JDBC. Leksykon kieszonkowy

Autor: Donald Bales
Tłumaczenie: Jarosław Gierlicki
ISBN: 83-7361-165-7
Tytuł oryginału: JDBC Pocket Reference
Format: B5, stron: 196

Podręczny przewodnik dla programistów Javy

JDBC jest jednym z najwszechstronniejszych i najszerzej stosowanych mechanizmów umożliwiających nie tylko dostęp do danych z programów napisanych w Javie, ale również wykonywanie rozmaitych operacji na bazach danych. Kosztem wszechstronności jest złożoność – JDBC implementuje ogromną liczbę klas, interfejsów i metod, z którymi trzeba się gruntownie zapoznać. Niemożliwością jest zapamiętanie ich wszystkich.

„JDBC. Leksykon kieszonkowy" jest podręcznym leksykonem ułatwiającym szybkie odnalezienie opisu każdej z klas, interfejsów czy metod tworzących JDBC API. Przydaje się zwłaszcza w sytuacjach, w których wiadomo, co się chce osiągnąć i potrzebna jest informacja na temat metod JDBC, które mogą w tym pomóc.

Oprócz opisu API książka zawiera także kilka krótkich rozdziałów będących swego rodzaju instrukcją obsługi JDBC, czyli przypomnieniem, w jaki sposób łączyć ze sobą różne elementy API w celu uzyskania rozwiązań konkretnych zadań.

Nie pamiętasz, jak naraz wykonać kilka operacji wstawiania lub modyfikacji lub w jaki sposób przechowywać i pobierać wielkie obiekty? Niniejsza książka na pewno odświeży Twoją pamięć.

Donald Bales jest konsultantem do spraw aplikacji komputerowych specjalizującym się w analizie, projektowaniu oraz programowaniu systemów rozproszonych. Posiada ponad pięcioletnie doświadczenie w pisaniu aplikacji w Javie operujących na bazach danych za pomocą JDBC.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Dynamiczne wykonywanie poleceń SQL

Za pomocą obiektu `Statement` można dynamicznie wykonać konstrukcje DDL SQL lub DML SQL. Obiekt `Statement` otrzymujemy na podstawie obiektu połączenia, wywołując jego metodę `createStatement()` w następujący sposób:

```
Statement stmt = null;
try {
  stmt = conn.createStatement();
  ...
}
catch (SQLException e) {
  ...
}
```

Obiekt `Statement` posiada trzy metody służące do wykonywania SQL-a: `execute()`, `executeUpdate()` oraz `executeQuery()`. Wybór odpowiedniej metody zależy od typu konstrukcji SQL, która ma zostać wykonana.

Wykonywanie konstrukcji DDL

Metoda `execute()` jest najodpowiedniejsza dla poleceń DDL czy całkowicie dynamicznych poleceń SQL. Zwraca ona wartość `true`, jeżeli wykonanie konstrukcji SQL spowodowało wygenerowanie zbioru wynikowego. Zbiór wynikowy można otrzymać wywołując dla obiektu konstrukcji metodę `getResultSet()`. Jeżeli metoda `execute()` zwróci wartość `false`, można za pomocą metody `getUpdateCount()` obiektu `Statement` określić liczbę wierszy objętych działaniem konstrukcji DML. Jeśli natomiast w trakcie wykonywania polecenia DDL wystąpi błąd, zostanie zgłoszony wyjątek `SQLException`. Oto przykład, w którym zostaje utworzona tabela o nazwie `PERSON` (przy założeniu, że połączenie — `conn` — istnieje):

```
Statement stmt = null;
try {
  stmt = conn.createStatement();
  stmt.execute(
    "create table PERSON ( " +
    "person_id  integer not null primary key, " +
    "last_name  varchar(30), " +
    "first_name varchar(30), " +
    "birth_date date, " +
    "gender     varchar(1) ) ");
  stmt.close();
  stmt = null;
}
catch (SQLException e) {
  ...
}
finally {
  if (stmt != null)
    try { stmt.close(); } catch (Exception i) { }
}
```

