Javaアプリケーション開発ガイド
応用編

第2版 2011年9月

富士通株式会社

まえがき

■本書の目的

本書は、JavaアプリケーションからSymfoware Serverを利用する方法に関する補足的な内容について、説明しています。

以下の内容が理解できることを目標としています。

- ネーミングサービス、コネクションプール、ステートメントキャッシュなどのアプリケーションサーバの機能の利用方法
- BLOB、ストアドプロシジャ、排他制御などのSymfoware Serverの機能の利用方法
- Symfoware Serverのチューニングとデバッグの方法

■本書の読者

本書は、Symfoware Serverにアクセスするアプリケーションを開発される方に読んでいただくように書かれています。

本書を読むには、以下の知識が必要です。

- Symfoware Serverに関する知識
- SQLに関する知識
- Javaに関する一般知識
- JDBCに関する一般知識
- アプリケーションサーバに関する一般知識

※“Javaアプリケーション開発ガイド 入門編”および“Javaアプリケーション開発ガイド 一般編”の内容を理解できることを前提としています。

目次

第１章　ネーミングサービス 1
1.1　ネーミングサービスとは何か 2
1.2　Symfoware Serverでネーミングサービスを使用する 4
第２章　コネクションプール 8
2.1　コネクションプールとは何か 9
2.2　Symfoware Serverでコネクションプールを利用する 10
2.3　データベースへの接続の状態を確認する 12
第３章　ステートメントキャッシュ 13
3.1　ステートメントキャッシュとは何か 14
3.2　ステートメントキャッシュを使用する 15
3.3　ステートメントキャッシュの注意事項 16
第４章　トランザクション 17
4.1　トランザクションの制御方法 18
4.2　デッドロックの対処方法 20
第５章　BLOB 24
5.1　BLOBデータを扱う方法 25
第６章　ストアドプロシジャ 28
6.1　ストアドプロシジャとは何か 29
6.2　CallableStatement 30
第７章　チューニングパラメーター 33
7.1　動作環境ファイル 34
7.2　ctuneparam 36
第８章　デバッグ 37
8.1　エラー情報 38
8.2　スナップ情報の採取 39

第1章　ネーミングサービス

1.1　ネーミングサービスとは何か

実際の業務アプリケーションでは、DriverManager ではなく DataSource を使用する方法が一般的です。

DataSource を使用する方法では、データベースに接続するために必要な情報(データベース名やポート番号など)を、アプリケーションの外部で管理します。それらの接続情報はデータソースとして一元管理します。こうすることによって、接続先のデータベースの環境を変更した場合などでも、一元管理されている情報を変更するだけで、アプリケーションの修正をしなくて済みます。

データソースに名前をつけて管理する機能は、ネーミングサービスによって提供されます。ネーミングサービスは、Java アプリケーションやデータベースサーバとは別に動作しているプログラムです。Java アプリケーションがネーミングサービスを利用するための API が JNDI です。

Java アプリケーションはデータソース名を指定して JNDI の API を呼び出すことで、ネーミングサービスが管理している DataSource オブジェクトを得ることができます。その DataSource オブジェクトを用いてデータベースに接続します。

JNDI の API を用いて DataSource を得て、データベースに接続する手順は、以下のようになります。

(例)

```
InitialContext ctx = new InitialContext(env);

DataSource ds = (DataSource)ctx.lookup("jdbc/DataSource1");

Connection con = ds.getConnection();
```

使用するデータベースソフトが変わっても、JDBC でデータベースに接続する手順は同じです。上記の例では、InitialContext オブジェクトの lookup メソッドに指定した文字列の、「DataSource1」がデータソース名です。ネーミングサービスに関する情報は、InitialContext オブジェクトを作成するときの引数 env や、外部の設定ファイルで指定します。

