# JDBC Introduction

JDBC API是一种可以访问任何类型表格数据，特别是存储在关系数据库中的数据的Java API。

JDBC帮你编写一个Java应用程序，用来管理以下三种编程活动。

1. 连接数据源比如一个数据库。
2. 向数据库发送查询和更新语句。
3. 接收并处理从数据库收到的作为对你查询回应的结果。

以下简单的代码片段给出了以上三个步骤的简单示例：

public void connectToAndQueryDatabase(String username, String password) {

Connection con = DriverManager.getConnection(

"jdbc:myDriver:myDatabase",

username,

password);

Statement stmt = con.createStatement();

ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table1");

while (rs.next()) {

int x = rs.getInt("a");

String s = rs.getString("b");

float f = rs.getFloat("c");

}

}

这个代码片段实例化一个DriverManager对象来连接到数据库驱动并登录数据库，它实例化一个Statement对象，将SQL查询语句传递给数据库。实例化一个ResultSet对象，用来获取你的查询结果，并且执行一个简单的while循环，该循环获取并显示这些结果。

## JDBC Product Components

JDBC包括四个组件：

1. **The JDBC API** —JDBC™API提供了从Java™编程语言到关系型数据的编程访问。使用JDBC API，应用程序可以执行SQL语句，获取结果，并将更改传播到底层数据源。JDBC API还可以在分布式异构环境中与多个数据源进行交互。

JDBC API是Java平台的一部分，它包括Java™标准版（Java™SE）和Java™企业版（Java™EE）。JDBC 4.0 API分为两个包：java.sql和javax.sql。这两个包都包含在Java SE和Java EE平台中。

1. **JDBC Driver Manager** —JDBC DriverManager类用来确定可将Java应用程序连接到JDBC驱动程序的对象。在传统上DriverManager已经是JDBC架构的骨干。它相当小而简单。

标准扩展包javax.naming 和javax.sql允许你用Java命名和目录接口（*Java Naming and Directory Interface*）™（JNDI）的命名服务注册一个DataSource对象，使用用它来建立与数据源的连接。您可以使用其他的连接机制，但建议尽可能使用DataSource对象。

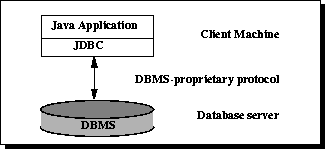
1. **JDBC Test Suite —** JDBC驱动测试套件可帮助您确定JDBC驱动将运行您的程序。这些测试并不全面和具体，但它们检测了许多JDBC API的重要功能。
2. **JDBC-ODBC Bridge** —Java Software桥接器通过ODBC驱动程序提供JDBC访问。请注意，您需要将ODBC二进制代码加载到使用此驱动程序的每个客户端计算机上。因此，ODBC驱动程序最适合在一个公司网络，这样客户端安装不是主要问题，或者是在一个在三层架构中服务端代码是用Java编写的应用程序上。

本教程使用这四个JDBC组件的前两个连接到数据库，然后构建一个使用SQL命令与测试关系型数据库进行通信的java程序。最后两个组件用于专门的环境中以测试Web应用程序，或与ODBC-aware DBMS进行通信。

## JDBC Architecture

JDBC API支持用来数据库访问的两层和三层处理模型。

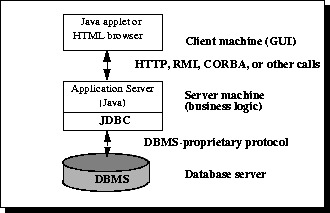
***Figure 1: Two-tier Architecture for Data Access.***



在两层模型中，Java applet或应用程序直接与数据源进行通信。这需要一个能够与被访问的特定数据源通信的JDBC驱动。它将用户的命令传递给数据库或其他数据源，并将这些语句的结果发送回用户。数据源可能部署在其他的机器上，用户需要通过网络进行连接。这被称为客户端/服务器配置，用户的机器是客户端，而托管数据源的机器是服务器。网络可以是内联网，例如在公司内部将员工连接起来，也可以是互联网。

在三层模型中，命令被发送给服务的中间层，然后它再将命令发给数据源。数据源处理命令并且将结果发回中间层，然后中间层再将结果发给用户。在管理信息系统中三层模型非常有吸引力，因为中间层可以保持对访问的控制以及对企业数据可以进行的更新种类。另一个优点是它简化了应用程序的部署。最后，在许多情景下，三层架构可以提供更好的性能优势。

***Figure 2: Three-tier Architecture for Data Access.***



直到最近，中间层还经常用C或C ++等语言编写，这提供了更快的性能。然而，随着能将Java字节码转换为高效的机器特定代码的编译器的优化，以及一些技术比如Enterprise JavaBeans™的引进，Java平台正在迅速成为中间层开发的标准平台。这是一个很大的加分，使它可以利用Java的强大，多线程和安全功能。

随着使用Java编程语言来编写服务端代码的企业的增长，JDBC API正在三层架构的中间层中被越来越多地使用。JDBC的支持连接池，分布式事务以及disconnected rowsets这些功能使其成为了一项服务器技术。JDBC API同样也允许从Java中间层访问数据源。

# JDBC Basics

# Processing SQL Statements with JDBC

通常，要使用JDBC处理任何SQL语句，请按照下列步骤操作：

1. 建立连接。
2. 创建statement
3. 执行查询
4. 处理ResultSet对象
5. 关闭连接

这节使用下面的CoffeesTables.viewTable方法来演示这些步骤。此方法输出COFFEES表里的内容。在后面的章节中会讨论该方法更多细节。

public static void viewTable(Connection con, String dbName)

throws SQLException {

Statement stmt = null;

String query = "select COF\_NAME, SUP\_ID, PRICE, " +

"SALES, TOTAL " +

"from " + dbName + ".COFFEES";

try {

stmt = con.createStatement();