Wykonywanie konstrukcji INSERT, UPDATE oraz DELETE

Metoda `executeUpdate()` nadaje się do wykonywania konstrukcji DML innych niż konstrukcja SELECT i zwraca liczbę wierszy objętych działaniem danej konstrukcji SQL. Poniżej przedstawiono przykład, w którym do tabeli `PERSON` utworzonej w poprzednim przykładzie zostaje wstawiony nowy wiersz (przy założeniu, że połączenie — `conn` — istnieje):

```
int       rows = 0;
Statement stmt = null;
try {
  stmt = conn.createStatement();
  rows = stmt.executeUpdate(
    "insert into PERSON ( " +
    "       person_id, " +
```

```
    "       last_name, " +
    "       first_name, " +
    "       birth_date, " +
    "       gender ) " +
    "values ( " +
    "       1, " +
    "       'DOE', " +
    "       'JOHN', " +
    "       { d '1980-01-01' }, " +
    "       'M' ) ");
  stmt.close();
  stmt = null;
  if (rows != 1) {
    System.err.println(
      "Błąd w konstrukcji: " +
      "wstawiono " + rows + ".");
  }
}
catch (SQLException e) {
  ...
}
finally {
  if (stmt != null)
    try { stmt.close(); } catch (Exception i) { }
}
```

Wykonywanie konstrukcji SELECT

Metoda `executeQuery()` **zwraca obiekt** `ResultSet`**, co świetnie nadaje się do wykonywania konstrukcji SQL SELECT. Poniżej przedstawiono przykład zapytania wykonanego dla tabeli** `PERSON` **(przy założeniu, że połączenie —** `conn` **— istnieje):**

```
ResultSet rset = null;
Statement stmt = null;
try {
  rows = 0;
  stmt = conn.createStatement();
  rset = stmt.executeQuery(
```

```
      "select person_id, " +
      "       last_name, " +
      "       first_name, " +
      "       birth_date, " +
      "       gender " +
      "from   PERSON " +
      "where  person_id = 1 ");
    while (rset.next()) {
      rows++;
      idOsoby    = rset.getInt(1);
      last_name  = rset.getString(2);
      first_name = rset.getString(3);
      dataUr     = rset.getDate(4);
      gender     = rset.getString(5);
      System.out.println(
        idOsoby    + "\t" +
        last_name  + "\t" +
        first_name + "\t" +
        dataUr     + "\t" +
        gender );
    }
    rset.close();
    rset = null;
    stmt.close();
    stmt = null;
  }
  catch (SQLException e) {
    ...
  }
  finally {
    if (rset != null)
      try { rset.close(); } catch (Exception i) { }
    if (stmt != null)
      try { stmt.close(); } catch (Exception i) { }
  }
```

Warto zwrócić uwagę, że w trzech ostatnich przykładach obiekt `Statement` **był zawsze zamykany. Zamykanie obiektu** `Statement`

zaraz po tym, gdy przestaje on być potrzebny, zmniejsza zużycie zasobów zarówno w programie klienta, jak i w bazie danych.

Wykonywanie prekompilowanych poleceń SQL

Wykonywanie konstrukcji SQL przy użyciu obiektów `PreparedStatement` może być bardziej efektywne (z punktu widzenia programu klienta oraz bazy danych) oraz łatwiejsze w programowaniu. Obiekt `PreparedStatement` można otrzymać z obiektu `Connection`, wywołując w następujący sposób jego metodę `prepareStatement()`:

```
PreparedStatement pstmt = null;
try {
  pstmt = conn.prepareStatement(
    "insert into PERSON ( " +
    "       person_id, " +
    "       last_name, " +
    "       first_name, " +
    "       birth_date, " +
    "       gender ) " +
    "values ( " +
    "       ?, " +
    "       ?, " +
    "       ?, " +
    "       ?, " +
    "       ? ) ");
}
catch (SQLException e) {
  ...
}
```

Gotowa konstrukcja jest bardziej efektywna, jeżeli można przygotować ją raz, a potem wykorzystywać wiele razy. A łatwiejsza w programowaniu jest dlatego, że w trakcie tworzenia konstrukcji SQL uwalnia od obowiązku tworzenia złożonych konkatenacji

(sklejeń) łańcuchów znaków oraz od konieczności używania odpowiednich metod formatujących daty. Zamiast tego można jako oznaczenia miejsc występowania parametrów użyć znaków zapytania (?), a następnie przed wykonaniem konstrukcji SQL wartości tych parametrów ustalić programowo.