1.2 Symfoware Serverでネーミングサービスを使用する

DataSource を使用してデータベースに接続するには、あらかじめデータソースの情報をネーミングサービスに登録しておく必要があります。

Symfoware Server では、データソースを登録する方法には、以下の 2 つがあります。

- Interstage を利用する方法
  Interstage で Java アプリケーションを実行する場合、ネーミングサービスには Interstage を利用します。データソースの登録には、Interstage 管理コンソールを用います。
  詳細な手順は、“Interstage 管理コンソール”のヘルプを参照してください。
- Symfoware Server 専用のツールを使用する方法
  Symfoware Server には、専用のネーミングサービスと、データソース登録ツールが付属しています。
  詳細な手順は、マニュアル“アプリケーション開発ガイド(JDBC ドライバ編)”を参照してください。

ネーミングサービスを使用するには、どのネーミングサービスを使用するのか、つまり Interstage と Symfoware Server 専用ツールのどちらをネーミングサービスとして利用するのか、を指定する必要があります。どのネーミングサービスを使用するのかは、InitialContext オブジェクトを作成するときに決まります。

ネーミングサービスの情報は、InitialContext オブジェクトを作成するときの引数や、外部の環境設定ファイルによって指定します。

■Interstage を利用する場合

InitialContext オブジェクトを作成する際の引数として、INITIAL_CONTEXT_FACTORY パラメーターを指定します。また、JNDI 環境設定ファイルでその他のパラメーターを設定できます。

Interstage のバージョンや利用する機能によって設定方法や設定内容が変わることがあります。詳細は、“Interstage 管理コンソール”のヘルプを参照してください。

なお、Interstage のバージョンによっては、Interstage の設定の延長で Symfoware Server 専用のネーミングサービスとデータソース登録ツールを使用することになる場合があります。

(例)

```
// ネーミングサービスの InitialContext を作成する。
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
"com.fujitsu.interstage.j2ee.jndi.InitialContextFactoryForClient");

// InitialContext を利用して DataSource を作成する。
DataSource ds = (DataSource)ctx.lookup("jdbc/DataSource1");

// DataSource を利用して Connection を作成する。
Connection con = ds.getConnection();
```

■Symfoware Server 専用のツールを使用する場合

Symfoware Server 専用のツールを用いて登録したデータソースを使用する場合は、以下のようにします。

(例)

```
// ネーミングサービスの InitialContext を作成します。
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
        "com.fujitsu.symfoware.jdbc2.jndisp.SYMContextFactory");
env.put(Context.PROVIDER_URL,"SYM://myhost:26600");
InitialContext ctx = new InitialContext(env);

// InitialContext を利用して DataSource を作成します。
DataSource ds = (DataSource)ctx.lookup("jdbc/DataSource1");

// DataSource を利用して Connection を作成します。
Connection con = ds.getConnection();
```

INITIAL_CONTEXT_FACTORY の値は、以下のとおりです。

com.fujitsu.symfoware.jdbc2.jndisp.SYMContextFactory

Symfoware Server 専用のネーミングサービスが使用するポート番号は、デフォルトで 26600（V9 までは 10326）です。これは変更することもできます。変更する方法は、マニュアル“アプリケーション開発ガイド(JDBC ドライバ編)”を参照してください。

Symfoware Server 専用のネーミングサービスを使用する場合は、ネーミングサービスの環境設定ファイルなどは使用しません。InitialContext オブジェクトを作成するときの引数として情報を指定してください。

［補足］

Symfoware Server が正式にサポートしているのは上記の方法ですが、一般のアプリケーションサーバ(Tomcat や JBoss など)を利用することも可能です。方法はそれぞれのアプリケーションサーバのマニュアルを参照してください。

第2章　コネクションプール

2.1 コネクションプールとは何か

データベースアプリケーションを運用する上で、コネクションプールを活用することが重要です。

アプリケーションがデータベースを利用するためには、まずデータベースに接続する必要があります。接続する際、通信環境を整えたりユーザー認証を行ったりする一連の処理が行われます。この処理は比較的コストが高いため、データベースへの接続と切断を何度も繰り返すと、アプリケーションの性能を低下させる要因となります。

