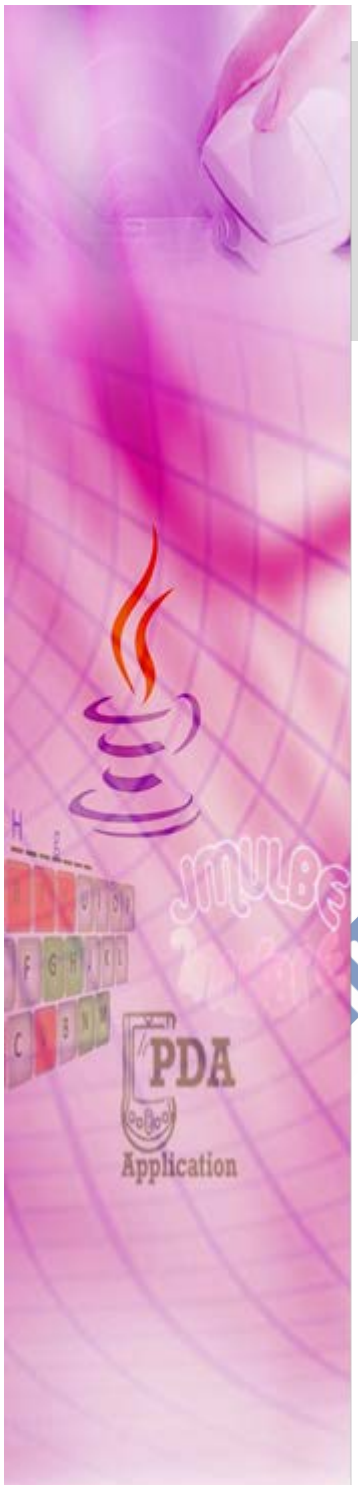


第 10 章

使用 JDBC 高级功能创建应用程序

您已学习如何使用 `java.sql.Statement` 对象插入、删除和更新数据库表。但是，通过使用该对象，您只能使用静态值执行插入、更新或删除操作。因此，为使用动态值，Java 提供了 `java.sql.PreparedStatement` 对象。

有时，在数据库中执行事务时，事务中的所有语句都应成功执行或失败。为实现该功能，Java 提供了数据库事务管理。此外，要提高应用程序的效率，Java 提供了批量更新。并且为了使用 SQL 过程和函数，Java 提供了 `java.sql.CallableStatement`。之后，为处理数据库和表的元数据，Java 提供了 `java.sql.DatabaseMetaData` 和 `java.sql.ResultSetMetaData`。

本章重点介绍使用 `java.sql.PreparedStatement` 创建应用程序，管理数据库事务，执行批量更新，使用 `java.sql.CallableStatement` 创建应用程序，以及创建应用程序以使用数据库和表的元数据。

目标

在本章中，您将学习：

- 使用 `PreparedStatement` 对象创建应用程序
- 管理数据库事务
- 实现 JDBC 中的批量更新
- 创建和调用 JDBC 中的存储过程
- 使用 JDBC 中的元数据


NIIT
training
china
.com

使用 PreparedStatement 对象创建应用程序

思考一个场景：一家出版公司 New Publishers 要在数据库中维护书籍和作者的详细信息。New Publishers 的管理层想要一款应用程序，用来根据不同条件访问有关作者的详细信息。例如，该应用程序应能够检索居住在运行时指定的特定城市的所有作者的详细信息。在该场景中，不能使用 Statement 对象检索详细信息，因为城市的值需要在运行时指定。您需要使用 PreparedStatement 对象，因为它能够接受运行时参数。

PreparedStatement 接口是从 Statement 接口派生而来的，可从 java.sql 包中获取。PreparedStatement 对象允许您将运行时参数传递到 SQL 语句以查询和修改表中的数据。

JDBC 仅编译并准备 PreparedStatement 对象一次。此后调用 PreparedStatement 对象不会重新编译 SQL 语句。这有助于减少数据库服务器上的负载，从而提高应用程序的性能。

PreparedStatement 接口的方法

PreparedStatement 接口从 Statement 接口继承以下方法以执行 SQL 语句：

- ResultSet executeQuery()：用于执行 SQL 语句并在 ResultSet 对象中返回结果。
- int executeUpdate()：执行 INSERT、UPDATE 或 DELETE 之类的 SQL 语句，并返回受影响的行数。
- boolean execute()：执行 SQL 语句并返回 boolean 值。

思考示例：您需要在运行时传递作者 id 来检索作者的详细信息。为此，可以使用具有参数化查询的以下 SQL 语句：

```
SELECT * FROM Authors WHERE au_id = ?
```

要将参数化查询从应用程序提交到数据库，您需要使用 Connection 对象的 prepareStatement() 方法创建 PreparedStatement 对象。您可以使用下面的代码段准备在运行时接受值的 SQL 语句：

```
stat=con.prepareStatement("SELECT * FROM Authors WHERE au_id = ?");
```

