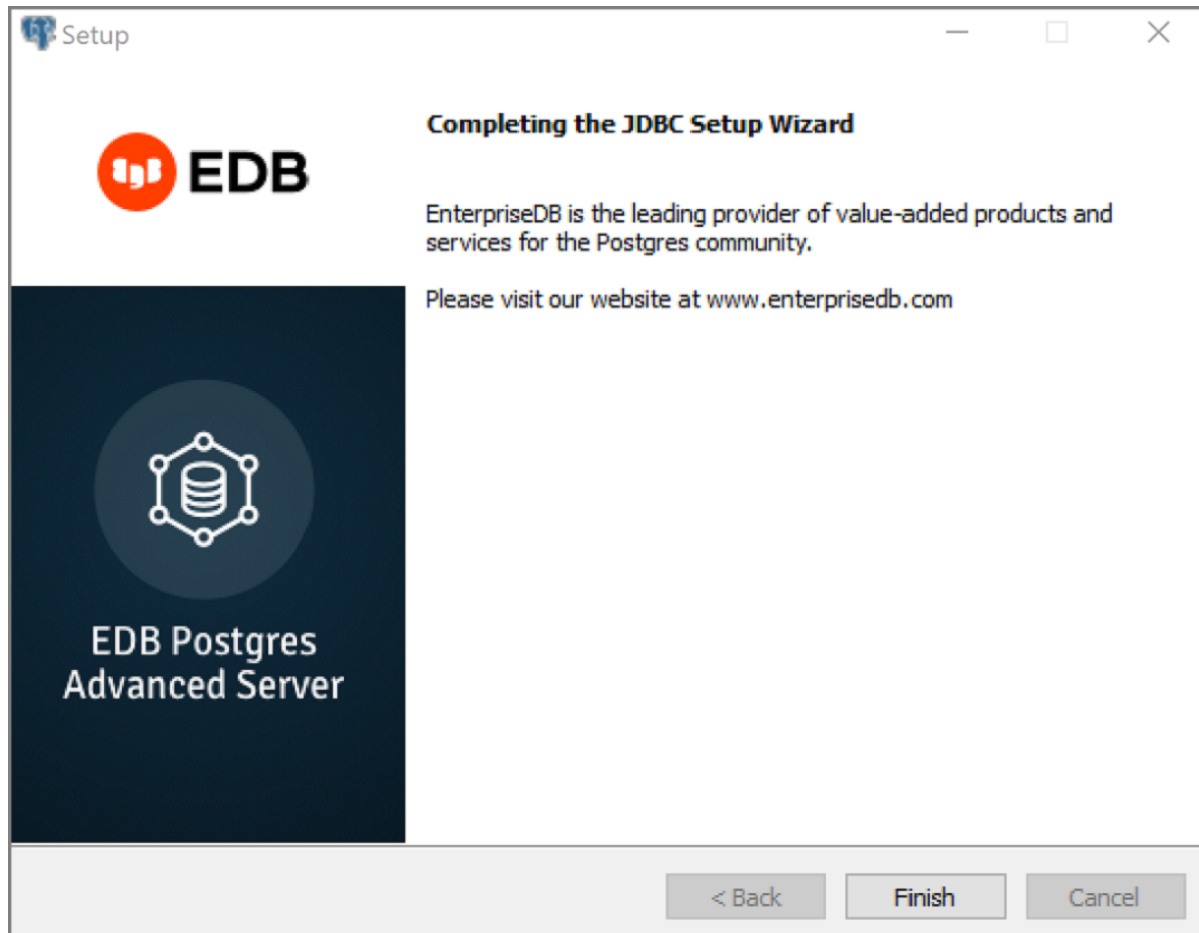


図 4: The installation is complete

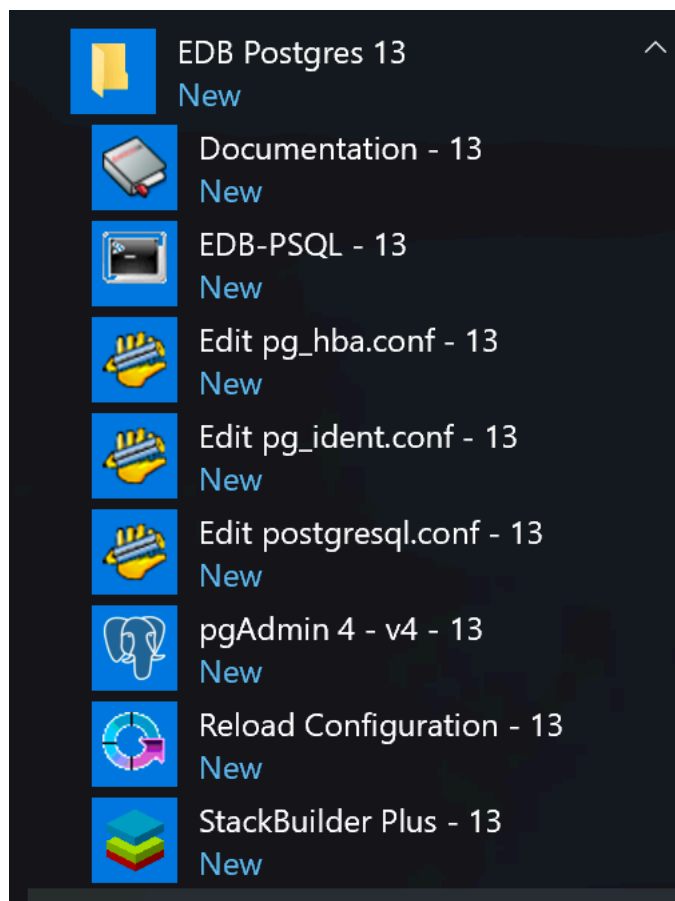


図 5: Starting StackBuilder Plus

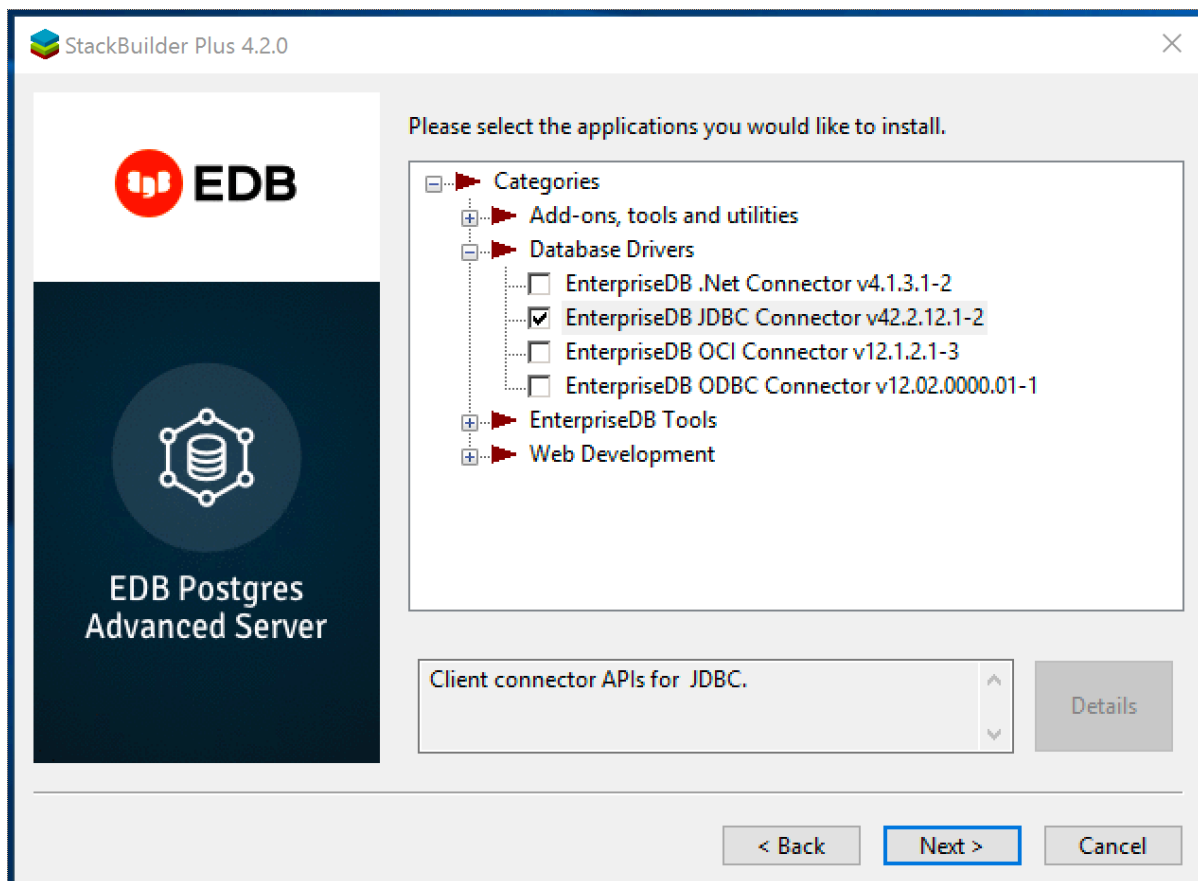


図 6: Selecting the Connectors installer

4.3 Configuring the EDB Postgres Advanced Server JDBC Connector

</div>

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

JDBC ドライバを Java で使用できるように make には、使用している JDBC バージョンの適切な java .jar ファイルを `$java_home/jre/lib/ext` ディレクトリにコピーするか、.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`

4.4 Upgrading a Linux installation

既存の JDBC コネクタ RPM インストールがある場合、yum または dnf を使用して、リポジトリ構成ファイルをアップグレードし、より新しい製品バージョンに更新できます。edb.repo ファイルを更新するには、スーパーユーザ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 リポジトリにアクセスできるようにします。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ソフト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

</div>

Java と EDB JDBC コネクタを配置すると、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、クエリー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());
        }
    }
}
```

(次のページに続く)

(前のページからの続き)

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

この例はシンプルですが、Java アプリケーションから EDB Postgres Advanced Server データベースと対話するために必要な基本的な手順を示しています。

JDBC ドライバをロードします。 - 接続プロパティを構築します。 - データベースサーバに接続します。 SQL ステートメントを実行します。 - 結果セットを処理します。 - クリーンアップ。 - 発生したエラーを処理します。

loading_the_advanced_server_jdbc_connector データベースへの接続実行中
_sql_statements_through_statement_objects 検索_results_from_a_resultset_object freeing_resources handling_errors

</div>

5.1 Loading the EDB Postgres Advanced Server JDBC Connector

</div>

EDB Postgres Advanced Server JDBC ドライバは Java で記述され、コンパイル済みの Java Archive (JAR) ファイルのフォームで配布されます。 `Class.forName()` メソッドを使用して、ドライバをロードします。 `forName()` メソッドは、ランタイムに 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 ドライバの名前付けは `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ディングした後、ドライバは、現在のシステムにインストールされているすべての JDBC ドライバの管理を担当する別の JDBC クラス (`DriverManager` 名前付け) に自分自身を登録します。

JVM が名前付けドライバを見つけることができない場合、`ClassNotFoundException` 例外 (プログラムの終わり近くで `catch` ブロックでインターセプトされます) を報告します。 `DriverManager` は、マルチプルの JDBC ドライバオブジェクトをハンドル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

</ div>