ResultSet rs = stmt.executeQuery(query);

while (rs.next()) {

String coffeeName = rs.getString("COF\_NAME");

int supplierID = rs.getInt("SUP\_ID");

float price = rs.getFloat("PRICE");

int sales = rs.getInt("SALES");

int total = rs.getInt("TOTAL");

System.out.println(coffeeName + "\t" + supplierID +

"\t" + price + "\t" + sales +

"\t" + total);

}

} catch (SQLException e ) {

JDBCTutorialUtilities.printSQLException(e);

} finally {

if (stmt != null) { stmt.close(); }

}

}

## Establishing Connections

首先与你要使用的数据源建立一个连接。数据源可以是一个DBMS，一个传统的文件系统，或者其他具有相应JDBC驱动的数据源。连接由Connection对象表示。更多信息参阅“建立连接”中。

## Creating Statements

Statement是一个代表一个SQL语句的接口。你执行Statement对象，并生成ResultSet对象，它是一个表示数据库结果集的数据表。要创建Statament对象，你需要一个Connection对象。

比如，CoffeesTables.viewTable用以下代码创建了一个Statement对象。

stmt = con.createStatement();

有以下三种形式的statement：

* Statement：用来实现简单的不带参数的SQL语句
* PrepareStatement：（继承自Statement）用来预编译可能包含输入参数的SQL语句。更多信息参阅“使用预编译语句”。
* CallableStatement：（继承自PreparedStatement）用来执行可能同时包含输入输出参数的存储过程。更多信息参阅“存储过程”

## Executing Queries

要执行查询操作，调用Statement中的execute方法，如下所示

* execute：如果查询返回的第一个对象是ResuleSet对象的话，则返回true。如果查询可能返回一个或多个ResultSet对象的话使用这个方法。可以通过反复调用Statement.getResultSet方法，获取查询返回的ResulteSet对象。
* executeQuery：返回一个Resultset对象。
* executeUpdate：返回一个表示受SQL语句影响行数的整数值。当执行INSERT,DELETE,或者UPDATE这些SQL语句时，使用该方法。

例如，CoffeesTables.viewTable使用以下代码执行Statement对象：

ResultSet rs = stmt.executeQuery(query);

更多信息参阅“从ResulteSet中获取和修改值”

## Processing ResultSet Objects

你在ResultSet对象中通过一个游标来访问数据。注意这个游标不是一个数据库游标。这个游标是一个指针指向了ResultSet对象中的一行数据。最初，游标位于第一行之前。你可以调用定义在ResultSet中的多种方法来移动这个游标。

比如CoffeesTables.viewTable方法反复调用ResultSet.next来将游标向前移动一行。每次调用next，这个方法都输出一个游标目前所在行的数据。

try {

stmt = con.createStatement();

ResultSet rs = stmt.executeQuery(query);

while (rs.next()) {

String coffeeName = rs.getString("COF\_NAME");

int supplierID = rs.getInt("SUP\_ID");

float price = rs.getFloat("PRICE");

int sales = rs.getInt("SALES");

int total = rs.getInt("TOTAL");

System.out.println(coffeeName + "\t" + supplierID +

"\t" + price + "\t" + sales +

"\t" + total);

}

}

// ...

更多信息参阅“从ResulteSet中获取和修改值”

## Closing Connections

当你结束使用Statement时，调用Statement.close方法来立即释放它使用的资源。当你调用这个方法时，它的ResultSet对象也被关闭。

比如，CoffeesTables.viewTable方法通过使用finally块包裹执行关闭的代码，来确保Statement对象在方法结束时被关闭，无论过程中是否有任何一个SQLException被抛出，

} finally {

if (stmt != null) { stmt.close(); }

}

当与数据源交互遇到错误的时候，JDBC会抛出一个SQLException异常。更多信息参阅“处理SQL异常”

在Java SE 7或更高版本可用的JDBC4.1中，你可以使用try-with-resources语句来自动关闭Connection，Statement以及ResultSet对象，无论是否有SQLException抛出。一个自动关闭资源语句由一个try语句以及一个或多个声明的资源组成。比如，比可以修改CoffeesTables.viewTable方法使它的Statement对象可以自动关闭，如下所示。

public static void viewTable(Connection con) throws SQLException {

String query = "select COF\_NAME, SUP\_ID, PRICE, " +

"SALES, TOTAL " +

"from COFFEES";

**try (Statement stmt = con.createStatement()) {**

ResultSet rs = stmt.executeQuery(query);

while (rs.next()) {

String coffeeName = rs.getString("COF\_NAME");

int supplierID = rs.getInt("SUP\_ID");

float price = rs.getFloat("PRICE");

int sales = rs.getInt("SALES");

int total = rs.getInt("TOTAL");

System.out.println(coffeeName + ", " + supplierID +

", " + price + ", " + sales +

", " + total);

}

} catch (SQLException e) {

JDBCTutorialUtilities.printSQLException(e);

}

}

下面是一个try-with-resources语句，它声明了一个资源stmt，当try块终止时，它将自动关闭：

try (Statement stmt = con.createStatement()) {

// ...

}

# Establishing a Connection

首先，你要与你要想要使用的数据源建立一个连接。数据源可以是一个DBMS，一个传统的文件系统，或者其他具有相应JDBC驱动的数据源。通常，一个JDBC应用程序使用以下两个类来与一个目标数据源建立连接。

* DriverManager：这个已经完全实现的类将应用程序连接到一个由数据库URL指定的数据源上。当这个类第一次尝试建立一个连接的时候，它会自动加载在class路径下的任何JDBC4.0驱动。需要注意的是，在4.0版本之前你的程序必须手动加载任何一个JDBC驱动。
* Datesource：这个接口优于DriverManager，因为它允许底层数据源的详细信息对你的程序透明。DataSource对象的属性设置为使其代表特定的数据源。更多信息参阅”使用Datasource对象连接“。有关使用DataSource类开发应用程序的更多信息，请参阅最新的Java EE教程。