そこで、一度接続したら切断せずに接続をキャッシュしておき、接続を使いまわすという方法が考案されました。これがコネクションプールです。

コネクションプールは、DataSource を使用してデータベースに接続する場合に利用できます。

2.2 Symfoware Serverでコネクションプールを利用する

Symfoware Server でコネクションプールを利用する方法は、以下の 2 種類があります。

- Interstage で設定する方法
- Symfoware Server で設定する方法

いずれの場合も、アプリケーション内でコネクションプールの設定を行ったり、特別なメソッドを呼び出したりする必要はありません。

■Interstage を利用する場合

Interstage を利用する場合、Interstage 管理コンソールからコネクションプールの設定を行うことができます。

Interstage 管理コンソールからは、Interstage のコネクションプールを利用するか、Symfoware Server のコネクションプールを利用するかを選択することができます。

Symfoware Server のコネクションプールを利用する場合は、コネクションプールの設定は Symfoware Server のデータソース登録ツールで行います。

Interstage V8 までは、Symfoware Server のコネクションプールを使用することのみが可能でした。Interstage V9 からは Interstage のコネクションプールを利用できるようになりました。Interstage を利用する場合には、Interstage のコネクションプールを使用した方が便利です。

Interstage での設定方法は、“Interstage 管理コンソール”のヘルプを参照してください。

■Symfoware Server 専用ツールを利用する場合

Symfoware Server のデータソース登録ツールで登録したデータソースを使用する場合、データソース登録時にコネクションプールの設定を行うことができます。データソース登録ツールの[JDBC データソース オプション設定]画面で[最大プールコネクション数]に、プールするコネクションの個数を設定します。

このデータソースを使用して接続すると、切断後もデータベースへの接続が保たれ、再度接続するときに再利用されます。

プールされた接続が再利用されるのは、同一のアプリケーション内です。また、最大プールコネクション数の設定はアプリケーションごとの値です。

例えば、最大プールコネクション数に 2 を設定したとします。3 個のアプリケーションが同じデータソースを使って 3 個ずつ接続したとすると、Symfoware Server への接続は全部で 9 個になります。

この状態で、すべての接続をクローズしたとすると、アプリケーションごとに 2 個ずつ接続がプールされて、6 個の接続が残ります。

アプリケーションを終了すると、接続は 0 個になります。

2.3 データベースへの接続の状態を確認する

Symfoware Server への接続の状態を調べるには、データベースサーバ上で rdbcninf コマンドを実行してください。

コネクションやトランザクションの実行状態、クライアントの IP アドレスやプロセス ID などを確認することができます。

第3章　ステートメントキャッシュ

3.1 ステートメントキャッシュとは何か

PreparedStatement を用いると、同じ SQL 文を何度も実行する場合には、SQL 文の解析を毎回実行するのではなく、先にやっておき、それを使いまわすことができます。

これは同じ SQL 文を使いまわすようなケースでは性能的に有利ですが、1 回使っただけでクローズする SQL 文の場合にはプリペアしても性能面でのメリットはありません。

そこで、クローズしたあともステートメントを保存しておき、同じ SQL 文であれば後から使いまわすことができる方法が考案されました。それがステートメントキャッシュ機能です。

ステートメントキャッシュ機能では、SQL 文をキーとしてキャッシュしておきます。同じ文をプリペアすると、キャッシュに保存しておいたものが使いまわされます。もしコネクションプールを使っていれば、セッションをまたいで SQL 文を使いまわすことができます。

ステートメントキャッシュは、PreparedStatement や CallableStatement を用いる場合に利用できます。

つまり、SQL 文を表すオブジェクトが Statement、Statement の解析結果を使いまわせるようにしたものが PreparedStatement、PreparedStatement をクローズ後にも使いまわせるようにしたものがステートメントキャッシュです。

3.2 ステートメントキャッシュを使用する

ステートメントキャッシュ機能は、以下の場合に使用できます。

- Interstage を使用していて、Interstage のコネクションプールを利用する場合。
  Interstage 管理コンソールからステートメントキャッシュの設定を行うことができます。
  詳細は Interstage のマニュアルを参照してください。
- Symfoware Server 専用のデータソース登録ツールを使用する場合。
  データソース登録ツールの[JDBC データソースオプション設定]画面の[ステートメントキャッシュ数]で、キャッシュする SQL 文の個数を指定することができます。デフォルトでは 32 個までキャッシュします。
  詳細はマニュアル“アプリケーション開発ガイド(JDBC ドライバ編)”を参照してください。