Ponieważ klasa `PreparedStatement` jest rozszerzeniem klasy `Statement`, posiada jej wszystkie trzy metody uruchomieniowe. Oprócz tego posiada także zestaw metod typu `setXXX()`, które służą do umieszczania w konstrukcji przed jej wykonaniem konkretnych wartości w miejscach oznaczonych znakami zapytania — sposób ich stosowania przedstawiono w poniższym przykładzie. Założono, że istnieją utworzone w poprzednim przykładzie obiekty połączenia — `conn` oraz obiekty gotowej konstrukcji — `pstmt`.

```
try {
  pstmt.setInt(1, 2);
  pstmt.setString(2, "DOE");
  pstmt.setString(3, "JOHN");
  pstmt.setDate(4, java.sql.Date.valueOf("1980-01-01"));
  pstmt.setString(5, "F");
  rows = pstmt.executeUpdate();
  pstmt.close();
  pstmt = null;
  if (rows != 1) {
    System.err.println(
      "Błąd w gotowej konstrukcji: " +
      "wstawiono " + rows + ".");
  }
}
catch (SQLException e) {
  ...
}
```

Do znaczników parametrów (?) występujących w konstrukcji SQL metody odwołują się począwszy od numeru 1, a potem kolejno od lewej do prawej. Tak więc w powyższym przykładzie polu `person_id` zajmującemu pierwszą pozycję zostaje przypisana wartość

liczbowa 2 (typu `int`), polu `last_name`, zajmującemu pozycję drugą, zostaje przypisana wartość łańcucha znaków `"DOE"` itd. Po ustaleniu wartości wszystkich parametrów konstrukcja zostaje uruchomiona poprzez wywołanie metody `executeUpdate()` obiektu `PreparedStatement`. Ponieważ treść konstrukcji SQL została przygotowana wcześniej, nie jest ona przekazywana jako parametr do żadnej z metod uruchomieniowych obiektu `PreparedStatement`.

Wykonywanie procedur osadzonych

Jeżeli pobranie lub zachowanie pewnej jednostkowej ilości danych w programie wymaga wykonania wielu konstrukcji SQL, wykorzystanie procedury osadzonej jest bardziej efektywne niż kolejne uruchamianie przez klienta wszystkich koniecznych konstrukcji. Do uruchomienia procedury osadzonej można użyć obiektu `CallableStatement`. Uzyskuje się go z obiektu `Connection`, wywołując w następujący sposób metodę `prepareCall()`:

```
CallableStatement cstmt = null;
try {
  cstmt = conn.prepareCall(
    "{ ? = call add( ?, ? ) }");
  ...
}
catch (SQLException e) {
  ...
}
```

Szczegóły dotyczące składni wywoływania procedur osadzonych opisano w rozdziale „Składnia wyrażeń przenośnych". Klasa `CallableStatement` jest rozszerzeniem klasy `PraparedStatement`, stąd w identyczny sposób ustala się dla niej miejsca występowania parametrów — za pomocą znaków zapytania (?). Jednak tym razem parametry te mogą być typu wejściowego, wyjściowego

lub wejściowo-wyjściowego. Po przygotowaniu wywołania procedury osadzonej, a przed jej uruchomieniem, należy wykonać jeszcze następujące czynności:

- Wszystkie parametry wyjściowe muszą zostać zarejestrowane za pomocą metody `registerOutParameters()`, która pobiera dwa argumenty. Pierwszy z nich jest pozycją odpowiedniego znacznika parametru (?) w przygotowanym wywołaniu (1 dla pierwszego znacznika, a następnie kolejno od lewej do prawej). Drugi parametr jest jedną ze stałych zdefiniowanych w `java.sql.Types` oznaczającą typ, co umożliwia sterownikowi rozpoznanie, jaki rodzaj danych jest zwracany przez procedurę osadzoną.
- Wartość każdego parametru wejściowego i wejściowo-wyjściowego musi być ustalona za pomocą jednej z funkcji `setXXX()` odpowiedniej dla typu danych występujących w bazie. Na przykład:

```
try {
  cstmt = conn.prepareCall(
   "{ ? = call add( ?, ? ) }");
  cstmt.registerOutParameter(1, Types.NUMERIC);
  cstmt.setDouble(2, 1.0);
  cstmt.setDouble(3, 1.0);
  ...
}
catch (SQLException e) {
  ...
}
```