**注意**：本教程中的示例使用DriverManager类而不是DataSource类，因为它更易于使用，并且示例不需要DataSource类的功能。

## Using the DriverManager Class

使用DriverManager类连接到您的DBMS涉及调用DriverManager.getConnection方法。以下JDBCTutorialUtilities.getConnection方法建立了一个数据库连接。

public Connection getConnection() throws SQLException {

Connection conn = null;

Properties connectionProps = new Properties();

connectionProps.put("user", this.userName);

connectionProps.put("password", this.password);

if (this.dbms.equals("mysql")) {

conn = DriverManager.getConnection(

"jdbc:" + this.dbms + "://" +

this.serverName +

":" + this.portNumber + "/",

connectionProps);

} else if (this.dbms.equals("derby")) {

conn = DriverManager.getConnection(

"jdbc:" + this.dbms + ":" +

this.dbName +

";create=true",

connectionProps);

}

System.out.println("Connected to database");

return conn;

}

DriverManager.getConnection方法用来建立一个数据库连接。这个方法需要一个数据库URL，它取决于您的DBMS。下面是一些数据库URL的示例。

1. Mysql：jdbc:mysql://localhost:3306/，localhost是你的数据库的服务主机名称，3306是端口号
2. Java DB：jdbc:derby:testdb;create=true,testdb]是要连接到的数据库的名字，而create = true则指示DBMS创建数据库。

**注意**：此URL与Java DB Embedded Driver建立数据库连接。Java DB还包括使用不同URL的Network Client Driver。

这个方法还需要指定用户名和密码，这里使用Properties对象用来访问DBMA。

**注意：**

* 通常在数据库URL中，你还需要指定你想连接的一个已经存在的数据库名称，URL jdbc:mysql://localhost:3306/mysql表示一个名为mysql 的MySQL数据库的URL。本教程中的示例使用不指定特定数据库的URL，因为本示例想要创建一个新的数据库。
* 在之前的JDBC版本，要想获取一个连接，你首先必须通过调用Class.forName方法来初始化你的JDBC驱动。该方法需要一个java.sql.Driver类型的对象。每个JDBC驱动都包含一个或多个实现java.sql.Driver接口的类。Java DB的driver类是org.apache.derby.jdbc.EmbeddedDriver 以及org.apache.derby.jdbc.ClientDriver，而MySQL Connector / J中的一个是com.mysql.jdbc.Driver. 请参阅DBMS驱动程序的文档，以获取实现接口java.sql.Driver的类的名称。

任何在class路径下找到的JDBC4.0的驱动都会被自动加载。（然而，你必须使用Class.forName方法手动加载任何早于JDBC4.0的驱动）

这个方法返回一个Connection对象，它代表一个同DBMS或者指定数据库的连接。通过这个对象来查询数据库。

## Specifying Database Connection URLs

一个数据库连接URL是一个你的DBMS JDBC驱动用来连接数据库的字符串。它可能包含这些信息，比如去哪里查找数据库，要连接的数据库名称，以及配置属性。数据库连接URL的具体语法由你的DBMS来指定。

### Java DB Database Connection URLs

以下是Java DB的数据库连接URL语法：

jdbc:derby:[subsubprotocol:][databaseName]

[;attribute=value]\*

* subsubprotocol指定Java DB应在目录，内存，类路径或JAR文件中搜索数据库的位置。通常省略。
* databaseName是要连接的数据库的名称。
* attribute = value表示可选的以分号分隔的属性列表。这些属性使您能够指示Java DB执行各种任务，包括以下内容：
  + - 创建连接URL中指定的数据库。
    - 加密连接URL中指定的数据库。
    - 指定目录以存储日志记录和跟踪信息。
    - 指定连接到数据库的用户名和密码。

### MySQL Connector/J Database URL

以下是MySQL Connector / J的数据库连接URL语法：

jdbc:mysql://[host][,failoverhost...]

[:port]/[database]

[?propertyName1][=propertyValue1]

[&propertyName2][=propertyValue2]...

* host:port是托管数据库的主机名和端口号。如果没有指定，host和port的默认值分别是127.0.0.1和3306。
* database是你要连接到的数据库的名称。如果没有指定，则连接不具有默认数据库。
* failover是备用数据库的名称（MySQL Connector / J支持故障切换）。
* propertyName = propertyValue表示一个可选的，和用用&符分隔的属性列表。这些属性使您能够指示MySQL Connector / J执行各种任务。

# Connecting with DataSource Objects

本节涵盖了DataSource对象，它是连接到数据源的首选方式。除了它的会在后面介绍的优点之外，Datasource对象还可以提供连接池和分布式事务。此功能对企业数据库计算至关重要。特别地，它是Enterprise JavaBeans（EJB）技术的组成部分

本章节展示如何使用DataSource接口来获取一个连接，以及如何使用分布式事务与连接池。这两者在你的JDBC应用程序中只涉及很少的代码修改。

系统管理员通常使用一个工具（比如Apache Tomcat 或 Oracle WebLogic Server）执行部署这些类来实现以上的功能操作，但这会根据正在部署的DataSource对象的类型而有所不同。因此，本节的大部分内容都致力于显示系统管理员如何设置环境，以便程序员可以使用DataSource对象获取连接。

## Using DataSource Objects to Get a Connection

在” Establishing a Connection“中，你学会了如何使用DriverManager类来获取一个连接。这个章节将会为你展示如何使用DataSource对象来获取一个连接，到你的数据源中，这是一个更好的方式。

由实现DataSource的类的实例化对象表示特定的DBMS或某些其他数据源，比如一个文件。一个DataSource对象表示一个特定的DBMS或者其他的数据源，比如一个文件。如果一个公司使用超过一个数据源的话，它会为其中的每个都部署一个单独的DataSource对象。DataSource接口由驱动程序供应商来实现。它可能以以下三种方式来实现。