3.3 ステートメントキャッシュの注意事項

Symfoware Server でステートメントキャッシュを使用する場合は、以下の点に注意してください。

- ステートメントキャッシュ機能を使うと、キャッシュする SQL 文に応じてメモリを消費します。メモリ使用量の目安は、1 文あたり 100 キロバイトです。
- MAX_SQL または CLI_MAX_SQL を超える数のステートメントキャッシュをすることはできません。MAX_SQL または CLI_MAX_SQL を超えた場合には、アプリケーション実行時にエラーとなる場合があります。
  MAX_SQL または CLI_MAX_SQL には、ステートメントキャッシュ数よりも大きい値を設定してください。
- SQL 文がキャッシュされるのは、PreparedStatement オブジェクトや CallableStatement オブジェクトをクローズしたとき、またはガベージコレクションによって破棄されたときです。
  明にオブジェクトをクローズせず、ガベージコレクションに任せているとキャッシュされるタイミングが遅くなります。
  オブジェクトは明にクローズするようにしてください。
- Symfoware Server のデータソース登録ツールで[ステートメント自動クローズ]を設定している場合、一定の条件を満たせば、明にクローズしなくても、自動的にステートメントがクローズされます。
  ただし、同じ内容の SQL 文を複数使用している場合、ステートメント自動クローズを使用すると、アプリケーションが意図しないタイミングで SQL 文がクローズされてしまい、アプリケーションがエラーになることがあります。

詳細は、マニュアル“アプリケーション開発ガイド(JDBC ドライバ編)”の“ステートメントキャッシュ”を参照してください。

第4章　トランザクション

4.1 トランザクションの制御方法

Symfoware Server では、SELECT 文や INSERT 文などの DML 文を実行することによって、自動的にトランザクションが開始されます。その後、COMMIT 文または ROLLBACK 文によってトランザクションを終了させるか、エラーによってトランザクションがロールバックされるまで、トランザクションが継続します。

Java アプリケーションで JDBC を用いる場合、デフォルトの状態では 1 個の SQL 文を実行するごとに、自動的にトランザクションがコミットされます。複数の SQL 文から構成されるトランザクションを実行するには、まず自動コミットを無効にする必要があります。自動コミットを無効にするには、Connection オブジェクトの setAutoCommit メソッドを用います。

また、トランザクション単位に排他制御の方法を指定する機能として独立性水準があります。独立性水準を指定するには、Connection オブジェクトの setTransactionIsolation メソッドを用います。

トランザクションをコミット、またはロールバックさせて終了させるには、Connection オブジェクトの commit メソッドと rollback メソッドを用います。

(例)

```
// データソースを用いてデータベースに接続します。
InitialContext ctx = new InitialContext(env);
DataSource ds = (DataSource)ctx.lookup("jdbc/DataSource1");
Connection con = ds.getConnection();

// 自動コミットを無効にします。
con.setAutoCommit(false);
```

```
        // 独立性水準を READ UNCOMMITTED に設定します。
        con.setTransactionIsolation(java.sql.Connection.TRANSACTION_READ_UNCOMMIT
TED);


        // INSERT 文などの DML 文を実行すると、トランザクションが自動的に開始されます。
        Statement stmt = con.createStatement();
        stmt.executeUpdate(
                "INSERT INTO GENERAL.EMPLOYEE(ID,NAME) VALUES(1,'tiger')");


        // トランザクションをコミットします。
        con.commit();
```

4.2 デッドロックの対処方法

■デッドロックとは

複数のトランザクションが同時に動作すると、それぞれのトランザクションによるデータの排他制御が競合して、互いに相手のロック解除を待つ状況になってしまうことがあります。これをデッドロックと呼びます。