Connection 对象的 prepareStatement() 方法取 SQL 语句作为参数。SQL 语句可以包含符号 ? 作为占位符，可在运行时替换为输入参数。

在执行 PreparedStatement 对象中指定的 SQL 语句之前，必须设置每个 ? 参数的值。这些值可以通过调用相应的 setXXX() 方法设置，其中 XXX 是参数的数据类型。例如，思考以下代码段：

```
stat.setString(1,"A001");
ResultSet result=stat.executeQuery();
```

在上述代码段中，setString() 方法的第一个参数指定 ? 占位符的索引值，第二个参数用于设置 ? 占位符的值。当从用户界面获得 ? 占位符的值时，可以使用下面的代码段：

```
stat.setString(1,aid.getText());
ResultSet result=stat.executeQuery();
```

在上述代码段中，setString() 方法用于将 ? 占位符的值设置为运行时从用户界面的 aid 文本框中检索到的值。

PreparedStatement 接口提供各种方法来为特定数据类型设置占位符的值。下表列出 PreparedStatement 接口的一些常用方法。

方法	*描述*
void setByte(int index, byte val)	*为对应于指定索引的参数设置 Java byte 类型值。*
void setBytes(int index, byte[] val)	*为对应于指定索引的参数设置 Java byte 类型数组。*
void setBoolean(int index, boolean val)	*为对应于指定索引的参数设置 Java boolean 类型值。*
void setDouble(int index, double val)	*为对应于指定索引的参数设置 Java double 类型值。*
void setInt(int index, int val)	*为对应于指定索引的参数设置 Java int 类型值。*
void setLong(int index, long val)	*为对应于指定索引的参数设置 Java long 类型值。*
void setFloat(int index, float val)	*为对应于指定索引的参数设置 Java float 类型值。*
void setShort(int index, short val)	*为对应于指定索引的参数设置 Java short 类型值。*
void setString(int index, String val)	*为对应于指定索引的参数设置 Java String 类型值。*

PreparedStatement 接口的方法

检索行

您可以使用以下代码段通过 PreparedStatement 对象从 Books 表检索作者撰写的书籍的详细信息：

```
String str = "SELECT * FROM Books WHERE au_id = ?";
PreparedStatement ps= con.prepareStatement(str);
ps.setString(1, "A001");
ResultSet rs=ps.executeQuery();
while(rs.next())
{
System.out.println(rs.getString(1) + " " + rs.getString(2));
}
```

在上述代码段中，str 变量存储包含一个输入参数的 SELECT 语句。setString() 方法用于设置 Books 表的 au_id 属性的值。SELECT 语句是使用 executeQuery() 方法执行的，返回 ResultSet 对象。

插入行

您可以使用以下代码段创建 PreparedStatement 对象以通过在运行时传递作者数据在 Authors 表中插入行：

```
String str = "INSERT INTO Authors (au_id, au_name) VALUES (?,?)";
PreparedStatement ps = con.prepareStatement(str);
ps.setString(1, "1001");
ps.setString(2, "Abraham White");
int rt=ps.executeUpdate();
```

在上述代码段中，str 变量存储包含两个输入参数的 INSERT 语句。setString() 方法用于设置 Authors 表的 au_id 和 au_name 列的值。INSERT 语句是使用 executeUpdate() 方法执行的，返回指定表中插入行数的整数值。

更新和删除行

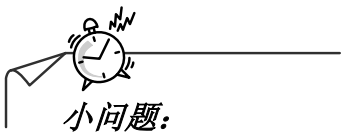
可以使用以下代码段，用 PreparedStatement 对象在 Authors 表中将城市为 Oakland 的州/省栏修改为 CA。

```
String str = "UPDATE Authors SET state= ?WHERE city= ?";
PreparedStatement ps = con.prepareStatement(str);
ps.setString(1, "CA");
ps.setString(2, "Oakland");
int rt=ps.executeUpdate();
```

在上述代码段中，两个输入参数 state 和 city 分别包含 Authors 表的 state 和 city 属性。

您可以使用以下代码段通过 PreparedStatement 对象从 Authors 表删除行，其中 au_name 是 Abraham White：

```
String str = "DELETE FROM Authors WHERE au_name= ?";
PreparedStatement ps = con.prepareStatement(str);
ps.setString(1, "Abraham White");
int rt=ps.executeUpdate();
```



小问题：

指出 PreparedStatement 接口的三个方法。

答案：

PreparedStatement 接口的三个方法有：

1. *ResultSet executeQuery()*
2. *int executeUpdate()*
3. *boolean execute()*

活动 10.1：创建使用 PreparedStatement 对象的应用程序

管理数据库事务

事务是一组作为一个单元执行的一个或多个 SQL 语句。仅当成功执行了事务中的所有 SQL 语句时，事务才完成。如果事务中的任何一个 SQL 语句失败，则整个事务就会回滚，从而维护数据库中数据的一致性。