* 一个基本的DataSource实现提供标准的Connection对象，这些对象不在连接池中或者在分布式事务中使用。
* 支持连接池的DataSource实现产生参与在连接池中的Connection对象，也就是说，连接可以被回收。
* 支持分布式事务的DataSource实现产生可在分布式事务中使用的Connection对象，也就是说，一个事务可以访问两个或多个DBMS服务器。

一个JDBC驱动应该至少包含一个基本的DataSource实现。比如，JavaDB JDBC驱动包含实现org.apache.derby.jdbc.ClientDataSource，并且对于MySQL，也包含一个DataSource实现com.mysql.jdbc.jdbc2.optional.MysqlDataSource. 如果您的客户端运行在Java 8 Compact Profile 2上，那么Java DB JDBC驱动程序就是org.apache.derby.jdbc.BasicClientDataSource40. 本教程的示例需要compact profile 3或更高版本。

一个支持分布式事务的DataSource类通常也会实现对连接池的实现。例如，由EJB供应商提供的DataSource类几乎总是支持连接池和分布式事务。

在之前的例子中，假设The Coffee Break商店蓬勃发展的连锁店的所有者决定通过在互联网上销售咖啡来进一步扩张。随着预期的大量在线交易，所有者无疑需要一个连接池。连接的开启和关闭涉及大量开销，并且所有者预计这种在线订购系统将需要大量的查询和更新。通过连接池，池中的连接可以一次又一次地使用，从而避免为每次数据库访问来创建新连接的开销。此外，所有者现在有第二个DBMS,其中包含最近收购的咖啡焙烧公司的数据。这意味着所有者会希望能够编写同时使用旧的DBMS服务器和新的服务器的分布式事务。

连锁经营者重新配置了计算机系统，为新的更加广大的客户群服务。所有者已经购买了最新的JDBC驱动程序和一个EJB应用程序服务器，这可以用来做分布式事务，并且能够获得连接池带来的性能增长。许多JDBC驱动程序可用于与最近购买的EJB服务器兼容。所有者现在拥有了一个三层架构，一个新的EJB应用程序服务器和JDBC驱动在中间层，并且两个DBMS服务器作为第三层。发出请求的客户端计算机是第一层。

## Deploying Basic DataSource Objects

系统管理员需要部署DataSource对象，这样The Coffee Break的编程团队可以开始使用它们。部署一个DataSource对象包含以下三个任务。

1. 创建一个DataSource类的实例
2. 设置它的属性。
3. 使用Java命名和目录接口（JNDI）API的命名服务注册它。

首先，考虑使用一个DataSource接口基本实现的最基础情况，也就是一个不支持连接池或分布式事务的DataSource。这种情况下只有一个DataSource对象需要部署。一个DataSource的基础实现提供与DriverManager类提供的相同种类的连接。

### Creating Instance of DataSource Class and Setting its Properties

假设一个公司只想要一个DataSource的基础实现，它已经从JDBC供应商DB Access Inc购买了一个驱动。这个驱动包含了com.dbaccess.BasicDataSource类，它实现了DataSource接口。下面的代码片段创建了一个BasicDataSource类的实例，并且设置了它的属性。在BasicDataSource的实例部署后，程序员可以调用DataSource.getConnection方法来获取一个到公司数据库CUSTOMER\_ACCOUNTS的连接。首先系统管理员使用默认构造器创建了BasicDataSource对象ds。然后系统管理员设置了三个属性。注意，以下代码通常由部署工具执行。

com.dbaccess.BasicDataSource ds = new com.dbaccess.BasicDataSource();

ds.setServerName("grinder");

ds.setDatabaseName("CUSTOMER\_ACCOUNTS");

ds.setDescription("Customer accounts database for billing");

变量ds现在表示托管在在服务器上的数据库CUSTOMER\_ACCOUNTS。任何由BasicDataSource对象ds产生的连接都是到数据库CUSTOMER\_ACCOUNTS的连接。

### Registering DataSource Object with Naming Service That Uses JNDI API

通过设置属性，系统管理员可以使用JNDI（Java命名和目录接口）命名服务注册BasicDataSource对象。所使用的命名服务通常由系统属性确定，这里不展示。下面代码片段注册BasicDataSource对象并且将它与逻辑名称jdbc/billingDB绑定在一起

Context ctx = new InitialContext();

ctx.bind("jdbc/billingDB", ds);

这里的代码使用了JNDI API。第一行创建了一个InitialContext对象，服务器把它作为命名起点，类似于一个文件系统的根目录。第二行将BasicDataSource对象ds关联或者说绑定到逻辑名jdbc/billingDB上。在下个代码片段，你传递给命名服务这个逻辑名，然后它返回给你BasicDataSource对象。逻辑名可以是任何字符串。在这种情况下，公司决定使用名称billingDB作为CUSTOMER\_ACCOUNTS数据库的逻辑名称。

在前面的示例中，jdbc是初始上下文中的一个子上下文，就像在根目录下的目录是一个子目录一样。Jdbc/billingDB就像一个路径名称，路径中最后一个项目类似于文件名。在这种情况下，billingDB是赋予BasicDataSource对象ds的逻辑名称。子上下文jdbc保留用于绑定到DataSource对象的逻辑名，因此jdbc将始终是数据源的逻辑名的第一部分。

### Using Deployed DataSource Object

在一个基础DataSource实现被系统管理员部署后，它已经可以被程序员所使用。这意味着程序员可以给出一个被绑定到DataSource类实例上的逻辑数据源名称，然JNDI命名服务会返回一个DataSource类的实例。DataSource对象上的getConnection方法可以被调用，用来获取一个到它所表示的数据源上的连接。比如，一个程序员可以编写以下两行代码来获取一个DataSource对象，它产生一个到数据库CUSTOMER\_ACCOUNTS的连接。

Context ctx = new InitialContext();

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

第一行代码获取一个初始上下文作为接收DataSource对象的起点。当你提供这个逻辑名jdbc/billingDB给方法lookup，这个方法返回被系统管理员部署时绑定到jdbc/billingDB上的DataSource对象。因为lookup方法的返回值是一个Java Object类型，所以在将它赋给变量ds前，我们必须将它转换为更为具体的DataSource类型。