Do uruchomienia procedury osadzonej należy wykorzystać metodę `execute()`. Aby pobrać wartość któregoś z parametrów wyjściowych procedury osadzonej, należy w przedstawiony poniżej sposób użyć jednej z metod `getXXX()` obiektu `CallableStatement` — odpowiedniej dla danego typu danych SQL:

```
double answer = 0.0;
try {
  cstmt = conn.prepareCall("{ ? = call add( ?, ? ) }");
  cstmt.registerOutParameter(1, Types.NUMERIC);
  cstmt.setDouble(2, 1.0);
  cstmt.setDouble(3, 1.0);
  cstmt.execute();
  answer = cstmt.getDouble(1);
  cstmt.close();
}
catch (SQLException e) {
  ...
}
finally {
  if (cstmt != null)
    try { cstmt.close(); }
    catch (SQLException i) { }
}
```

Otrzymywanie wyników zapytań

Wyniki wykonania konstrukcji SELECT są zwracane przez metodę `executeQuery()` w postaci zbioru wynikowego (`ResultSet`). Obiekt `ResultSet` prezentuje zbiór wynikowy w postaci tabeli zawierającej wiersze i kolumny. Może on być przeglądany i modyfikowany, może także być wrażliwy na zmiany w bazie danych (jeżeli sterownik i baza danych oferują taką możliwość dla zapytań). Domyślnie zbiór wynikowy można przeglądać jedynie w przód, po jednym rekordzie, wartości w kolumnach nie można modyfikować, a sterownik nie jest w stanie wykryć zmian zachodzących w bazie danych. Aby stworzyć zbiór wynikowy, który można przeglądać, modyfikować i który będzie wykrywać zmiany w bazie, należy użyć alternatywnej wersji jednej z metod `CreateStatement()`, `prepareStatement()` lub `prepareCall()` obiektu `Connection`. Metody te pobierają dwa dodatkowe argumenty:

concurrency constant
(stała współbieżności obiektu `ResultSet`)
: Określa, czy zbiór wynikowy może być tylko przeglądany, czy możliwe będzie także dokonywanie za jego pomocą modyfikacji. Dostępne są dwie wartości stałej tego typu:

```
CONCUR_READ_ONLY
CONCUR_UPDATABLE
```

type constant
(stała typu obiektu `ResultSet`)
: Określa, w jaki sposób można przeglądać zbiór wynikowy powstały w efekcie wykonania zapytania oraz ustala, czy sterownik ma wykrywać zmiany w bazie danych dotyczące rekordów objętych zbiorem wynikowym. Oto dostępne wartości tej stałej:

`TYPE_FORWARD_ONLY`
: Dla zbioru wynikowego przeglądanego tylko w przód, niewrażliwego na modyfikacje.

`TYPE_SCROLL_INSENSITIVE`
: Dla zbioru wynikowego przeglądanego dwukierunkowo, niewrażliwego na modyfikacje.

`TYPE_SCROLL_SENSITIVE`
: Dla zbioru wynikowego przeglądanego dwukierunkowo, wrażliwego na modyfikacje.

Poniżej przedstawiono przykład utworzenia zbioru wynikowego, który może być przeglądany, za pomocą którego można modyfikować rekordy w bazie i który jest w stanie wykryć zmiany dokonywane w bazie (przy założeniu, że połączenie — `conn` — istnieje):

```
ResultSet rset = null;
Statement stmt = null;
try {
  rows = 0;
  stmt = conn.createStatement(
```

```
      ResultSet.CONCUR_UPDATABLE,
      ResultSet.TYPE_SCROLL_SENSITIVE);
    rset = stmt.executeQuery(
      "select person_id, " +
      "       last_name, " +
      "       first_name, " +
      "       birth_date, " +
      "       gender " +
      "from   PERSON " +
      "where  person_id = 1 ");
    ...
  }
  catch (SQLException e) {
    ...
  }
```