JDBC API 为事务管理提供支持。例如，JDBC 应用程序用于从一个银行账户将资金转移到另一个账户。当钱从第一个账户中扣除，并添加到第二个账户中时，此事务完成。若处理 SQL 语句时出现错误，则这两个账户将维持不变。在两个账户之间转移资金的这组 SQL 语句表示 JDBC 应用程序中的事务。

可在 JDBC 应用程序中通过以下两种方式提交数据库事务：

- **隐式**：`Connection` 对象使用*自动提交*方式隐式执行 SQL 语句。自动提交模式指一旦 SQL 语句执行完毕，就自动提交事务中的每个 SQL 语句。 默认情况下，JDBC 应用程序中的所有事务语句都是自动提交的。
- **显式**：为在 JDBC 应用程序中显式提交事务语句，您需要使用 `setAutoCommit()` 方法。该方法接受 `true` 或 `false` 两个值中的任意一个，用于设置或重置数据库的自动提交模式。自动提交模式设置为 `false` 以显式提交事务。您可以使用以下代码段将自动提交模式设置为 `false`：

  ```
  con.setAutoCommit(false);
  ```

  在上述代码段中，`con` 表示 `Connection` 对象。

提交事务

当您将自动提交模式设为 `false` 时，SQL 语句执行的操作不会永久反映在数据库中。您需要显式调用 `Connection` 接口的 `commit()` 方法以反映数据库中事务所作的更改。`setAutoCommit(false)` 方法和 `commit()` 方法之间出现的所有 SQL 语句都视为一个事务并作为一个单元执行。

`rollback()` 方法用于撤消最后一个提交操作之后数据库中所作的更改。您需要显式调用 `rollback()` 方法来还原上一次提交状态中的数据库。当调用 `rollback()` 方法时，取消数据库中所有暂挂的事务，且数据库还原到它以前提交的状态。 您可以使用以下代码段调用 `rollback()` 方法：

```
con.rollback();
```

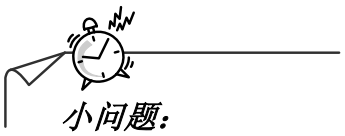
在上述代码段中，`con` 表示 `Connection` 对象。

您可以使用以下代码创建包含两个 `INSERT` 语句的事务，并使用 `commit()` 方法显式提交该事务：

```
import java.sql.*;
public class CreateTrans
{
public static void main(String arg[])
{
try
{
/*Initialize and load the Type 4 JDBC driver*/
Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
```

```
/*Establish a connection with a data source*/
try (Connection con =
DriverManager.getConnection("jdbc:sqlserver://sqlserver01;databaseName=Librar
y;user=user1;password=password#1234");)
{
/*Set the auto commit mode to false*/
con.setAutoCommit(false);

/*Create a transaction*/
try (PreparedStatement ps = con.prepareStatement("INSERT INTO Publishers
(pub_id, pub_name) VALUES (?, ?)");)
{
/*Specify the value for the placeholders in the PreparedStatement object*/
ps.setString(1, "P006");
ps.setString(2, "Darwin Press");
int firstctr = ps.executeUpdate();
System.out.println("First Row Inserted but not committed");

/*Insert a row in the database table using the prepareStatement() method*/
try (PreparedStatement ps2 = con.prepareStatement("INSERT INTO
Publishers(pub_id, pub_name) VALUES (?, ?)");)
{
ps2.setString(1, "P007");
ps2.setString(2, "MainStream Publishing");
int secondctr = ps2.executeUpdate();
System.out.println("Second Row Inserted but not committed");

/*Commit a transaction*/
con.commit();
System.out.println("Transaction Committed, Please check table for data");

}
}
}
}
catch (Exception e)
{
System.out.println("Error :" + e);
}
}
}
```

在上述代码中，使用 `setAutoCommit()` 方法将自动提交模式设置为 `false`。将自动提交模式设置为 `false` 之后执行的所有语句都被数据库视为事务。



小问题：

您可以如何显式地提交事务？

答案：

您可以通过将自动提交模式设置为 `false` *并使用* `commit()` *方法显式提交事务。*

实现 JDBC 中的批量更新

一个*批次*是发送到数据库作为一个单元执行的一组更新语句。可以使用相同的 Connection 对象在单个请求中将批发送到数据库。这可以减少应用程序和数据库之间的网络调用。因此，相对于处理单个 SQL 语句，处理批中的多个 SQL 语句是更有效的方法。

注释

JDBC 事务可以包括多个批。

Statement 接口提供以下方法创建和执行 SQL 语句的批处理：

- `void addBatch(String sql)`：向批添加 SQL 语句。
- `int[] executeBatch()`：将一个批次发送到数据库中进行处理，并返回更新的行的总数。
- `void clearBatch()`：从批中移除 SQL 语句。