变量ds是实现了DataSource接口的类com.dbaccess.BasicDataSource的实例。调用方法ds.getConnection产生一个到数据库CUSTOMER\_ACCOUNTS的连接。

Connection con = ds.getConnection("fernanda","brewed");

getConnection方法只需要用户名和密码，因为在变量ds的属性中有与数据库CUSTOMER\_ACCOUNTS建立连接所需的其他信息，比如数据库名称和位置。

**Advantages of DataSource Objects**

由于它能设置属性，与获取连接的DriverManager类相比，DataSource对象是一个更好的选择。程序员不再需要在其应用程序中硬编码驱动程序名称或JDBC URL，这使得它们更易移植。此外，DataSource的属性使维护代码更简单。如果有更改，系统管理员可以更新数据源属性，而不用担心更改与数据源建立连接的每个应用程序。例如，如果将数据源移动到其他服务器，所有系统管理员需要做的是设置serverName属性为新的服务器名称。

除了可移植性和易于维护之外，使用DataSource对象获取连接还可以提供其他优势。当DataSource接口实现为使用ConnectionPoolDataSource实现时，由该DataSource类的实例生成的所有连接将自动成为池连接。类似地，当DataSource的实现也是使用XADataSource的实现时，它产生的所有连接将自动成为可以在分布式事务中使用的连接。下一节将介绍如何部署这些类型的DataSource实现。

**Deploying Other DataSource Implementations**

系统管理员或相同身份的人可以部署DataSource对象，使其生成的连接是池连接。要想做到这一点，他或她首先部署一个ConnectionPoolDataSource对象，然后部署一个DataSource对象，实现和它一起工作。设置ConnectionPoolDataSource对象的属性，以便它表示要生成连接的数据源。在ConnectionPoolDataSource对象已经使用JNDI命名服务注册之后，DataSource对象被部署。通常只需要为DataSource对象设置两个属性：description和dataSourceName。赋给dataSourceName属性的值是标识先前部署的ConnectionPoolDataSource对象的逻辑名称，该ConnectionPoolDataSource对象包含进行连接所需的属性。

在ConnectionPoolDataSource和DataSource对象部署后，你可以调用DataSource对象上的方法DataSource.getConnection来获取池连接。此连接将连接到ConnectionPoolDataSource对象的属性中指定的数据源。

以下示例描述了The Coffee Break的系统管理员如何部署实现的提供池连接的DataSource对象。系统管理员通常将使用部署工具，因此本节中显示的代码片段是部署工具将执行的代码。

为了获得更好的性能，Coffee Break公司已经从DB Access，Inc.购买了一个JDBC驱动程序，其中包含实现ConnectionPoolDataSource接口的com.dbaccess.ConnectionPoolDS类。系统管理员创建创建此类的实例，设置其属性，并使用JNDI命名服务进行注册。Coffee Break从其EJB服务器供应商Application Logic公司购买了DataSource类的实现com.applogic.PooledDataSource。com.applogic.PooledDataSource类通过使用ConnectionPoolDataSource类com.dbaccess.ConnectionPoolDS提供的底层支持来实现连接池。

ConnectionPoolDataSource对象必须首先被部署。下面代码创建了一个com.dbaceccesss.ConnectionPoolDS类的实例并且设置了它的属性。

com.dbaccess.ConnectionPoolDS cpds = new com.dbaccess.ConnectionPoolDS();

cpds.setServerName("creamer");

cpds.setDatabaseName("COFFEEBREAK");

cpds.setPortNumber(9040);

cpds.setDescription("Connection pooling for " + "COFFEEBREAK DBMS");

在ConnectionPoolDataSource对象被部署后，系统管理员部署DataSource对象。下面的代码使用JNDI命名服务来注册com.dbaccess.ConnectionPoolDS对象cpds。注意，与变量cpds相关联的逻辑名让它的子上下文pool添加到了子上下文jdbc下面，这类似于在一个分层文件系统中添加一个子目录到另一个子目录下。com.dbaccess.ConnectionPoolDS类的任何实例的逻辑名会一直以jdbc/pool开头。Oracle建议将所有ConnectionPoolDataSource对象放在子上下文jdbc / pool下：

Context ctx = new InitialContext();

ctx.bind("jdbc/pool/fastCoffeeDB", cpds);

接下来，一个被实现与cpds以及其他com.dbaccess.ConnectionPoolDS类实例交互的DataSource对象被部署。以下代码创建此类的实例并设置其属性。注意，com.applogic.PooledDataSource的实例只设置了两个属性。Description属性被设置，是因为它一直都是需要的。另一个被设置的属性是dataSourceName，它为com.dbaccess.ConnectionPoolDS 类的实例cpds提供一个逻辑JNDI名称。换句话说，cpds表示为DataSource对象实现连接池的ConnectionPoolDataSource对象。

下面的代码可能会由一个部署工具来执行，创建一个PooledDataSource对象，设置它的属性，并且将它绑定在逻辑名jdbc/fastCoffeeDB上。

com.applogic.PooledDataSource ds = new com.applogic.PooledDataSource();

ds.setDescription("produces pooled connections to COFFEEBREAK");

ds.setDataSourceName("jdbc/pool/fastCoffeeDB");

Context ctx = new InitialContext();

ctx.bind("jdbc/fastCoffeeDB", ds);

此时，部署了一个DataSource对象，应用程序可以从该对象获取到数据库COFFEEBREAK的池连接。

**Getting and Using Pooled Connections**

连接池是一个数据库连接对象的缓存。这些对象表示物理的数据库连接，可以由应用程序用于连接到数据库。在运行时期，应用程序从池中请求获取一个连接。如果池内包含可以满足请求的连接，则返回给应用程序这个连接。如果池内没有连接，则会创建一个新的连接并返回给应用程序。应用程序使用连接来在数据库上执行一些操作，然后返回连接对象给连接池。这样连接可用于下一个连接请求。