Przeglądanie zbioru wynikowego

Jeżeli zbiór wynikowy został ustalony jako możliwy do przeglądania tylko w przód, do przesuwania się co jeden wiersz należy używać metody `next()`. Zazwyczaj ta operacja wykonywana jest w pętli o na przykład takiej strukturze (przy założeniu, że połączenie — `conn` — istnieje):

```
  ResultSet rset = null;
  Statement stmt = null;
  try {
    rows = 0;
    stmt = conn.createStatement();
    rset = stmt.executeQuery("select ...");
    while (rset.next()) {
      ...
    }
    ...
  }
  catch (SQLException e) {
    ...
  }
```

Jeżeli zbiór może być przeglądany dwukierunkowo, do ustawiania pozycji kursora można używać jednej z następujących metod: `absolute()`, `first()`, `last()`, `next()`, `previous()` oraz `relative()`. Niezależnie od tego, czy zbiór można przeglądać jedno- czy dwukierunkowo, bieżącą pozycję kursora można określić za pomocą jednej z metod: `isAfterLast()`, `isBeforeFirst()`, `isFirst()`, `isLast()` lub `getRow()`.

Pobieranie wartości z kolumn

Bez względu na rodzaj zbioru wynikowego, do pobrania wartości z określonej kolumny służy jedna z metod `getXXX()` odpowiednia dla typu danych SQL przechowywanych w kolumnie. Pierwszym argumentem metody jest indeks kolumny, który jest względną pozycją kolumny w konstrukcji SQL (1 dla pierwszej, a następnie kolejno od lewej do prawej). Poniżej przedstawiono przykład (przy założeniu, że połączenie — `conn` — istnieje):

```
int           personId  = 0;
String        lastName  = null;
String        firstName = null;
java.sql.Date birthDate = null;
String        gender    = null;
int           rows      = 0;
ResultSet     rset      = null;
Statement     stmt      = null;
try {
  stmt = conn.createStatement();
  rset = stmt.executeQuery(
    "select person_id, " +
    "       last_name, " +
    "       first_name, " +
    "       birth_date, " +
    "       gender " +
    "from   PERSON " +
    "where  person_id = 1 ");
  while (rset.next()) {
```

```
      rows++;
      personId  = rset.getInt(1);
      lastName  = rset.getString(2);
      firstName = rset.getString(3);
      birthDate = rset.getDate(4);
      gender    = rset.getString(5);
      ...
    }
    rset.close();
    rset = null;
    stmt.close();
    stmt = null;
  }
  catch (SQLException e) {
    ...
  }
  finally {
    if (rset != null)
      try { rset.close(); } catch (Exception i) { }
    if (stmt != null)
      try { stmt.close(); } catch (Exception i) { }
  }
```

W powyższym przykładzie zmiennej `personId` typu `int` zostaje przypisana wartość znajdująca się w kolumnie `person_id` zbioru wynikowego. Kolumna ta występuje w zbiorze wynikowym jako pierwsza, stąd w wywołaniu metody `getInt()` podanym indeksem jest liczba 1. Zmiennej `lastName` typu `String` przypisana jest wartość z kolumny `last_name`, pobrana ze zbioru wynikowego za pomocą metody `getString()`, do której jako argument przekazano wartość 2.

Wstawianie, modyfikowanie i usuwanie wierszy za pomocą zbioru wynikowego

Jeżeli utworzony zbiór wynikowy umożliwia modyfikację, można za jego pomocą wstawiać nowe wiersze, modyfikować wartości w istniejących wierszach, a także usuwać wiersze.

Wstawienie nowego wiersza do modyfikowalnego zbioru wynikowego wymaga najpierw przeniesienia kursora do roboczego obszaru zwanego *insert row* (tam tworzy się nowy wiersz) za pomocą metody `moveToInsertRow()`. Następnie, za pomocą metod z rodziny `updateXXX()`, należy ustalić wartości w poszczególnych kolumnach. Właściwego wstawienia wiersza do tabeli dokonuje się przy użyciu metody `insertRow()`. Na końcu trzeba z powrotem ustawić kursor na rekordzie, na który wskazywał przed rozpoczęciem wstawiania — służy do tego metoda `moveToCurrentRow()`.

Aby zmodyfikować wiersz istniejący w zbiorze wynikowym, trzeba najpierw ustawić kursor na żądanym rekordzie, a następnie za pomocą metod `updateXXX()` wpisać do niego nowe wartości. Modyfikacji wiersza w tabeli dokonuje metoda `updateRow()`.