ドライバが `DriverManager` に自分自身をロードして登録した後、`ListEmployees` クラスは、次のコードフラグメントに示すように、データベースサーバ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` でスタートします。 `DriverManager` クラスは、データベースへの接続を作成する `getConnection()` という静的メソッドを提供します。 `getConnection()` メソッドを呼び出すとき、 `DriverManager` はデータベースへの接続に使用する JDBC ドライバ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 は次の一般的なフォーマットを取ります。

```
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）は `edb` です。

Advanced Server JDBC URL は、次のいずれかの形式を取ります。

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

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

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

次の表に、さまざまな接続パラメーターを示します。

Name	Description
host	サーバーのホスト名前。デフォルトはローカルホストです。
port	サーバーがリッ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	データベース名前。

additional_connection_properties preferring_synchronous_secondary_database_servers

</ div>

5.2.1 Additional connection properties

</ div>

EDB Postgres Advanced Server JDBC ドライバは、標準の接続パラメーターに加えて、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 `sslfactory=org.postgresql.ssl.NonValidatingFactory` の設定の組み合わせにより、接続が暗号化されますが、SSL 証明書は検証されません。証明書のバリデーションを強制するには、`Custom SSLSocketFactory` を使用する必要があります。`Custom SSLSocketFactory` の記述の詳細については、[the PostgreSQL JDBC driver documentation](#) を参照してください。

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

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

追加の接続プロパティの一部を次の表に示します。

Name	Type	Description
user	String	接続を確立するデータベースユーザ。
password	String	データベースユーザーのパスワード。
ssl	Boolean	認証済みの暗号化された SSL 接続を要求します。
charSet	String	`charSet` の値は、データベースとの間で送受信されるデータに使用されるキャラクタセットを決定します。
prepareThreshold	Integer	`prepareThreshold` の値は、サーバサイドの準備済みステートメントに切り替える前に必要な `Prepared-Statement` の実行回数を決定します。デフォルトは 5 です。

次のページに続く

表 2 – 前のページからの続き

Name	Type	Description
loadBalanceHosts	Boolean	デフォルトモード（無効）では、ホストは指定されたオーダーで接続されます。有効な場合、ホストは適切な候補のセットからランダムに選択されます。
targetServerType	String	必要な状態のサーバーのみへの接続を開くことができます。許可される値は、 <i>any</i> 、 <i>primary</i> 、 <i>secondary</i> 、 <i>preferSecondary</i> 、および <code>`preferSyncSecondary`</code> です。現在、プライマリ/二次的な区別は、サーバーが書き込みを許可しているかどうかを観察することによって行われます。値 <code>`preferSecondary`</code> は、使用可能なセカンダリがある場合はセカンダリへの接続を試み、それ以外の場合は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 <code>`preferSyncSecondary`</code> をサポートします。これにより、同期的セカンダリまたはアクティブな同期的セカンダリがない場合はプライマリのみに接続できます。
skipQuotesOnReturning	Boolean	<code>`true`</code> に設定すると、 <code>`RETURNING`</code> 句の列名は引用されません。これにより、列名前の大文字と小文字を区別した比較が不要になります。 <code>`false`</code> （デフォルト設定）に設定すると、列名が引用されます。