连接池促进连接对象的重用，并减少创建连接对象的次数。连接池显着提高了数据库密集型应用程序的性能，因为创建连接对象在时间和资源方面都是昂贵的。

现在DataSource和ConnectionPoolDataSource对象都已经被部署，程序员可以使用DataSource对象获得一个池连接。获取池连接的代码就像获取非池连接的代码一样，如以下两行所示：

ctx = new InitialContext();

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

变量ds表示DataSource对象，它产生与数据库COFFEEBREAK的池连接。你只需要获取这个DataSource对象一次，因为你可以使用它按照需要产生许多池连接。在ds变量上调用getConnection方法会自动生成一个池连接，因为ds变量所表示的DataSource对象被配置为产生池连接。

连接池通常对程序员是透明的。使用池化连接时，只需要做两件事情：

1. 使用DataSource对象而不是DriverMananger类来获取一个连接。在下面一行代码中，ds是实现和部署的DataSource对象，因此它将创建池连接，username和password是表示可以访问数据库的用户凭据的变量：

Connection con = ds.getConnection(username, password);

1. 使用finally语句来关闭一个池连接。下面的finally块出现在应用于使用池连接的代码的try / catch块之后：

try {

Connection con = ds.getConnection(username, password);

// ... code to use the pooled

// connection con

} catch (Exception ex {

// ... code to handle exceptions

} finally {

if (con != null) con.close();

}

除此之外，使用池连接的应用程序与使用常规连接的应用程序相同。程序员可能唯一注意到的一件事是，当连接池使用后性能会更好。

以下示例代码获取一个DataSource对象，该对象生成与数据库COFFEEBREAK的连接，并使用它来更新表COFFEES中的价格：

import java.sql.\*;

import javax.sql.\*;

import javax.ejb.\*;

import javax.naming.\*;

public class ConnectionPoolingBean implements SessionBean {

// ...

public void ejbCreate() throws CreateException {

ctx = new InitialContext();

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

}

public void updatePrice(float price, String cofName,

String username, String password)

throws SQLException{

Connection con;

PreparedStatement pstmt;

try {

con = ds.getConnection(username, password);

con.setAutoCommit(false);

pstmt = con.prepareStatement("UPDATE COFFEES " +

"SET PRICE = ? " +

"WHERE COF\_NAME = ?");

pstmt.setFloat(1, price);

pstmt.setString(2, cofName);

pstmt.executeUpdate();

con.commit();

pstmt.close();

} finally {

if (con != null) con.close();

}

}

private DataSource ds = null;

private Context ctx = null;

}

此代码示例中的连接参与连接池，因为：

* 一个实现ConnectionPoolDataSource的类的实例已经被部署。
* 一个实现DataSource的类的实例已经被部署，并且设置了它的dataSource属性值为我们绑定到之前部署的ConnectionPoolDataSource对象的逻辑名。

请注意，虽然这段代码与以前见过的代码非常相似，但它在以下方面是不同的：

* 除了java.sql外，它还导入了javax.sql,javax.ejb以及javax.naming包。

DataSource和ConnectionPoolDataSource接口在javax.sql包中，而JNDI构造器InitialContext和Context.lookup方法是javax.naming包中的一部分。该特定示例代码采用EJB组件的形式，该组件使用javax.ejb包中的API。此示例的目的是显示你使用池连接的方式与使用非池连接的相同，因此您不用担心不了解EJB API。

* 它使用DataSource对象来获取连接，而不是使用DriverManager。
* 它使用finally块来确保连接已关闭。

获取和使用池化连接与获取和使用常规连接类似。当作为系统管理员的人员正确部署了一个ConnectionPoolDataSource对象和一个DataSource对象时，应用程序使用该DataSource对象来获取一个池化的连接。应用程序应该使用finally块来关闭池化连接。为了简单起见，前面的例子使用了一个finally块，但没有catch块。如果try块中的方法抛出异常，它会依照默认抛出异常，而finally子句将在任何情况下执行。

**Deploying Distributed Transactions**

DataSource对象可以被部署用来获取可在分布式事务中使用的连接。和连接池一样，有两个不同的类实例必须被部署：一个XADataSource对象和一个DataSource对象需要被实现来配合使用。

假设Coffee Break企业家购买的EJB服务器包括DataSource类com.applogic.TransactionalDS，它与XADataSource类，如com.dbaccess.XATransactionalDS，配合工作。事实上，它适用于任何使EJB服务器可以跨JDBC驱动程序移植的XADataSource类。当DataSource和XADataSource对象部署后，生成的连接将能够参与分布式事务。在这个情况下com.applogic.TransactionalDS类同样被实现用来提供池连接。这种DataSource类实现情况通常被作为EJB服务实现的一部分而提供。

XADataSource对必须首先被部署。下面代码创建类一个com.dbaccess.XATransactionalDS的实例，并且设置了它的属性。

com.dbaccess.XATransactionalDS xads = new com.dbaccess.XATransactionalDS();

xads.setServerName("creamer");

xads.setDatabaseName("COFFEEBREAK");

xads.setPortNumber(9040);

xads.setDescription("Distributed transactions for COFFEEBREAK DBMS");

下面代码使用JNDI命名服务来注册com.dbaccess.XATransactionalDS对象xads。注意，与xads相关联的逻辑名称有一个在jdbc下的子上下文xa。Oracle建议com.dbaccess.XATransactionalDS类的任何实例的逻辑名称始终以jdbc / xa开头。

Context ctx = new InitialContext();

ctx.bind("jdbc/xa/distCoffeeDB", xads);