您可以创建 Statement 对象来执行批量更新。当创建 Statement 对象时，空数组与此对象关联。您可以向空数组添加多个 SQL 语句，将其作为批来执行。在 JDBC 中处理批量更新时，需要使用 setAutoCommit(false) 方法禁用自动提交模式。这使您能够在批次中的任何 SQL 语句失败时，回滚使用批量更新执行的整个事务。您可以使用以下代码段创建一批 SQL 语句：

```
con.setAutoCommit(false);
Statement stmt=con.createStatement();
stmt.addBatch("INSERT INTO Publishers (pub_id, pub_name) VALUES (P001, 'Sage
Publications')");
stmt.addBatch("INSERT INTO Product (pub_id, pub_name) VALUES (P002, 'Prisidio
Press')");
```

在上述代码段中，con 是 Connection 对象。setAutoCommit() 方法用于将自动提交模式设为 false。批包含使用 addBatch() 方法添加到批的两个 INSERT 语句。

按 SQL 语句在批中出现的顺序处理批中的语句。您可以使用以下代码段执行一批 SQL 语句：

```
int[] updcount=stmt.executeBatch();
```

在上述代码段中，updcount 是存储 executeBatch() 方法返回的更新计数值的整数数组。更新计数是处理 SQL 语句时受影响的行的总数。如果它处理成功，则 executeBatch() 方法返回批中每个 SQL 语句的更新计数。

批量更新中的异常处理

批量更新操作可以抛出两类异常：SQLException 和 BatchUpdateException。处理数据库发生任何问题时，JDBC API 方法 addBatch() 和 executeBatch() 将抛出 SQLException。当您尝试使用 executeBatch() 方法执行 SELECT 语句时抛出 SQLException 异常。BatchUpdateException 类是从 SQLException 类派生的。当由于以下原因导致无法执行批中的 SQL 语句时，将抛出 BatchUpdateException 异常：

- SQL 语句中出现非法参数。
- 缺少数据库表。

BatchUpdateException 使用更新计数的数组来标识抛出异常的 SQL 语句。成功执行的所有 SQL 语句的更新计数在数组中的存储顺序与批中 SQL 语句的出现顺序相同。您可以遍历更新计数数组，以确定批中没有执行成功的 SQL 语句。更新计数的数组中的 null 值表示批中执行失败的 SQL 语句。您可以使用以下代码来使用 SQLException 类中可用于通过 BatchUpdateException 对象打印批中 SQL 语句的更新计数的方法：

```
import java.sql.*;

public class BatchUpdate
{
public static void main(String args[])
{
try
{
/*Batch Update Code comes here*/
}
catch (BatchUpdateException bexp)
{
/*Use the getMessage() method to retrieve the message associated with the
exception thrown*/
System.err.println("SQL Exception:"+ bexp.getMessage());
System.err.println("Update Counts:");
/*Use the getUpdateCount() method to retrieve the update count for each SQL
statement in a batch*/
int[] updcount = bexp.getUpdateCounts();
for (int i = 0; i <= updcount.length; i++)
{
/*Print the update count*/
System.err.println(updcount[i]);
}
}
}
}
```

创建使用批量更新在表中插入行的应用程序

您可以使用批量更新将 Statement 和 PreparedStatement 接口的多个对象作为 JDBC 应用程序中的批量执行。您可以使用下面的代码用批量更新在表中插入数据：

```
import java.sql.*;

public class BookInfo
{
public static void main(String args[])
{
try
{
/*Initialize and load Type 4 JDBC driver*/
Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
```

```
/*Connect to a data source using Library DSN*/
try (Connection con =
DriverManager.getConnection("jdbc:sqlserver://sqlserver01;databaseName=Librar
y;user=user1;password=password#1234");
/*Create a Statement object*/
Statement stmt = con.createStatement();)
{
con.setAutoCommit(false);

/*Add the INSERT statements to a batch*/
stmt.addBatch("INSERT INTO Books (book_id, book_name) VALUES ('B004', 'Kane
and Able')");
stmt.addBatch("INSERT INTO Books (book_id, book_name) VALUES ('B005', 'The
Ghost')");
stmt.addBatch("INSERT INTO Books (book_id, book_name) VALUES ('B006', 'If
Tommorrow Comes')");

/*Execute a batch using executeBatch() method*/
int[] results = stmt.executeBatch();
System.out.println("");
System.out.println("Using Statement object");
System.out.println("-------------------------");
for (int i = 0; i < results.length; i++)
{
System.out.println("Rows affected by " + (i + 1) + " INSERT statement:" +
results[i]);
}

/*Use the PreparedStatement object to perform batch updates*/
try (PreparedStatement ps = con.prepareStatement("INSERT INTO Books (book_id,
price) VALUES ( ?, ?)");)
{
System.out.println("");
System.out.println("---------------------------------");
System.out.println("Using PreparedStatement object");
System.out.println("---------------------------------");

/*Specify the value for the placeholders*/
ps.setString(1, "B007");
ps.setInt(2, 575);
ps.addBatch();
ps.setString(1, "B008");
ps.setInt(2, 350);

/*Add the SQL statement to the batch*/
ps.addBatch();

/*Execute the batch of SQL statements*/
int[] numUpdates = ps.executeBatch();
for (int i = 0; i < numUpdates.length; i++)
{
System.err.println("Rows affected by " + (i + 1) + " INSERT statement:" +
numUpdates[i]);
}
```

```
/*Commit the INSERT statements in the batch*/
con.commit();


}
}
}
catch (BatchUpdateException bue)
{
System.out.println("Error :" + bue);
}
catch (SQLException sqle)
{
System.out.println("Error :" + sqle);
}
catch (Exception e) {
System.out.println("Error :" + e);
}
}
}
```