5.2.2 Preferring synchronous secondary database servers

</ div>

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用の同期的スタンバイサーバを優先するため、非同期サーバーは無視されます。

接続 URL 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の後にポート番号が続きます。**primary** が名前の場合、Java プログラムを実行しているホストの `/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にプライマリデータベースサーバを指定できます。二次的なデータベースサーバの前に置く必要はありません。

secondary_n:port_n

スタンバイ二次的な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の後にポート番号が続きます。**secondary_n** が名前の場合、Java プログラムを実行しているホストの `/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 `make` のデータベースの名前。

以下は、接続 URL の例です。

```
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でプライマリデータベースサーバを指定します。
- Java プログラムが使用するデータベースにアクセスの接続は、最初に同期的二次的なで試行されます。二次的なサーバーは読み取り専用操作に使用できます。
- 非同期モードで実行されているサーバーへの接続は試行されません。
- 接続試行のオーダーは、`loadBalanceHosts` 接続プロパティによって決定されます。デフォルト設定である無効にすると、接続リストで指定された左から右のオーダーで接続が試行されます。有効にすると、接続の試行がランダムに行われます。
- 同期的二次的なに接続できない場合、プライマリデータベースサーバ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がアクティブでない場合、接続の試行は失敗します。

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

プライマリおよび二次的なデータベースサーバーの構成の概要

プライマリおよび二次的なデータベースサーバーを構成するプロセスは、PostgreSQL の文書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。

プライマリから二次的な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法について説明) またはセクション 25.3.3、[ベースバックアップの作成低レベル API](#) セクション 25.3 *

[The PostgreSQL Core Documentation](#) の[継続的アーカイブとポイントインタイムリカバリ \(PITR\)](#)

ホットスタンバイで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：プライマリおよび二次的なデータベースサーバー

次の例では、次のとおりです。

- プライマリデータベースサーバは、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の名前付けは `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の名前付けは `secondary2` 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、すべての二次的な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 `enterprisedb` が、192.168.2.22 の `secondary1` データベースサーバ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の両方のレプリケーションデータベースユーザ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

`pg_hba.conf` ファイルとともに `postgresql.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、`secondary1` のホスト 192.168.2.22 で次のコマンドを呼び出して、データベースサーバ `secondary1` を作成します。

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

二次的なデータベースサーバ `/opt/secondary1` では、`recovery.conf` ファイルがデータベースクラスタに生成されます。これは、`primary_conninfo` 文字列のパートとして `application_name=secondary1` 設

定を追加し、`pg_basebackup` によって自動的に生成される他の不要なオプションの一部を削除することにより、次の例で編集されました `standby_mode = 'on'` パラメータの使用にもノートしてください。

```
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前 `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` パラメータに含まれている必要があります。

二次的なデータベースサーバ（`secondary2`）は、プライマリが使用する同じホスト上に別の方法で作成されます。

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

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

二次的なデータベースサーバ `/opt/secondary2` で、データベースクラスタに `recovery.conf` ファイルを作成します。次の例に示すように、`application_name=secondary2` 設定は `primary_conninfo` 文字列のパートです。また、`standby_mode = 'on'` パラメータを必ず含めてください。

```
standby_mode = on
primary_conninfo = user=enterprisedb password=password host=localhost port=5444
→application_name=secondary2
```

アプリケーション名前 `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` パラメータに含まれている必要があります。

二次的なデータベースサーバーの `postgresql.conf` ファイルの設定パラメータ設定が適切に設定されていることを保証する必要があります（特に `hot_standby=on`）。

注釈: EDB Postgres Advanced Server v12 では、`recovery.conf` ファイルは無効になりました。 `standby.signal` ファイルに置き換えられています。その結果、`primary_conninfo` は `recovery.conf` ファイルから `postgresql.conf` ファイルに移動されます。 `standby.signal` ファイルの存在は、スタンバイモードで実行するようにクラス

ターに通知します。 *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持しようとしても、サーバーはスタートに失敗し、エラーを報告します。

パラメータ `standby_mode=on` も EDB Postgres Advanced Server v12 から削除され、 `trigger_file` パラメータ名前が `promote_trigger_file` に変更されました。

次の表は、二次的な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の基本的な `postgresql.conf` 設定パラメータ設定を示しています。

Parameter	Primary	Secondary	Description
<code>archive_mode</code>	<code>on</code>	<code>off</code>	アーカイブストレージに送信された完了した WAL セグメント。
<code>archive_command</code>	<code>cp %p /archive_dir/%f</code>	<code>n/a</code>	完了した WAL セグメントをアーカイブします。
<code>wal_level</code> (9.5 or prior)	<code>hot_standby</code>	<code>minimal</code>	WAL セグメントに書き込まれた情報。
<code>wal_level</code> (9.6 or later)	<code>replica</code>	<code>minimal</code>	WAL セグメントに書き込まれた情報。
<code>max_wal_senders</code>	<i>n</i> (positive integer)	<code>0</code>	スタンバイサーバからの最大同時接続。
<code>wal_keep_segments</code>	<i>n</i> (positive integer)	<code>0</code>	スタンバイサーバ用に保持する過去のログセグメントの最小数。
<code>syn-chronous_standby_names</code>	<i>n</i> *(<i>*secondary1, secondary2,...</i>)	<code>n/a</code>	同期レプリケーションのスタンバイサーバのリスト。同期レプリケーションを有効にするには、存在する必要があります。これらは、各スタンバイサーバの <code>`recovery.conf`</code> ファイルの <code>`primary_conninfo`</code> パラメータの <code>`application_name`</code> オプションから取得されます。
<code>hot_standby</code>	<code>off</code>	<code>on</code>	クライアントアプリケーションは、スタンバイモードで二次的なサーバーに接続してクエリを実行できます。

二次的なデータベースサーバ（ `secondary1` ）が開始されます。

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

二次的なデータベースサーバ（ `secondary2` ）が開始されます。

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

二次的なデータベースサーバーが同期的モードで正しくセットアップされていることを保証には、プライマ

リデータベースサーバで次のクエリーを使用します。sync_state 列には、アプリケーション secondary1 および secondary2 が同期としてリストされます。

```
edb=# SELECT username, application_name, client_addr, client_port, sync_state FROM pg_stat_replication;
```

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

(2 rows)

接続 URL は次のとおりです。

```
String url = "jdbc:edb://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、優先同期的二次的な接続オプションにより、次のプログラムに示すように、secondary1 で最初の使用が試行され、secondary1 がアクティブでない場合は secondary2、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、接続先の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();
```

(次のページに続く)

(前のページからの続き)

```

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

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

```

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

```

$ java InetServer
192.168.2.22/32:5445
Command successfully executed

```

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

```

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

$ java InetServer
192.168.2.24/32:5446
Command successfully executed

```

ケース 3: secondary2 もシャットダウンすると、192.168.2.24 ポート 5444 の 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

</div>

EDB Postgres Advanced Server JDBC コネクタドライバをローディングしてサーバーに接続すると、サンプルアプリケーションのコードは JDBC Statement オブジェクトを構築し、SQL クエリーを実行し、結果を表示します。

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

- Statement オブジェクトを使用して、パラメーターなしでシンプル 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に使用されます。
- CallableStatement オブジェクトは、データベースストアドプロシージャの呼び出しを実行に使用されます。