Symfoware Server では、デッドロックが発生すると、自動的にどちらかのトランザクションがロールバックされます。どちらのトランザクションがロールバックされるのかは、事前には分かりません。

また、行単位の排他制御を行っている場合、コミット処理でもデッドロックが発生する可能性があります。

■デッドロックの発生頻度を下げる方法

デッドロックが発生する可能性を 0 にすることはできません。以下のような手法によって、デッドロックになる可能性を低減させることができます。

- データにアクセスする順番を統一します。
  排他制御の競合が起きても、順にロック解除されるのを待つだけで、互いに待ち状態になることを防ぐことができます。
- 排他制御のロック強度を弱くします。
  例えば独立性水準を READ UNCOMMITTED に設定することで、データを参照する際にロックを行わなくなります。ロックしないので、互いにロック解除待ちになる可能性を減らすことができます。
  ただし、ダーティリードが発生することがあります。また、更新処理では必ずデータがロックされます。
- 排他制御のロック強度を強くします。
  例えば、検索処理時に FOR UPDATE 句をつけることで、非共有モードでデータをロック

します。通常の検索では共有モードでロックするため、他のトランザクションから多重にロックされる可能性があり、いざ自トランザクションで更新しようとしたときにロック解除待ちになってしまうことがあります。最初から更新を前提とした非共有モードのロックを行うことで、ロック解除待ちになる可能性を減らすことができます。ただし、複数のトランザクションの多重実行性が低下するため、性能が劣化する可能性があります。

■デッドロック発生時の対処方法

データベースにアクセスするアプリケーションを開発する場合は、デッドロックが発生することを前提としてください。デッドロックを回避するさまざまな手法を駆使したとしても、デッドロックが発生する可能性は残ります。

デッドロックが発生すると、トランザクションがロールバックされます。デッドロックが発生してもデータが破壊される等の危険はありません。そのため、デッドロックが発生いしたら、単にそのトランザクションを再実行すればよいのです。

トランザクションを再実行するには、以下の方法があります。

- ユーザーにエラーを通知し、再実行を促す。
- デッドロックが起きたことをアプリケーション内で検知し、処理を再実行する。

JDBC の API を使用して何らかのエラーが発生すると、例外として SQLException が発生します。SQLException 内に SQLSTATE というエラー情報が格納されています。SQLSTATE を参照することでエラーの原因が分かるので、デッドロックが起きた場合にトランザクションを再実行することができます。

以下に SQLSTATE からデッドロックが起きたことを検知して、トランザクションを再実行する例を示します。

(例)

```
// トランザクションの再実行回数を保持するカウンタを定義する。
int retry = 0;
```

```
// デッドロックによってトランザクションがロールバックされても、再度実行できるように
// WHILE 文の内部でトランザクションを実行する。
while(retry<10)
{
    try
    {
        // Statement を用意する。
        Statement stmt = con.createStatement();

        // 2 個の表をそれぞれ更新する。もし 2 個の表を逆の順番で更新する
        // アプリケーションが同時に動作していると、デッドロックが起きる
        // 可能性がある。
        stmt.executeUpdate("UPDATE GENERAL.EMPLOYEE SET
NAME= 'tiger' WHERE ID=1");
        stmt.executeUpdate("UPDATE GENERAL.EMPLOYEE2 SET
NAME2= 'tiger' WHERE ID2=4");

        // 更新をコミットする。
        con.commit();
        stmt.close();

        // トランザクションが正常終了したら、ループを抜けるために
        // ループの実行条件を偽にする値(この例では 10)を設定する。
```

```
                retry=10;
            }
            catch (SQLException e)
            {
                // JDBC でエラーが発生した場合、デッドロックかどうかを確認する。
                // デッドロックの場合、SQLSTATE=40001 が設定されている。
                if(e.getSQLState().equals("40001")){
                    System.out.println("Deadlock. Transaction rollback");
                    retry++;
                }
                else{
                // デッドロック以外のエラーの場合、トランザクションの再実行はせず、
                // 例外を再スローする。
                    throw e;
                }
                // トランザクションを何度か再実行してもデッドロックが解消されない
                // 場合はあきらめてエラー終了する。この例では 2 回まで再実行。
                if(retry>2){
                    throw e;
                }
            }
        }
```