在上述代码中，使用 `Statement` 和 `PreparedStatement` 对象创建两个批。`INSERT` 语句使用 `addBatch()` 方法添加到批，并使用 `executeBatch()` 方法执行。`executeBatch()` 方法返回数组，此数组存储在批中执行的所有 SQL 语句的更新计数。您可以使用 `for` 循环显示受到批中每个 SQL 语句影响的行数。

创建和调用 JDBC 中的存储过程

java.sql 包提供包含各种方法的 CallableStatement 接口，使您能够调用数据库存储过程。CallableStatement 接口派生自 PreparedStatement 接口。

创建存储过程

可以使用 JDBC 应用程序来创建存储过程。您可以使用 executeUpdate() 方法来执行 CREATE PROCEDURE SQL 语句。存储过程有两种类型：参数化存储过程和非参数化存储过程。

您可以使用以下代码段在 JDBC 应用程序中创建非参数化存储过程：

```
String str = "CREATE PROCEDURE Authors_info "
+"AS "
+ "SELECT au_id,au_name "
+ "FROM Authors "
+ "WHERE city = 'Oakland' "
+ "ORDER BY au_name";
Statement stmt=con.createStatement();
int rt=stmt.executeUpdate(str);
```

在上述代码段中，SELECT 语句指定按照 au_name 列中的顺序从 Authors 表检索数据。Connection 对象 con 用于将 CREATE PROCEDURE SQL 语句发送到数据库。在执行该代码时，将创建 Authors_info 存储过程并将其存储在数据库中。

您可以使用以下代码段创建参数化存储过程：

```
str = " CREATE PROCEDURE Authors_info_prmtz @auth_id varchar(15) ,@auth_name
varchar(20) output, @auth_city varchar(20) output,@auth_state varchar(20)
output "
+ " AS "
+ " SELECT @auth_name=au_name, @auth_city=city, @auth_state=state "
+ " FROM Authors "
+ " WHERE au_id=@auth_id ";
Statement stmt=con.createStatement();
int rt=stmt.executeUpdate(str);
```

在上述代码段中，创建了 Authors_info_prmtz 存储过程，接受作者 id 作为参数，并从数据库中检索相应的作者信息。检索到的信息存储在 OUT 参数中。output 关键字用于表示 OUT 参数。

存储过程可以接受一个或多个参数。存储过程的参数可以采取以下任意一种形式：

- IN：指传递给存储过程的参数。
- OUT：指存储过程的返回值。
- INOUT：结合了 IN 和 OUT 参数的功能。INOUT 参数使您可以将参数传递给存储过程。此参数也可用于存储存储过程的返回值。

注释

使用存储过程执行数据库操作时，可以减少网络流量，因为只需要执行单个存储过程，而不需要将多个 SQL 语句发送到数据库。

调用不带参数的存储过程

Connection 接口提供了用于创建 CallableStatement 对象的 prepareCall() 方法。此对象用于调用数据库的存储过程。prepareCall() 方法是具有多种格式的重载方法。常用的一些数值函数如下：

- CallableStatement prepareCall(String str)：创建 CallableStatement 对象调用存储过程。prepareCall() 方法接受字符串作为参数，其中包含调用存储过程的 SQL 语句。您还可以指定接受 SQL 语句中参数的占位符。
- CallableStatement prepareCall(String str, int resSetType, int resSetConcurrency)：创建 CallableStatement 对象，返回具有指定结果集类型和并发模式的 ResultSet 对象。该方法接受以下参数：
  - String 对象：包含调用存储过程的 SQL 语句。该 SQL 语句可以包含一个或多个参数。
  - ResultSet 类型：指定三种 Resultset 类型 TYPE_FORWARD_ONLY、TYPE_SCROLL_INSENSITIVE 或 TYPE_SCROLL_SENSITIVE 中的任意一个。
  - ResultSet 并发模式：为结果集指定并发模式 CONCUR_READ_ONLY 或 CONCUR_UPDATABLE。
- CallableStatement prepareCall(String str, int resSetType, int resSetConcurrency, int resSetHoldability)：创建 CallableStatement 对象以返回具有指定结果类型、并发模式和常量的 ResultSet 对象以设置结果集状态。