SQL ステートメントを実行する前に、Statement オブジェクトをコンストラクト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 オブジェクトの作成を依頼します。

Statement オブジェクトは、さまざまなタイプの SQL ステートメントを実行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 SQL ステートメントを必要とします。executeQuery() は、ResultSet オブジェクトのクエリーからデータを返します。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 を結果、ResultSet は返しません。

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

CallableStatement オブジェクトを使用して、パラメータ値をストア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、実際の引数の値を指定できます。ステートメント内の各 `IN` または `INOUT` パラメータマーカの値を設定する必要があります。

序数表記を使用して `CallableStatement` オブジェクトに値を渡す場合、パラメータタイプに対応するセッターメソッドを使用します。例、`STRING` 値を渡すときは、`setString` setter メソッドを使用します。ステートメント (?) の各パラメータマーカは、序数値を表します。順序パラメータを使用する場合、プロシージャ定義で仮引数が指定されているオーダーで、実際のパラメータ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表記を使用することもできます。名前付きパラメータ表記を使用すると、プロシージャに必要なパラメータのみに値を指定でき、受け入れ可能なデフォルト値を持つパラメータは省略できます。名前付けパラメータは任意のオーダーで指定することもできます。

名前付け表記を使用する場合、各パラメータ名前は `DatabaseMetaData.getProcedureColumns` メソッドの呼び出しによって返される `COLUMN_NAME` に対応する必要があります。ステートメント呼び出しに名前付けパラメータを含める場合は、`=>` 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` パラメータマーカを識別します。

例

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

```
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を提供しています。

```
CallableStatement cstmt = con.prepareCall("{CALL hire_emp(?,?,?,?,?,?)}");
//Bind a value to each parameter.
cstmt.setString(1, "SMITH");
cstmt.setInt(2, 8888);
cstmt.setString(3, "Sales");
cstmt.setInt(4, 5500);
```

(次のページに続く)

```
cstmt.setDate(5, Date.valueOf("2016-06-01"));
cstmt.setInt(6, 7566);
cstmt.setInt(7, 30);
```

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

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

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

5.4 Retrieving results from a ResultSet object

</div>

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プライマリストレージメカニズムです。各 **ResultSet** オブジェクトには、**ResultSetMetaData** オブジェクトのフォームでデータとメタデータの両方が含まれています。**ResultSetMetaData** には、SQL コマンドによって返される結果に関する有用な情報（列名、列数、行数、列長さなど）が含まれます。

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ために、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。さまざまな **getter** メソッドがあります。各メソッドは、特定のタイプの値を返します。例、**getString()** メソッドは **STRING** 型を返し、**getDate()** メソッドは **Date** 型を、**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を保持します。**executeQuery()** メソッドが **ResultSet** を返す場合、ポインタは最初の行の前に配置されます。ポインタを移動する前にアプリケーションが **getter** メソッドを呼び出すと、**getter** メソッドは失敗します。次の（または最初の）行に進むには、**ResultSet** の **next()** メソッドを呼び出します。**ResultSet.next()** はブール型のメソッドです。**ResultSet** に別の行がある場合は **TRUE** を返し、最後の行を超えて移動した場合は **FALSE** を返します。

ポインタを最初の行に移動した後、サンプルアプリケーションは **getString()** **getter** メソッドを使用して最初の列の値を取得し、その値を出力します。**ListEmployees** はループで **rs.next()** と **rs.getString()**

を呼び出すため、結果セットの行を処理します。rs.next() は、rs.next() がポインタを最後の行を超えて移動し、FALSE を返すときにループを終了します。

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

ResultSet インタフェースを使用する場合：

- 値を読み取る前に next() を呼び出す必要があります。別の行が使用可能な場合、next() は true を返し、処理のために行を準備します。JDBC 仕様では、アプリケーションは ResultSet の行に 1 回だけアクセスする必要があります。現在、EDB Postgres Advanced Server JDBC ドライバを使用すると、必要な回数だけフィールドにアクセスできますが、このルールを守るのが最も安全です。
- ResultSet の使用が終了したら、close() メソッドを呼び出して、そのオブジェクトが保持しているリソースをフリーします。

5.5 Freeing resources

</ div>

すべての JDBC オブジェクトはリソースを消費します。例、ResultSet オブジェクトには、クエリー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リソースが不要になったら、これらのリソースをフリー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

</ div>

外部リソース（データベースサーバなど）に接続すると、エラーが発生します。コードには、これらのエラーをハンドルする方法を含める必要があります。JDBC と EDB Postgres Advanced Server JDBC コネクターの両方が、さまざまなタイプのエラーハンドリングを提供します。

ListEmployees class example は、try/catch

ブロックを使用してエラーをハンドルする方法を示しています。

JDBC オブジェクトがエラー（`SQLException` 型または `SQLException` から派生した型のオブジェクト）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());
```

例、指定したホストに存在しないデータベースにサーバーが接続しようとする、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。ユーザー定義のメッセージはキューに保存され、キューのコレクションはキューテーブルに保存されます。依存するキューを作成する前に、まずキュー表を作成する必要があります。

サーバサイドでは、`DBMS_AQADM` パッケージのプロシージャがメッセージキューとキューテーブルを作成および管理します。`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。

クライアントサイドでは、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 サーバー側のセットアップ

JMS ベースの Java アプリケーションで高度なキュー機能を使用するには、まずユーザー定義型、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でキューをスタートします。Java アプリケーションでは、EDB-PSQL または EDB- JDBC JMS API のいずれかを使用できます。

6.2.1 EDB-PSQL を使用する

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 データタイプを指定するには、ユーザー定義型を作成します。この例では、`mytype` 名前付け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

キューテーブルは、同じペイロードタイプのマルチプルのキューを保持できます。この例では、MSG_QUEUE_TABLE 名前付け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

この例では、MSG_QUEUE_TABLE というテーブルに MSG_QUEUE 名前付け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

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スタートします。

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

次の JMS API 呼び出しは、EDB-PSQL を使用して実行される同じステップを実行します：-EDB EDB\_
→Postgres Advanced Server データベースに接続します-ユーザー定義タイプを作成します-キュー
テーブルとキューを作成します-キューを開始します

```
```Text
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();
```

6.3 クライアント側の例

ユーザー定義タイプを作成し、その後にキュー表とキューを作成した後、キューをスタートします。次に、EDB-JDBC ドライバーの JMS API を使用して、メッセージをキューに登録またはデキューできます。

Java プロジェクトを作成し、edb-jdbc インストレーションディレクトリから edb-jdbc18.jar をライブラリに追加します。

作成したタイプに対応する 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;
    }

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

(次のページに続く)

```

/**
 * @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 メッセージをエンキューおよびデキューする

メッセージをエンキューおよびデキューするには：

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。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 メッセージをキューに入れる

メッセージをキューに入れるには：

1. セッションから `EDBJmsMessageProducer` を作成します。エンキューメッセージ.3 を作成します。`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：

1. セッションから `EDBJmsMessageConsumer` を作成します。`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("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。ユーザー定義タイプ、キューテーブル、およびキューは EDB-PSQL を使用して作成され、キューが開始されます。

```
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;
```

(次のページに続く)

```
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.JMSException;
import javax.jms.Message;
import javax.jms.Session;

public class JMSClient {

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

        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);

            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);
```