第5章　BLOB

5.1 BLOBデータを扱う方法

画像データなどの巨大なバイナリデータをデータベースに格納するために、Symfoware Server には BLOB 型の列を定義することができます。

Java のアプリケーションから BLOB 型の列のデータを扱うには、ストリームオブジェクトを使用します。

ResultSet オブジェクトの getter メソッドや、PreparedStatement オブジェクトの setter メソッドで、バイナリストリームを指定することで、BLOB 型の列のデータを扱うことができます。

データベースの BLOB 型の列に格納されているデータを SELECT 文で取り出すには以下のようにします。

(例)

```
// Statement オブジェクトを作って、BLOB 列を含む表の検索を行う。
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery(
        "SELECT COL_BLOB FROM BLOBTBL WHERE COL_INT = 1");


while (rs.next())
{
        // ResultSet から BLOB 列の値を、バイナリストリームとして取り出す。
        InputStream oIS = rs.getBinaryStream(1);
```

```
                // 取り出した BLOB 列の値の、出力ファイルを準備する。
                FileOutputStream fs = new FileOutputStream("blobdat.dat");


                // BLOB 列の値をバイト列として取り出し、出力ファイルに書きこむ。
                byte[] bTmp = new byte[oIS.available()];
                oIS.read(bTmp);
                fs.write(bTmp);
                int iBData=0;
                while ((iBData=oIS.read()) != -1)
                {
                        fs.write(iBData);
                }

                // ストリームオブジェクトをクローズする。
                oIS.close();
                fs.close();
        }
```

データベースの BLOB 型の列に INSERT 文でデータを格納するには、以下のようにします。

(例)

```
        // ストリームオブジェクトを指定するためのプレースホルダを持つ SQL 文を用意する。
        PreparedStatement pstmt = con.prepareStatement(
```

```
                "INSERT INTO BLOBTBL VALUES(1, ?)");

    // BLOB 列に格納するデータのファイルを準備する。
    FileInputStream oFIS = new FileInputStream("blobdat.dat");

    // SQL 文のプレースホルダに入力データのストリームをセットする。
    pstmt.setBinaryStream(1, oFIS, oFIS.available());

    // INSERT 文を実行する。
    pstmt.executeUpdate();

    //  ストリームオブジェクトをクローズする。
    oFIS.close();
```

［補足］

BLOB 型の列を扱うために、BLOB オブジェクト(java.sql.Blob 型)と getBlob メソッド/setBlob メソッドを用いることもできます。

第6章　ストアドプロシジャ

6.1 ストアドプロシジャとは何か

ストアドプロシジャとは、複数の SQL 文からなる一連の処理を、Symfoware Server に登録しておく機能です。登録しておいたストアドプロシジャは、アプリケーションから呼び出して実行することができます。アプリケーションは、ストアドプロシジャを呼び出す処理だけを行えば、一連の SQL 処理がまとめて実行されます。個々の SQL 文をアプリケーションから順に実行する場合に比べて、クライアントとサーバの通信回数を減らすことができ、性能向上に役立ちます。

Symfoware Server では、ストアドプロシジャは SQL 文と各種の制御文によって記述します。ストアドプロシジャ自体を Java で記述することはできません。

ストアドプロシジャについては、マニュアル“SQL リファレンス”の“ストアドプロシジャ”を参照してください。

6.2 CallableStatement

CallableStatement クラスは、PreparedStatement クラスを拡張したものです。

CallableStatement クラスは、ストアドプロシジャを呼び出すために使用します。

ストアドプロシジャには入力引数を与えたり、結果を引数として受け取ったりすることができます。これらのパラメーターは、CallableStatement オブジェクトのメソッドで指定することができます。