接下来，一个实现与xads以及其他XADataSource对象交互的DataSource对象被部署。请注意，DataSource类，com.applogic.TransactionalDS可以与任何JDBC驱动程序供应商提供的XADataSource类配合使用。部署DataSource对象涉及创建一个com.applogic.TransactionalDS类的实例并设置其属性。dataSourceName属性设置为jdbc / xa / distCoffeeDB，这是与com.dbaccess.XATransactionalDS关联的逻辑名。它是实现DataSource类的分布式事务功能的XADataSource类。下面的代码部署了一个DataSource类的实例。

com.applogic.TransactionalDS ds = new com.applogic.TransactionalDS();

ds.setDescription("Produces distributed transaction " +

"connections to COFFEEBREAK");

ds.setDataSourceName("jdbc/xa/distCoffeeDB");

Context ctx = new InitialContext();

ctx.bind("jdbc/distCoffeeDB", ds);

现在com.applogic.TransactionalDS 和 com.dbaccess.XATransactionalDS的实例都已经被部署，应用程序可以在TransactionalDS类的实例上调用getConnection方法，以获得可以在分布式事务中使用的COFFEEBREAK数据库的连接。

**Using Connections for Distributed Transactions**

要获得可用于分布式事务的连接，必须使用已正确实施和部署的DataSource对象，如“Deploying Distributed Transactions”部分所示。使用这样的DataSource对象，调用它上面的getConnection方法。在你获取连接后，使用它就像你使用任何其他的连接一样。因为jdbc / distCoffeesDB在JNDI命名服务中已经和XADataSource对象相关联，所以以下代码生成可用于分布式事务的Connection对象：

Context ctx = new InitialContext();

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

Connection con = ds.getConnection();

当你使用这个连接，而它属于分布式事务的一部分的时候，这儿有一些微小但是重要的限制。事务管理器控制分布式事务何时开始以及何时提交或回滚; 因此，应用程序代码不应该调用Connection.commit或Connection.rollback方法。应用程序也应该永远不会调用Connection.setAutoCommit（true），它启用自动提交模式，因为这也会干扰事务管理器对事务边界的控制。这就解释了为什么默认情况下，在分布式事务范围内创建的新连接的自动提交模式被禁用。请注意，这些限制仅适用于连接参与分布式事务时;当连接不是在分布式事务中时，是没有这个限制的。

对于以下示例，假设咖啡订单已发货，这会触发位于不同DBMS服务器上的两个表的更新。第一个表是一个新的INVENTORY表，第二个表是COFFEES表。因为这些表位于不同的DBMS服务器上，涉及它们的事务将是分布式事务。以下示例的代码中获取一个连接，更新了COFFEES表并关闭了连接，这是分布式事务的第二部分。

请注意，代码没有显式提交或回滚更新，因为分布式事务的范围由中间层服务器的基础系统基础架构控制。此外，假设用于分布式事务的连接也是池连接，则应用程序使用finally块关闭连接。这保证了即使一个异常被抛出，这个有效的连接也会被关闭，从而确保将连接返回到连接池以进行回收。

下面的代码演示了一个enterprise Bean，这是一个实现可以由客户端计算机调用的方法的类。此示例的目的是演示分布式事务的应用程序代码与其他代码没有任何区别，但它不调用Connection方法commit，rollback或setAutoCommit（true）。因此，您不需要担心了解所使用的EJB API。

import java.sql.\*;

import javax.sql.\*;

import javax.ejb.\*;

import javax.naming.\*;

public class DistributedTransactionBean implements SessionBean {

// ...

public void ejbCreate() throws CreateException {

ctx = new InitialContext();

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

}

public void updateTotal(int incr, String cofName, String username,

String password)

throws SQLException {

Connection con;

PreparedStatement pstmt;

try {

con = ds.getConnection(username, password);

pstmt = con.prepareStatement("UPDATE COFFEES " +

"SET TOTAL = TOTAL + ? " +

"WHERE COF\_NAME = ?");

pstmt.setInt(1, incr);

pstmt.setString(2, cofName);

pstmt.executeUpdate();

stmt.close();

} finally {

if (con != null) con.close();

}

}

private DataSource ds = null;

private Context ctx = null;

}

**Handling SQLExceptions**

**Overview of SQLException**

当JDBC在与数据源交互期间遇到错误时，它会抛出SQLException异常而不是Exception。（此上下文中的数据源表示Connection对象连接到的数据库。）SQLException实例包含以下信息，可以帮助您确定错误的原因：

* 错误的描述。通过调用方法SQLException.getMessage来获取包含此描述的String对象。
* 一个SQLState码。这些代码及其各自的含义已经通过ISO / ANSI和Open Group（X / Open）标准化，尽管已经为数据库供应商保留了一些代码来为自己定义。此String对象由五个字母字符组成。通过调用SQLException.getSQLState方法来获取此码。
* 一个错误码。它是一个整数值，标识导致抛出SQLException实例的错误。它的值和含义是特定实现的，并且可能是底层数据源返回的实际错误代码。通过调用SQLException.getErrorCode方法来获取错误。
* 一个引发原因。SQLException实例可能具有因果关系，它由一个或多个导致抛出SQLException实例的Throwable对象组成。要浏览这个原因链，递归调用SQLException.getCause方法，直到返回一个空值。
* 一个异常链的引用。如果发生多个错误，则通过此链引用异常。通过调用抛出异常的SQLException.getNextException方法来检索这些异常。

## Retrieving Exceptions

以下方法，JDBCTutorialUtilities.printSQLException输出SQLException中包含的SQLState，错误代码，错误描述和引发原因（如果有的话）以及链向此异常的任何其他异常：

public static void printSQLException(SQLException ex) {

for (Throwable e : ex) {

if (e instanceof SQLException) {

**if (ignoreSQLException(**

**((SQLException)e).**

**getSQLState()) == false) {**

e.printStackTrace(System.err);

System.err.println("SQLState: " +

((SQLException)e).getSQLState());

System.err.println("Error Code: " +

((SQLException)e).getErrorCode());

System.err.println("Message: " + e.getMessage());

Throwable t = ex.getCause();

while(t != null) {

System.out.println("Cause: " + t);

t = t.getCause();

}

}

}

}

}

例如，如果您使用Java DB作为DBMS调用CoffeesTable.dropTable方法，而表COFFEES不存在，并删除对JDBCTutorialUtilities.ignoreSQLException的调用，输出将类似于以下内容：

SQLState: 42Y55

Error Code: 30000

Message: 'DROP TABLE' cannot be performed on

'TESTDB.COFFEES' because it does not exist.