(前のページからの続き)

```

        queue.setDequeue_mode(DequeueMode.BROWSE);
        queue.setType("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();
        }
    }
}
}
}

```

この例は、ユーザー定義タイプ、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;
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;
    }
}

```

(次のページに続く)

```
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");
    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");
```

(次のページに続く)

(前のページからの続き)

```
        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

</div>

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()` はクエリーステートメントを実行ように設計されているため、クエリー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）を実行ために使用する 2 番目のメソッドを提供します。行のコレクションを返す代わりに、`executeUpdate()` メソッドは、実行する SQL コマンドの影響を受ける行の数を返します。

`executeUpdate()` メソッドのシグネチャは次のとおりです。

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

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

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。

excla_to_rst_1

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

(次のページに続く)

(前のページからの続き)

```
}  
}
```

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

```
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る行の数を返します（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します）。

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

`executeUpdate()` がエラーなしで戻る場合、`System.out.println` の呼び出しは、影響を受けた行の数を示すメッセージをユーザに表示します。

```
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` ブロックは適切なエラーメッセージをユーザに表示します。

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

`executeUpdate()` は、結果セットを結果ない 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 `PrepareStatements` を使用する必要があります。

7.2 PreparedStatements を使用して SQL コマンドを送信する

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

以下は、従業員 ID と従業員名前を受け入れ、その従業員情報を `emp` テーブルに挿入する `PreparedStatement` の呼び出しを示しています。

```
public void AddEmployee(Connection con)  
{  
    try {
```

(次のページに続く)

(前のページからの続き)

```

    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 ステートメントで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（それぞれがマークで表される）を含む INSERT ステートメントを示しています。

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

パラメーター化された SQL ステートメントを手にすると、`AddEmployee()` メソッドは `Connection` オブジェクトにそのステートメントを準備して `PreparedStatement` オブジェクトを結果ように依頼できます。

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

この時点で、`PreparedStatement` は INSERT ステートメントを解析および計画しましたが、テーブルに追加する値がわかりません。`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の値を指定する必要があります。`setObject()` には2つの引数が必要です。-パラメータ番号。パラメーター番号1は最初のマークに対応し、パラメータ番号2は2番目のマークに対応します。-プレースホルダの代わりとなる値。

`AddEmployee()` メソッドは、ユーザに従業員 ID と名前の入力を求め、ユーザが指定した値で `setObject()` を呼び出します。

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

次に、`PreparedStatement` オブジェクトにステートメントの実行を要求します。

```
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した場合、エラーハンドリングコードはエラーメッセージを表示します。

SQL コマンドを送信する `PreparedStatement` を使用したいいくつかのシンプル構文例は次のとおりです。

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、データベースに新しいテーブルを追加するには：

```
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

</div>

少し余分な作業を行うと、プログラムにグラフィカルユーザインタフェースを追加できます。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、クエリーによって返されたデータをその JTable にコピーする Java アプリケーションの作成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()* ファンクション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);
        }
    }
}
```

(次のページに続く)

```

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

```

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

```

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

```

`showEmployees()` メソッドは、呼び出し元から `Connection` オブジェクトが提供されることを想定しています。`Connection` オブジェクトは、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 クエリーを実行します。

```

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（結果セット内の行と列の数、各列のデータタイプ、各列の名前など）を記述するメタデータが含まれます。`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));
}
```

列ヘッダーを配置すると、`showEmployees()` は `ResultSet` から行を抽出し、それを新しいベクトル（`rows` 名前付け）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` ベクトルの各エントリーには、その行のデータ値を含むベクトルが含まれます。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` ベクトルを作成した後、プログラムは（`rs.next()` を呼び出して）`ResultSet` の行を読み取ります。各行の各行について、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);
}
```

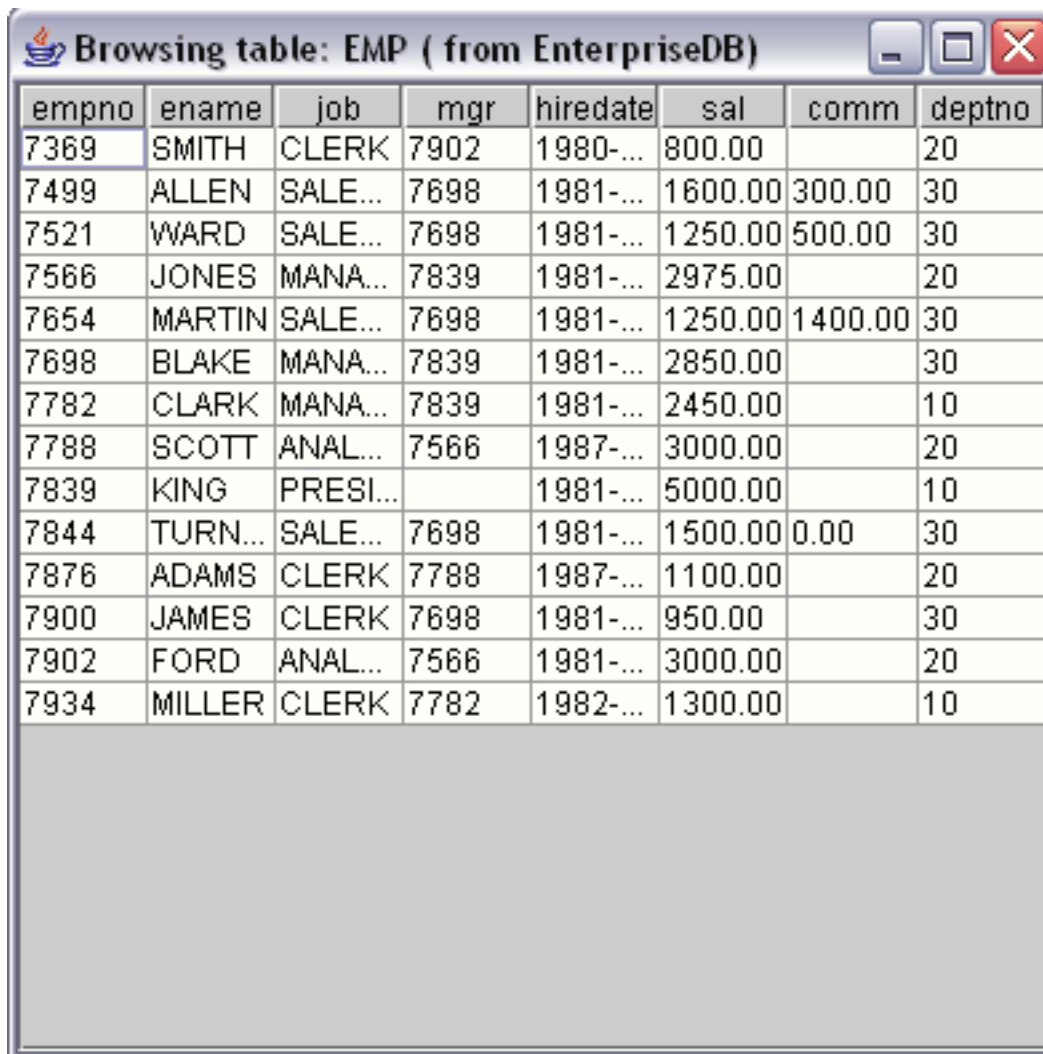
この時点で、ベクトル（`labels`）には列ヘッダーが含まれ、2 番目の 2 次元ベクトル（`rows`）にはテーブルのデータが含まれます。これで、ベクトルと `JTable` を保持する `JFrame` から `JTable` を作成できます。