ストアドプロシジャを呼び出すには、以下のようにします。

(1) 実行したい CALL 文を内容とする String オブジェクトを作成します。
このとき、引数は「?」としておきます。
CALL 文全体を{}で括る必要がある点に注意してください。これは JDBC の仕様です。
(2) Connection オブジェクトの prepareCall メソッドを用いて、CallableStatement オブジェクトを作成します。
(3) CallableStatement オブジェクトの setter メソッドを用いて、プレースホルダに値を設定します。
(4) CallableStatement オブジェクトの registerOutParameter メソッドを用いて、出力パラメーターの型を設定します。
(5) CallableStatement オブジェクトの executeUpdate メソッドを用いてストアドプロシジャを呼び出します。
(6) CallableStatement オブジェクトの getter メソッドを用いて、ストアドプロシジャの出力引数の値をプレースホルダから取り出します。

以下のような 3 個の引数を持つプロシジャを呼び出すとします。

```
CREATE PROCEDURE GENERAL.PROC01 (IN X integer, INOUT Y char(10), OUT Z integer)
```

1 個目の引数は入力専用、2 個目の引数は入出力兼用、3 個目の引数は出力専用です。

入力に使用する 1 個目と 2 個目の引数には、setter メソッドを用いてプレースホルダに値を設定しておきます。

出力に使用する 2 個目と 3 個目の引数については、プロシジャで定義されている引数の型を指定します。ここで指定する型はプロシジャで定義した型なので、java.sql.Types の型になります。

プロシジャが出力用引数で返却した値は、getter メソッドを用いて取り出すことができます。

なお、CallableStatement オブジェクトを用いて CALL 文を実行する際、CALL 文全体を「{}」で括る必要がある点に注意してください。「{}」で括るのを忘れると、CALL 文を実行したときにエラーになります。

(例)

```
// プロシジャを呼び出す CALL 文を準備する。
// CALL 文を{}で括るのを忘れずに。
String sql = "{CALL GENERAL.PROC01[?,?,?]}";

CallableStatement cstmt = con.prepareCall(sql);

// CALL 文のプレースホルダに、プロシジャの入力引数をセットする。
cstmt.setInt(1,1);
cstmt.setString(2,"XYZ");
```

```
// プロシジャの出力引数の型を指定する。
cstmt.registerOutParameter(2,java.sql.Types.CHAR);
cstmt.registerOutParameter(3,java.sql.Types.INTEGER);

// プロシジャを呼び出す。
cstmt.executeUpdate();

// プロシジャの出力引数の値を取り出す。
String outY = cstmt.getString(2);
int outZ = cstmt.getInt(3);

System.out.println("Y " + outY);
System.out.println("Z " + outZ);

// トランザクションをコミットする。
cstmt.close();
con.commit();
```

第7章　チューニングパラメーター

7.1 動作環境ファイル

Symfoware Server では、アプリケーションの動作をチューニングするために、動作環境ファイルというパラメーター設定ファイルを使用します。

動作環境ファイルには、以下の 3 種類があります。

- システム用の動作環境ファイル
  Symfoware Server Standard Edition、Symfoware Server Enterprise Edition、または、Symfoware Server Enterprise Extended Edition の場合のみ、RDB システムごとに 1 個存在します。(Symfoware Server Lite Edition では、システム用の動作環境ファイルは存在しません。)
- クライアント用の動作環境ファイル
  必要に応じてアプリケーションごとに作成します。
- サーバ用の動作環境ファイル
  必要に応じてデータベースへの接続ごとに作成します。

動作環境ファイルには、アプリケーションの動作をチューニングするためのさまざまなパラメーターを設定することができます。設定できるパラメーターの種類は、動作環境ファイルの種類によって決まっています。パラメーターによっては、異なる種類の動作環境ファイルに重複して設定できるものもあります。この場合、パラメーターの指定の優先順位は、以下のようになります。

(1) サーバ用の動作環境ファイル
(2) クライアント用の動作環境ファイル
(3) システム用の動作環境ファイル

例えば、行単位の排他を行うかどうかを指定するパラメーターR_LOCK は、システム用の動作環境ファイルとクライアント用の動作環境ファイルに設定することがあります。