以下签名用于调用不带参数的存储过程：

```
exec <procedure_name>
```

您可以使用下面的代码段来调用不接受参数的存储过程：

```
String str = "exec Authors_info";
CallableStatement cstmt = con.prepareCall(str);
ResultSet rs = cstmt.executeQuery();
while (rs.next())
                {
System.out.println(" Author Id :" + rs.getString(1) + "\t");
System.out.println(" Author Name :" + rs.getString(2) + "\t");
}
```

在上述代码段中，con 是调用 prepareCall() 方法的 Connection 对象。str 变量包含对 Authors_info 存储过程的调用，该调用作为参数传递给 prepareCall() 方法。

调用带有参数的存储过程

SQL 转义语法用于调用带参数的存储过程。SQL 转义语法是从 RDBMS 调用存储过程的一种标准方式，它独立于 RDBMS。驱动程序在代码中搜索 SQL 转义语法，并将该 SQL 转义语法转换为数据库兼容的形式。有两种形式的 SQL 转义语法，一种包含结果参数，另一种则不包含结果参数。这两种形式都可以携带多个参数。如果 SQL 转义语法包含了结果参数，则此结果参数用于从存储过程返回值。结果参数是一个 OUT 参数。SQL 转义语法的其他参数可以包含 IN、OUT 或 INOUT 参数。SQL 转义语法的签名是：

```
{[?=] call <procedure_name> [<parameter1>,<parameter2>, ., <parameterN>]}
```

占位符用于表示过程调用中存储过程的 IN、OUT 和 INOUT 参数。调用带参数的存储过程的语法是：

```
{ call <procedure_name>(?)};
```

在执行 CallableStatement 对象之前，需要设置 IN 参数的值。否则，将在处理存储过程时抛出 SQLException 异常。set 方法用于指定 IN 参数的值。CallableStatement 接口继承来自 PreparedStatement 接口的 set 方法。设置 IN 参数的值的签名是：

```
<CallableStatement_object>.setInt(<value>);
```

在上述签名中，setInt() 方法用于设置整数类型的 IN 参数的值。

如果存储过程包含 OUT 和 INOUT 参数，则在处理对存储过程的调用之前，应用相应的 JDBC 类型注册这些参数。JDBC 类型决定了检索 OUT 和 INOUT 参数的值时 get 方法中使用的 Java 数据类型。registerOut() 方法用于注册参数。如果未注册表示 OUT 和 INOUT 参数的占位符，将抛出 SQLException。registerOut() 方法的原型为：

- registerOut(int index, int stype)：接受在 java.sql.Types 类中的常量和占位符的位置作为参数。java.sql.Types 类包含多种 JDBC 类型的常量。例如，如果您想要注册 VARCHAR SQL 数据类型，则应使用 java.sql.Types 类的 STRING 常量。您可以使用以下调用方法来调用 registerOut() 方法以注册参数：

  ```
  cstmt.registerOutParameter(1, java.sql.Types.STRING);
  ```

- registerOut(int index, int stype, int scale)：接受占位符位置、java.sql.Types 类中的常量和作为参数返回的值的小数位数。注册数值数据类型（如 NUMBER、DOUBLE 和 DECIMAL）时，需要定义参数的小数位数。例如，如果您想要注册具有三位小数的 DECIMAL SQL 数据类型，则小数位数参数值应为 three。您可以使用下面的代码段来指定调用 registerOut() 方法时的小数位数参数：

  ```
  cstmt.registerOutParameter(1, java.sql.Types.DECIMAL, 3);
  ```

可以使用 prepareCall() 方法来调用接受参数的存储过程。prepareCall() 方法在处理了过程主体中定义的 SQL 和控制语句后返回结果。您可以使用以下代码调用带参数的存储过程：

```
import java.sql.*;

public class CallProc
{
```

```
public static void main(String args[])
{
String id, name, address, city;
try
{
String str = "{call Authors_info_prmtz(?, ?, ?, ?)}";

/*Initialize and load Type 4 JDBC driver*/
Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
/*Establish a connection with the database*/
try (Connection con =
DriverManager.getConnection("jdbc:sqlserver://sqlserver01;databaseName=Librar
y;user=user1;password=password#1234");
/*Call a stored procedure*/
CallableStatement cstmt = con.prepareCall(str);)
{
/*Pass IN parameter*/
cstmt.setString(1, "A001");
/*Register OUT parameters*/
cstmt.registerOutParameter(2, Types.VARCHAR);
cstmt.registerOutParameter(3, Types.VARCHAR);
cstmt.registerOutParameter(4, Types.VARCHAR);
/*Process the stored procedure*/
cstmt.execute();
/*Retrieve Authors information*/
name = cstmt.getString(2);
address = cstmt.getString(3);
city = cstmt.getString(4);
/*Display author information*/
System.out.println("");
System.out.println("Displaying Author Information");
System.out.println("-----------------------------");
System.out.println("First name:" + name);
System.out.println("Address:" + address);
System.out.println("City:" + city);

}
}
catch (Exception e)
{
System.out.println("Error " + e);
}
}
}
```