```
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、適切なメッセージをユーザに表示する `catch` ブロックが含まれています。

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

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



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

</ div>

これらの例は、EDB Postgres Advanced Server JDBC コネクタの高度な諸機能の一部を示しています。

```

ucing_client_side_resource_requirements          using_preparedstatements_to_send_sql_commands
execution_stored_procedures          using_ref_cursors_with_java          using_bytea_data_with_java          us-
ing_object_types_and_collections_with_javasynchronous_notification_handling_with_noticelistener

```

</ div>

9.1 Reducing client-side resource requirements

</ div>

EDB Postgres Advanced Server JDBC ドライバは、SQL クエリーの結果を **ResultSet** オブジェクトとして取得します。クエリーがバッチ **ResultSet** を使用してラージの行を返す場合：

- 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することで時間を節約します。
- 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小さくすると、ドライバはネットワークを介して（サーバーからクライアントに）**ResultSet** 全体をコピーしません。代わりに、ドライバ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、ドライバはサーバーから次のバッチ行をフェッチします。

すべての状況でバッチ結果セットを使用できるわけではありません。次の制限を順守しないと、ドライバは **ResultSet** 全体を一度にフェッチするように静かにフォールバックします。

- 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** タイプの **ResultSet** タイプ（デフォルト）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** を介してのみ前進できます。
- クエリーは、単一の SQL ステートメントで構成する必要があります。

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、データの取得が高速化され、クライアント側アプリケーションに必要なリソースが削減されます。次のコードは、バッチサイズが 5 行に制限された `Statement` オブジェクトを作成します。

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

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

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

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

`conn.setAutoCommit(false)` の呼び出しにより、最初の行を取得する機会が得られる前にサーバーが `ResultSet` を閉じないことが保証されます。`Connection` を準備したら、`Statement` オブジェクトをコンストラクトできます。

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

次のコードは、クエリー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` オブジェクトの行について、`println()` の呼び出しは `a row was returned` を出力します。

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

`ResultSet` にはテーブル内のすべての行が含まれていますが、サーバーから一度に 5 行しかフェッチされません。クライアントの観点からビューと、`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最初の行を短時間で結果とすることができることです。

9.2 Using PreparedStatements to send SQL commands

</ div>

多くのアプリケーションは同じ SQL ステートメントを繰り返し実行し、各反復の間にステートメント内の 1 つ以上のデータ値を変更します。Statement オブジェクトを使用して SQL ステートメントを繰り返し実行する場合、サーバーはステートメントを毎回解析、計画、および最適化する必要があります。JDBC は、この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 と従業員名前を受け入れ、その従業員情報を emp テーブルに挿入する PreparedStatement の呼び出しを示しています。

```
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 ステートメントで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。以下に、2 つのプレースホルダー（それぞれがマークで表される）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すると、AddEmployee() メソッドは Connection オブジェクトにそのステートメントを準備して PreparedStatement オブジェクトを結果ように依頼できます。

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

この時点で、PreparedStatement は INSERT ステートメントを解析および計画しましたが、テーブルに追加する値がわかりません。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の値を指定する必要があります。setObject() には 2 つの引数が必要です。

- パラメータ番号。パラメーター番号 1 は最初のマークに対応し、パラメータ番号 2 は 2 番目のマークに対応します。
- プレースホルダを置き換える値。

AddEmployee() メソッドは、ユーザに従業員 ID と名前の入力を求め、ユーザが指定した値で setObject() を呼び出します。

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

次に、PreparedStatement オブジェクトにステートメントの実行を要求します。

```
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した場合、エラーハンドリングコードはエラーメッセージを表示します。

9.3 Executing stored procedures

</div>

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

ストアードプロシージャは、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、ストアードプロシージャを使用してバッチプログラムでデータを操作することも効率的です。

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ー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) に使用される変数数のパラメーターを持つことができます。

JDBC でストアー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。大括弧はオプションパラメータを示します。それらはコマンド構文のパートではありません。

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

結果パラメータ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([?, ?, ...])}
```

各マークは、パラメータのプレースホルダとして機能します。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 コードがマッチする必要があるかどうかを判別します。

シンプル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予期せず、情報を結果ません。

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

以下は、increaseSalary プロシージャを呼び出す方法を示しています。

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

Java アプリケーションからストア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 パラメータを使用してストアドプロシージャを実行する が示すように、アプリケーションは通常、prepareCall()

の呼び出しと execute() の呼び出しの間でパラメータ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();
```

このストアードプロシージャ（`increaseSalary`）は IN パラメータを予期せず、呼び出し側に情報を結果なかったため（OUT パラメータを使用）、プロシージャの呼び出しは `CallableStatement` オブジェクトを作成し、そのオブジェクトの `execute()` メソッドを呼び出すことです。

IN パラメータを使用してストアードプロシージャを実行する

次の例のコードは、最初に `empInsert` 名前付け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;
```

以下は、Java からストアードプロシージャを呼び出す方法を示しています。

```
public void CallExample2(Connection con)
{
    try
    {
        Console c = System.console();
        String commandText = "{call empInsert(?,?,?,?,?,?)}";
        CallableStatement stmt = con.prepareCall(commandText);
        stmt.setObject(1, new String(c.readLine("Employee Name :")));
        stmt.setObject(2, new String(c.readLine("Job :")));
    }
}
```

(次のページに続く)

(前のページからの続き)

```

    stmt.setObject(3, new Float(c.readLine("Salary :")));
    stmt.setObject(4, new Float(c.readLine("Commission :")));
    stmt.setObject(5, new Integer(c.readLine("Department No :")));
    stmt.setObject(6, new Integer(c.readLine("Manager")));
    stmt.execute();
}
catch(Exception err)
{
    System.out.println("An error has occurred.");
    System.out.println("See full details below.");
    err.printStackTrace();
}
}

```

コマンド (`commandText`) の各プレースホルダ (?) 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 プレースホルダにバインドします。 `setObject()` を呼び出すたびに、パラメータ番号とそのパラメータ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")));

```

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

OUT パラメータを使用してストアードプロシージャを実行する

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。このプロシージャでは、IN パラメータ (部門番号) が 1 つ必要であり、部門番号に対応する 2 つの OUT パラメータ (部門名前と場所) を返します。

```

CREATE OR REPLACE PROCEDURE deptSelect
(
    p_deptno IN INTEGER,
    p_dname OUT VARCHAR,
    p_loc OUT VARCHAR

```

(次のページに続く)

```

)
AS
DECLARE
    CURSOR deptCursor IS SELECT dname, loc FROM dept WHERE deptno=p_deptno;
BEGIN
    OPEN deptCursor;
    FETCH deptCursor INTO p_dname, p_loc;

    CLOSE deptCursor;
END;

```

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

```

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

```

コマンド（commandText）の各プレースホルダ（?）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 プレースホルダにバインドします。setObject() を呼び出すときは、プレースホルダを（序数で）特定し、そのプレースホルダの代わりに置換する値を提供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 パラメータの JDBC タイプを登録します。JDBC タイプの登録は、registerOutParameter() メソッドを使用して行われます。