Przed usunięciem istniejącego wiersza poprzez zbiór wynikowy trzeba ustawić na wybranym wierszu kursor, a następnie wywołać metodę `deleteRow()`.

Poniższy przykład obrazuje wykonanie tych trzech operacji (przy założeniu, że połączenie — `conn` — istnieje):

```
ResultSet rset = null;
Statement stmt = null;
try {
  stmt = conn.createStatement(
    ResultSet.CONCUR_UPDATABLE,
    ResultSet.TYPE_SCROLL_SENSITIVE);
  rset = stmt.executeQuery(
    "select person_id, " +
    "       last_name, " +
    "       first_name, " +
    "       birth_date, " +
    "       gender " +
    "from   PERSON ");

  // Wstawienie wiersza.
  rset.moveToInsertRow();
  rset.updateInt(1, 3);
```

```
    rset.updateString(2, "DOUGH");
    rset.updateString(3, "PLAY");
    rset.updateNull(4);
    rset.updateNull(5);
    rset.insertRow();
    rset.moveToCurrentRow();

    // Zmodyfikowanie wiersza.
    rset.absolute(1);
    rset.updateString(2, "DOUGH");
    rset.updateRow();

    // Usunięcie wiersza.
    rset.absolute(1);
    rset.deleteRow();

    rset.close();
    rset = null;
    stmt.close();
    stmt = null;
  }
  catch (SQLException e) {
    ...
  }
  finally {
    if (rset != null)
      try { rset.close(); } catch (Exception i) { }
    if (stmt != null)
      try { stmt.close(); } catch (Exception i) { }
  }
```

Dynamiczne określanie charakterystyki zbioru wynikowego

Do określania charakterystyki zbioru wynikowego w trakcie wykonywania programu służy klasa `ResultSetMetaData`. Istnieje wiele sytuacji, w których może pojawić się konieczność określenia

cech otrzymanego zbioru. Na przykład, po wykonaniu dynamicznej konstrukcji SELECT, za pomocą obiektu ResultSetMetaData można ustalić, co tak naprawdę ta konstrukcja zwróciła.

Poniżej przedstawiono sposób uzyskiwania obiektu ResultSetMetaData dla bieżącego zbioru wynikowego za pomocą metody getMetaData() (przy założeniu, że połączenie — conn — istnieje):

```
int               cols = 0;
ResultSet         rset = null;
Statement         stmt = null;
ResultSetMetaData rsmd = null;
try {
  stmt = conn.createStatement();
  rset = stmt.executeQuery(
    "select person_id, " +
    "       last_name, " +
    "       first_name, " +
    "       birth_date, " +
    "       gender " +
    "from   PERSON " +
    "where  person_id = 1 ");
  while (rset.next()) {
    rows++;
    if (rows == 1) {
      rsmd = rset.getMetaData();
      cols = rsmd.getColumnCount();
      if (cols > 0) {
        for (int i = 1;i <= cols;i++) {
          if (i > 1) {
            System.out.print("\t" +
              rsmd.getColumnName(i));
          }
          else {
            System.out.print(
              rsmd.getColumnName(i));
          }
        }
        System.out.println("");
```

```
        }
      }
      if (cols > 0) {
        for (int i = 1;i <= cols;i++) {
          if (i > 1) {
            System.out.print("\t" +
              rset.getString(i));
          }
          else {
            System.out.print(
              rset.getString(i));
          }
        }
        System.out.println("");
      }
    }
    rset.close();
    rset = null;
    stmt.close();
    stmt = null;
  }
  catch (SQLException e) {
    ...
  }
```

Korzystając z obiektu `ResultSetMetaData`, program najpierw określa liczbę kolumn ujętych w zapytaniu SQL. Następnie, podczas przetwarzania pierwszego wiersza zbioru wynikowego, wypisuje nagłówki kolumn, używając nazw występujących w bazie danych. Potem program wyświetla kolejno wartości z poszczególnych wierszy. Informacje na temat wszystkich możliwości obiektu `ResultSetMetaData` znajdziesz w podrozdziale „ResultSetMetaData" w rozdziale „JDBC API".