システム用の動作環境ファイルで「R_LOCK=NO」を指定し、クライアント用の動作環境ファイルで「R_LOCK=YES」を指定した場合、クライアント用の動作環境ファイルの設定が優先され、行単位の排他が行われます。ただし、クライアント用の動作環境ファイルは、アプリケーション単位なので、その動作環境ファイルを使用しないアプリケーションでは、システム用の動作環境ファイルの設定が使用されます。

動作環境ファイルで設定できるパラメーターについては、以下のマニュアルを参照してください。

- Symfoware Server V10 以降
  “アプリケーション開発ガイド(共通編)”の“動作環境ファイルのパラメタ一覧”
- Symfoware Server V9 以前
  “アプリケーション開発ガイド(埋込み SQL 編)”の“動作環境ファイルのパラメタ一覧”

Java アプリケーションの場合、Symfoware Server V9 までのバージョンでは、クライアント用の動作環境ファイルは使用しません。Symfoware Server V10 以降は Java アプリケーションでもクライアント用の動作環境ファイルを用いたチューニングが可能になっています。

7.2 ctuneparam

Symfoware Server V9 まではクライアント用の動作環境ファイルによるチューニングは、Java アプリケーションでは行うことができませんでした。

代わりに、クライアント用の動作環境ファイルと同等のパラメーター設定を、ctuneparam というパラメーターによって行うことができます。

ctuneparam によるチューニングは、Symfoware Server V10 以降でも使用できます。ただし、クライアント用の動作環境ファイルを用いたほうが変更しやすいため、V10 以降ではクライアント用の動作環境ファイルの使用を推奨します。

ctuneparam を含めた、動作環境ファイルのパラメーターの指定の優先順位は以下のようになります。

(1) サーバ用の動作環境ファイル
(2) ctuneparam
(3) クライアント用の動作環境ファイル
(4) システム用の動作環境ファイル

ctuneparam は、以下の場所で指定することができます。

- DataSource を用いてデータベースに接続する場合
  データソース登録ツールの[JDBC データソースオプション設定]画面の[その他パラメタ]欄。
  詳細は、マニュアル“アプリケーション開発ガイド(JDBC ドライバ編)”の“データソースのオプション情報設定画面”を参照してください。
- DriverManager を用いてデータベースに接続する場合
  getConnection で指定する URL のオプション部分。
  詳細は、マニュアル“アプリケーション開発ガイド(JDBC ドライバ編)”の“URL 記述形式”を参照してください。

第8章　デバッグ

8.1 エラー情報

Java アプリケーションでエラーが発生すると、例外オブジェクトが生成されます。JDBC の処理でエラーが発生した場合、例外オブジェクトとして SQLException が生成されます。

SQLException には、エラーメッセージと SQLSTATE が保持されています。これらの情報を取り出すことで、エラーの原因を調べることができます。

8.2 スナップ情報の採取

使用した JDBC の API ごとに、処理の経過時間と入出力データの情報を採取することができます。この情報を JDBC スナップと呼びます。

JDBC スナップの内容を調べることによって、処理の性能を分析したり、異常動作の原因を探ったりすることができます。

JDBC スナップを採取するには、以下のようにします。

(1) JDBC スナップの実行パラメーターを記述したテキストファイルを作成します。
ファイルの内容は以下のように記述します。

JDBC_SNAP=(出力モード[,スナップファイル名][,出力レベル])

ファイル名は任意です。この例では、/tmp/symjdbc.env であるとします。

(例)

```
JDBC_SNAP=(ON,/tmp/symfo.snp,1)
```

出力レベル 1 で、JDBC スナップを採取し、結果を/tmp/symfo.snp に出力します。

JDBC スナップの詳細については、マニュアル“アプリケーション開発ガイド(JDBC ドライバ編)”の“JDBC スナップによる対処”を参照してください。

(2) アプリケーション実行時に、java コマンドの引数としてファイルを指定します。
環境変数 SYMJDBCENV に、実行パラメーターを指定したファイル名を絶対パスで指定します。

(例)

```
java -DSYMJDBCENV=/tmp/symjdbc.env
```