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

ステートメントの実行後、CallableStatement getter メソッドは OUT パラメータ値を取得します。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、(ストアードプロシージャの実行後) OUT パラメーターで返された値を取得します。両方の OUT パラメーターが 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 パラメータを使用したストアードプロシージャの実行

次の例のコードは、1 つの IN パラメータ (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、p_hiredate および p_sal) で定義された empQuery 名前付けのストアードプロシージャを作成して呼び出します 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 名前付けのストアードプロシージャを作成します。

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

次のコードは、empQuery プロシージャの呼び出し、IN パラメータの値の提供、および OUT および IN OUT パラメータのハンドリングを示しています。

```
public void CallSample4(Connection con)
{
    try
    {
        Console c = System.console();
        String commandText = "{call empQuery(?,?,?,?,?)}";
        CallableStatement stmt = con.prepareCall(commandText);
        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("Hiredate : " + stmt.getTimestamp(5));
        System.out.println("Salary : " + stmt.getBigDecimal(6));
    }
    catch(Exception err)
    {
        System.out.println("An error has occurred.");
        System.out.println("See full details below.");
        err.printStackTrace();
    }
}
```

コマンド（commandText）の各プレースホルダ（?）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() の呼び出しは、パラメータ番号を指定し、そのプレースホルダの代わりに置き換える値を提供します。

```
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 パラメータの JDBC タイプを登録する必要があります。

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

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

- パラメータに値を割り当てます。
- パラメータのタイプを登録します。
- ゲッターメソッドで返された値を取得します。

9.4 Using REF CURSORS with Java

</div>

REF CURSOR は、OPEN ステートメントによって返されるクエリー結果セットへのポインタを含むカーソル変数です。静的カーソルとは異なり、REF CURSOR は特定のクエリーに結び付けられていません。異なるクエリを含む OPEN ステートメントを使用して、同じ REF CURSOR 変数を何度でも開くことができます。毎回、そのクエリー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可能になります。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、強く型付けされた変数と弱く型付けされた REF CURSOR 変数の両方の宣言をサポートしています。強く型付けされたカーソルは、予期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ーでは、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ーでカーソルを開くと、サーバーは例外を結果ます。一方、弱い型指定のカーソルは、任意の形状の結果セットで機能します。

強く型付けされた REF CURSOR を宣言するには：

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

弱く型付けされた 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

以下に示すストアードプロシージャ（`getEmpNames`）は、サーバー上に 2 つの REF CURSOR 変数を構築します。最初の REF CURSOR には、`emp` テーブルの委託された従業員のリストが含まれています。2 番目の REF CURSOR には、`emp` テーブル内のサラリーマン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;
```

以下に示す `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た名前を表示します。

```
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)
```

(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();
}
}
```

CallableStatement は、各 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 パラメータとして返されます。

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

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

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

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

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

同じゲッターメソッドが 2 番目のパラメータから ResultSet を取得し、RefCursorExample がそのカーソルを反復処理して、給与を受け取る各従業員の名前を出力します。

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

9.5 Using BYTEA data with Java

</div>

BYTEA データタイプは、バイトシーケンスでバイナリ文字列を格納します。デジタル画像と音声ファイルは、しばしばバイナリデータとして保存されます。EDB Postgres Advanced Server は、BYTEA データタイプを使用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、BYTEA データを EDB Postgres Advanced Server データベースに保存し、そのデータを取得する方法を示しています。

まず、以下は BYTEA 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 つの列が含まれます。

- 最初の列には従業員の ID 番号（INT 型）が格納され、テーブルのプライマリキーとして機能します。
- 2 番目の列には、従業員の写真が BYTEA フォーマットで保存されます。

```
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、以下は、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 BYTEA データを EDB Postgres Advanced Server に挿入する

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

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

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

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

InsertPic() は、従業員番号とイメージファイルの名前の入力をユーザに求めます。

```
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...");
}
```

(次のページに続く)

(前のページからの続き)

```
return;
}
```

次に、InsertPic() は、ADD_PIC プロシージャを呼び出す CallableStatement オブジェクト (stmt) を準備します。最初のプレースホルダ (?) は、ADD_PIC (p_empno) が予期する最初のパラメータを表します。2 番目のプレースホルダは、2 番目のパラメータ (p_photo) を表します。これらのプレースホルダの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。最初のパラメータは INTEGER 型であるため、InsertPic() は setInt() メソッドを呼び出して p_empno の値を提供します。2 番目のパラメータは 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();
```

すべてうまくいけば、InsertPic() はイメージ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がわかったので、次に、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);
```

(次のページに続く)

(前のページからの続き)

```

        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、ユーザに従業員 ID 番号の入力を求めることから開始します。

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

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

```

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

```

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

```

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

```

プログラムは、写真を保存するファイルの名前をユーザに要求します。

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

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

```

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

</div>

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、一般に配列と呼ばれるコレクションの作成にも使用されます。これは、EDB Postgres Advanced Server データベースにも保存されます。

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

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

ネストした表タイプのコレクションは、CREATE TYPE AS TABLE OF コマンドを使用して作成されます。VARRAY タイプのコレクションは、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 사용되는 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、住所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体には、従業員情報を表示するメソッドが含まれています。

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

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

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

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

        // initialize emp_obj_typ structure
        // 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
```

(次のページに続く)

(前のページからの続き)

```

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

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

```

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

```

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

```

createStruct() は、それぞれ address および emp 名前付けのオブジェクトタイプ addr_object_type および emp_obj_typ のインスタンスを初期化および作成します。

```

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

```

registerOutParameter() の呼び出しは、emp_obj_typ のパラメータタイプ (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 プレースホルダにバインドします。

```

stmt.setObject(1, emp);

```

stmt.execute() の呼び出しは、display_emp() メソッドの呼び出しを実行します。

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

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

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

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

9.6.2 コレクションを使用する

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、コレクションタイプ NUMBER_ARRAY および CHAR_ARRAY を IN パラメーターとして使用し、CHAR_ARRAY を OUT パラメーターとして使用する SPL 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前を連結しパラメータ。結果の連結エントリは、CHAR_ARRAY OUT パラメータで返されます。

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

(次のページに続く)

```
END LOOP;  
RETURN arrEmpIdNames;  
END;
```

以下は、前のファンクションを呼び出して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"});  
  
        // 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();  
        }  
    }  
}
```

concatEmpIdName() ファンクションを呼び出すために CallableStatement オブジェクトが準備されます。

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

`createArrayOf()` は、`empIdArray` および `empNameArray` 名前付け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()` の呼び出しは、OUT パラメータのパラメータタイプ (`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()` ファンクションが呼び出されます。

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

`getArray()` は、ファンクション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

</div>

EDB Postgres Advanced Server JDBC コネクタは、非同期通知ハンドリング機能を提供します。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する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）プログラムの実行の結果として通知がクライアントに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。通知は、SQL ステートメントの実行時にクライアントに配信されます。通知は、CallableStatement オブジェクト、PreparedStatement オブジェクト、またはシンプル Statement オブジェクトなど、あらゆるタイプのステートメントオブジェクトで機能します。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生成する 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
    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;
```

以下に、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.");
  }
}
```

(次のページに続く)

(前のページからの続き)

```

        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 オブジェクトが必要です。

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

NoticeExample() は、前述の 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 インタフェー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 (サーバーで adjustSalary プロシージャを呼び出します) を実行し、ユーザ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() の呼び出しが完了する前に noticeReceived() メソッドを (おそらく何回も) 呼び出します。