在上述代码中，调用 `Authors_info_prmtz` 存储过程。此存储过程接受一个 `IN` 类型参数和四个 `OUT` 类型参数。`IN` 参数用于指定要检索其信息的作者的 id。`OUT` 参数用于检索作者的姓、名、地址和城市。`setString()` 方法用于指定作者 id，`registerOut()` 方法用于注册 `OUT` 参数。

使用 JDBC 中的元数据

*元数据*是数据的信息，例如表结构和属性。例如，数据库中的 `employee` 表包含 `name`、`address`、`salary`、`designation` 和 `department` 列。`employee` 表的元数据包括列名、每列的数据类型和约束等特定信息，以便在表的列中输入数据值。JDBC API 提供了以下两个元数据接口来检索关于数据库和结果集的信息：

- `DatabaseMetaData`
- `ResultSetMetaData`

使用 DatabaseMetaData 接口

`DatabaseMetaData` 接口提供让您确定数据库属性的方法。这些属性包括数据库表名、数据库版本、SQL 关键字和存储在数据库中的数据的隔离级别。

您可以使用 `Connection` 接口的 `getMetaData()` 方法创建 `DatabaseMetaData` 的对象。您可以使用以下代码段来创建 `DatabaseMetaData` 接口的对象：

```
DatabaseMetaData dm=con.getMetaData();
```

在上述代码段中，`con` 指 `Connection` 接口的对象。

`DatabaseMetaData` 接口中声明的方法检索特定于数据库的信息。下表列出了 `DatabaseMetaData` 接口的一些常用方法。

方法	*描述*
`ResultSet getColumns(String catalog, String schema, String table_name, String column_name)`	*检索指定数据库目录中可用的数据库表的列信息。*
`Connection getConnection()`	*检索创建 `DatabaseMetaData` 对象的数据库连接。*
`String getDriverName()`	*检索 `DatabaseMetaData` 对象的 JDBC 驱动程序的名称。*
`String getDriverVersion()`	*检索 JDBC 驱动程序的版本。*
`ResultSet getPrimaryKeys(String catalog, String schema, String table)`	*检索有关数据库表主键的信息。*

方法	*描述*
String getURL()	*检索数据库的 URL。*
boolean isReadOnly()	*返回 boolean 值，指示数据库是否为只读数据库。*
boolean supportsSavepoints()	*返回 boolean 值，指示数据库是否支持保存点。*

DatabaseMetaData 接口的方法

DatabaseMetaData 接口中的方法检索有关与 Java 应用程序连接的数据库的信息。您可以使用下面的代码，使用 DatabaseMetaData 接口的方法检索并显示各种数据库表的名称：

```
import java.sql.*;

public class TableNames
{
public static void main(String args[])
{
try {
/*Initialize and load the Type 4 driver*/
Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
/*Establish a connection with the database*/
try (Connection con =
DriverManager.getConnection("jdbc:sqlserver://sqlserver01;databaseName=Librar
y;user=user1;password=password#1234");)
{
/*Create a DatabaseMetaData object*/
DatabaseMetaData dbmd = con.getMetaData();
String[] tabTypes = {"TABLE"};
/*Retrieve the names of database tables*/
System.out.println("");
System.out.println("Tables Names");
System.out.println("----------------");
ResultSet tablesRS = dbmd.getTables(null, null, null, tabTypes);
while (tablesRS.next()) /*Display the names of database tables*/
{
System.out.println(tablesRS.getString("TABLE_NAME"));
}
}
}
catch (Exception e)
{
System.out.println("Error :" + e);
}
}
}
```

在上述代码中，与 Library 数据源建立连接。DatabaseMetaData 接口的对象是使用 getMetaData() 方法来声明的。DatabaseMetaData 对象用于使用 getTables() 方法检索数据库表的名称。

使用 ResultSetMetaData 接口

ResultSetMetaData 接口包含各种方法，使您能够检索有关结果集中数据的信息，如列数、列名和列的数据类型。ResultSet 接口提供了 getMetaData() 方法，可以创建 ResultSetMetaData 接口的对象。您可以使用以下代码段来创建 ResultSetMetaData 接口的对象：