```

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

```

(次のページに続く)

(前のページからの続き)

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

JDBC が `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` オブジェクトが作成されます。

各 `Statement` オブジェクトは、`NoticeListeners` のリストを保持します。JDBC ドライバは、サーバーから通知を受け取ると、`Statement` オブジェクトによって保持されているリストを調べます。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()` を呼び出して通知を取得できます。）リストが空でない場合、JDBC ドライバは、リスナーが `Statement` に追加されたオーダーで `SQLWarning` を各リスナに配信します。

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ネイティブサポートを備えています。これには、クライアント・サーバシステムの両方に OpenSSL がインストールされ、ビルド時に PostgreSQL のサポートが有効になっている必要があります。

</ div>

using_ssl scram_compatibility support_for_gssapi_encrypted_connection

</ div>

10.1 Using SSL

</ div>

SSL を使用する場合、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

サーバーの設定クライアントの設定テスト_ssl_jdbc_connection_certificate_authentication_without_a_password
を使用したテスト

</ div>

10.1.1 Configuring the server

</ div>

SSL 用の PostgreSQL または EDB Postgres Advanced Server の構成については、[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できることを保証してください。SSL 接続を確立した場合、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.

edb=#
```

10.1.2 Configuring the client

</ div>

クライアントを SSL 用に構成するために、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。

異なる値を渡すと、SSL 接続パラメーターの動作が異なります。接続パラメータ `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 SSL 接続が使用されます。

`sslmode` 接続パラメータを使用すると、SSL 接続をより適切に制御できます。このパラメータは `libpq` `sslmode` パラメータと同じであり、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することもあります。サーバーは、このホスト/IP アドレスの SSL 接続を受け入れるように構成されており、サーバーがクライアント証明書を認識します。

注釈: このモードでは、JDBC ドライバはすべてのサーバ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、サーバー証明書に保存されている名前と一致 make 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。

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

クライアント証明書、クライアントキー、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。これらのデフォルトは、それぞれ/defaultdir/postgresql.crt、/defaultdir/postgresql.pk8、および/defaultdir/root.crt です。ここで、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 ドライバがサーバーの ID を検証し、「中間者」攻撃を防ぎます。これは、サーバー証明書が信頼できる権限によって署名されていること、および接続先のホストが証明書のホスト名と同じであることを確認することによりこれを行います。

10.1.3 Testing the SSL JDBC connection

</ div>

Java のデフォルトメカニズム（LibPQFactory ではない）を使用して SSL 接続を作成している場合は、サーバー証明書を Java で利用可能に make 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 フォーマットに変換します。

```
$ 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て、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、`-Djavax.net.debug=ssl` を Java コマンドに追加します。

証明書のバリデーションなしで SSL を使用する

デフォルトでは、`SSL=true` と接続 URL パラメータ `sslfactory=com.edb.ssl.NonValidatingFactory` の設定の組み合わせにより、接続が暗号化されますが、SSL 証明書は検証されません。証明書のバリデーションを強制するには、`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

</div>

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 フォーマットに変換します。

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

2. クライアントキーを DER フォーマットに変換します。

```
$ 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

</div>

EDB JDBC ドライバは、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 コネクタリリース 42.2.2.1 以降で利用可能です。JRE / JDK バージョン 1.7 の場合、このサポートは EDB JDBC コネクタリリース 42.2.5 以降で利用できます。

10.3 Support for GSSAPI-encrypted connection

</div>

EDB JDBC ドライバは、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` パラメータは、GSSAPI 暗号化接続を制御します。パラメータには、次の値のいずれかを指定できます。

- **Disable**。GSS 暗号化モードを使用した接続試行を無効にします。
- **Allow**。プレーンテキストで接続を試みます。その後、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み、暗号化された接続の取得に失敗した場合、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を試み、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 ドライバは SSPI を使用して接続を試みます。- Linux では、EDB JDBC ドライバは GSSAPI を使用して接続を試みます。

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

- **auto**。サーバーが要求し、EDB JDBC クライアントが Windows で実行され、SSPI に必要なワッフルライブラリが CLASSPATH にある場合、ドライバは SSPI 認証を試みます。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 が利用可能な場合でも 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。Linux または SSPI が利用できない場合、この認証は失敗します。

10.3.2 SSPI の使用 (Windows のみの環境)

EDB Postgres Advanced Server と JDBC クライアントの両方が Windows にある場合、JDBC ドライバ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");
```

サーバーは SSPI 認証を必要とするため、.. **Note ::**

- **gsslib=sspi** はオプション。-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の資格情報は、ユーザ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 パスが <PATH_DRIVER> であり、ワッフルライブラリパスが <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 クライアントの両方が Linux 上にある場合、JDBC ドライバ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);
```

サーバーは GSSAPI 認証を必要とするため、.. **Note::** **gsslib=gssapi** はオプション。

例

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

次の内容の pgjdbc.conf 名前付け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 サンプルの実行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 GSSAPI として設定された認証を備えた Linux 上にあり、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 認証が構成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");
```

(次のページに続く)

(前のページからの続き)

```
String databaseUrl = "jdbc:edb://pg.myrealm.example:5444/edb?gsslib=gssapi";
con = DriverManager.getConnection(databaseUrl, connectionProps);
```

Kerberos 資格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（`c:\temp` など）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 チケットマネージャを使用するか、`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 \(Linux のみの環境\)](#) で説明されている内容と同じ `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 サンプルの実行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される場合の JDBC コネクタの問題を解決するのにヘルプロギングの使用をサポートします。JDBC コネクタは、JDK 1.4 以降 Java のパートであった `java.util.logging` のロギング API を使用します。 `java.util.logging` については、 [The PostgreSQL JDBC Driver](#) を参照してください。

注釈: 以前のバージョンの EDB Postgres Advanced Server JDBC コネクタは、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 から前進するバージョンで `java.util.logging` を使用するようになりました。古いメカニズムは使用できなくなりました。

Advanced Server JDBC Connector 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 以降では使用できなくなりました。

11.1 ロギング `.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を動的に構成できます（例、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、ランタイムにロギングを簡単に有効/無効にできます。次の例は、ロギングを動的に構成する方法を示しています。

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

デフォルトのファイル出力は、ユーザーのホームディレクトリにあります。

```
java.util.logging.FileHandler.pattern = %h/EDB-JDBC%u.log
java.util.logging.FileHandler.limit = 50000000
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ingle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

</div>

次の表に、EDB Postgres Advanced Server および JDBC コネクタでサポートされる JDBC データ型を示します。setObject() メソッドを使用して 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、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タイプを返します。

JDBC Type	Advanced Server Type	getObject() returns
INTEGER	INT4	java.lang.Integer
TINYINT, SMALLINT	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, LONGVAR-CHAR	VARCHAR	java.lang.String
DATE	DATE	java.sql.Date
TIME	TIME, TIMETZ	java.sql.Timestamp
TIMESTAMP	TIMESTAMP, TIMESTAMPTZ	java.sql.Timestamp
BINARY	BYTEA	byte[](primitive)
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 ドライバがタイプを判別できるようにする場合にも使用されます。パラメータが *java.util.UUID* のインスタンスである場合、ドライバは適切な内部タイプを決定し、サーバーに送信します。