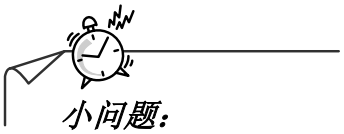
```
ResultSetMetaData rm=rs.getMetaData();
```

在上述代码段中，rs 引用 ResultSet 接口的对象。rs 调用 getMetaData() 方法以创建 ResultSetMetaData 接口的对象。

下表列出了 ResultSetMetaData 接口的一些常用方法。

方法	*描述*
int getColumnCount()	*返回整数，指示 ResultSet 对象中列的总数。*
String getColumnLabel(int column_index)	*检索对应于指定索引的表列的标题。*
String getColumnName(int column_index)	*检索对应于指定索引的表列的名称。*
int getColumnType(int column_index)	*检索对应于指定索引的表列的 SQL 数据类型。*
String getTableName(int column_index)	*检索包含对应于指定索引的列的数据库表的名称。*
boolean isAutoIncrement(int column_index)	*返回 boolean 值，指示对应于指定索引的表列是否自动递增。*
boolean isCaseSensitive(int column_index)	*返回 boolean 值，指示对应于指定索引的表列是否区分大小写。*
boolean isReadOnly(int column_index)	*返回 boolean 值，指示 ResultSet 列中对应于指定索引的列是否为只读列。*
boolean isWritable(int column_index)	*返回 boolean 值，指示对应于指定索引的 ResultSet 列是否可写。*

ResultSetMetaData 接口的方法



小问题：

检索数据库和结果集的有关信息使用哪个元数据接口？

答案：

检索数据库和结果集的有关信息使用的元数据接口是：

1.`DatabaseMetaData`

2.`ResultSetMetaData`

活动 10.2：创建应用程序以确定表的结构

练习问题

1. _____ 接口提供了方法，使您能够确定数据库或 RDBMS 的属性。

2. 指出 `PreparedStatement` 接口的方法中为对应于指定索引的参数设置 Java `byte` 类型值的方法。

   a. `setByte(int index, byte val)`
   b. `setBytes(int index, byte[] val)`
   c. `setString(int index, String val)`
   d. `setShort(int index, short val)`

3. 您为何需要在使用批量更新时禁用自动提交模式？

4. 从 RDBMS 调用存储过程的标准方式是什么？

5. `Connection` 接口的哪个方法用于创建 `callableStatement` 对象？

   a. `prepareCall()`
   b. `getMetaData()`
   c. `prepareStatement()`
   d. `createStatement()`

小结

在本章中，您学习了：

- PreparedStatement 对象允许你使用占位符将运行时参数传递到 SQL 语句。
- 在单个 SQL 语句中可以有多个占位符。与每个占位符相关联的索引值取决于 SQL 语句中占位符的位置。
- 占位符存储分配给它的值，直到此值被显式更改为止。
- 事务是一组作为一个单元执行的一个或多个 SQL 语句。仅当事务中的所有 SQL 语句都执行成功时，事务才完成。
- 如果 setAutoCommit() 方法设置为 true，SQL 语句执行的数据库操作会在数据库中自动提交。
- commit() 方法反映了由 SQL 语句在数据库中作出的永久更改。
- rollback() 方法用于撤消上次提交操作后执行的所有 SQL 操作的效果。
- 一个批次是发送到数据库作为一个单元执行的一组更新语句。可以使用相同的 Connection 对象作为单个请求将批发送到数据库。
- executeBatch() 方法返回整数数组，此数组存储在批中成功执行的所有 SQL 语句的更新计数。更新计数是受每个 SQL 语句执行的数据库操作影响的数据库行的数目。
- 批量更新操作可以抛出两类异常：SQLException 和 BatchUpdateException。
- 当数据库访问出现问题时，会抛出 SQLException。在批中执行返回 ResultSet 对象的 SELECT 语句时，也抛出 SQLException。
- 由于无法访问指定表或 SQL 语句中出现非法参数而导致无法执行批中的 SQL 语句时，会抛出 BatchUpdateException。
- CallableStatement 接口包含使您能够从数据库中调用存储过程的各种方法。
- 存储过程的参数可以采取以下三种形式中的任意一种：
  - IN
  - OUT
  - INOUT
- 元数据是有关数据的信息，如表的结构和属性。
- JDBC API 提供两种元数据接口来检索有关数据库和结果集 DatabaseMetaData 和 ResultSetMetaData 的信息。
- DatabaseMetaData 接口声明让您确定数据库属性的方法。
- ResultSetMetaData 接口声明让您能够确定结果集的信息的方法。
- Connection 接口的 getMetaData() 方法使您能够获得 DatabaseMetaData 接口的对象。DatabaseMetaData 接口中的方法仅检索有关与 Java 应用程序连接的数据库的信息。
- ResultSet 接口的 getMetaData() 方法，使您能够创建 ResultSetMetaData 接口的实例。