相比输出SQLException信息，你可以首先获取SQLState，然后相应地处理SQLException。例如，如果SQLState等于代码42Y55，方法JDBCTutorialUtilities.ignoreSQLException返回true（并且您正在使用Java DB作为您的DBMS），这将导致JDBCTutorialUtilities.printSQLException忽略这个SQLException：

public static boolean ignoreSQLException(String sqlState) {

if (sqlState == null) {

System.out.println("The SQL state is not defined!");

return false;

}

// X0Y32: Jar file already exists in schema

if (sqlState.equalsIgnoreCase("X0Y32"))

return true;

// 42Y55: Table already exists in schema

if (sqlState.equalsIgnoreCase("42Y55"))

return true;

return false;

}

**Retrieving Warnings**

SQLWarning对象是SQLException的子类，用来处理数据库访问警告。警告不会停止应用程序的运行，而异常会；警告只是简单的提醒用户有些事情没有按计划发生。例如，警告可能会让您知道您尝试撤销的权限未被撤销，或者告诉你在请求断开期间发生了一个错误。

一个警告可能是由一个Connection对象，一个Statement对象（包括PreparedStatement和CallableStatement对象），或者一个ResultSet对象报告。这些类都有一个getWarnings方法，为了看到调用对象报告的第一个警告消息，你必须调用该方法。如果getWarning返回一个警告，可以调用SQLWarning的getNextWarning方法来获取任何其他警告。执行一条语句会自动清空来自上一个语句中的警告，所以它们不会累积。但是，这意味着，如果要获取在语句上报告的警告，则必须在执行其他语句之前执行此操作。

JDBCTutorialUtilities中的以下方法说明了如何获取有关Statement或ResultSet对象上报告的任何警告的完整信息：

public static void getWarningsFromResultSet(ResultSet rs)

throws SQLException {

JDBCTutorialUtilities.printWarnings(rs.getWarnings());

}

public static void getWarningsFromStatement(Statement stmt)

throws SQLException {

JDBCTutorialUtilities.printWarnings(stmt.getWarnings());

}

public static void printWarnings(SQLWarning warning)

throws SQLException {

if (warning != null) {

System.out.println("\n---Warning---\n");

while (warning != null) {

System.out.println("Message: " + warning.getMessage());

System.out.println("SQLState: " + warning.getSQLState());

System.out.print("Vendor error code: ");

System.out.println(warning.getErrorCode());

System.out.println("");

warning = warning.getNextWarning();

}

}

最常见的警告是DataTruncation警告，它是SQLWarning的子类。所有DataTruncation对象的SQLState为01004，表示读取或写入数据有问题。DataTruncation方法可以让您了解哪个列或参数数据被截断，截断是读取还是写入操作，应该传输多少字节以及实际传输的字节数。

**Categorized SQLExceptions**

您的JDBC驱动程序可能会抛出与常见SQLState或常见错误状态相对应的SQLException子类，这不与特定的SQLState类值相关联。这使您能够编写更易移植的错误处理代码。这些异常是以下类之一的子类：

* SQLNonTransientException
* SQLTransientException
* SQLRecoverableException

有关这些子类的更多信息，请参阅java.sql包的最新Javadoc或JDBC驱动程序的文档。

**Other Subclasses of SQLException**

SQLException的以下子类也可以被抛出：

* BatchUpdateException在批量更新操作发生错误的时候会抛出。除SQLException提供的信息外，BatchUpdateException还提供在发生错误之前执行的所有语句的更新数。
* SQLClientInfoException在Connection上无法设置一个或多个客户端信息属性时会被抛出。除SQLException提供的信息外，SQLClientInfoException还提供未设置的客户端信息属性列表。

**Setting Up Tables**

**Retrieving and Modifying Values from Result Sets**

以下方法，CoffeesTable.viewTable输出COFFEES表的内容，并演示使用ResultSet对象和游标：

public static void viewTable(Connection con, String dbName)

throws SQLException {

Statement stmt = null;

String query =

"select COF\_NAME, SUP\_ID, PRICE, " +

"SALES, TOTAL " +

"from " + dbName + ".COFFEES";

try {

stmt = con.createStatement();

ResultSet rs = stmt.executeQuery(query);

while (rs.next()) {

String coffeeName = rs.getString("COF\_NAME");

int supplierID = rs.getInt("SUP\_ID");

float price = rs.getFloat("PRICE");

int sales = rs.getInt("SALES");

int total = rs.getInt("TOTAL");

System.out.println(coffeeName + "\t" + supplierID +

"\t" + price + "\t" + sales +

"\t" + total);

}

} catch (SQLException e ) {

JDBCTutorialUtilities.printSQLException(e);

} finally {

if (stmt != null) { stmt.close(); }

}

}

ResultSet对象是表示数据库结果集的数据表，通常通过执行查询数据库的语句生成。例如，CoffeeTables.viewTable方法通过Statement对象stmt执行查询时创建一个ResultSet对象rs。请注意，可以通过实现Statement接口的任何对象来创建一个ResultSet对象，包括PreparedStatement，CallableStatement和RowSet。

您可以通过游标访问ResultSet对象中的数据。请注意，此游标不是数据库游标。此游标是指向ResultSet中的一行数据的指针。最初，游标位于第一行之前。 ResultSet.next方法将游标移动到下一行。如果游标位于最后一行之后，此方法返回false。该方法使用while循环重复调用ResultSet.next方法来遍历ResultSet中的所有数据。

**ResultSet Interface**

ResultSet接口提供检索和操作执行查询后获得的结果的方法，ResultSet对象可以具有不同的功能和特性。这些特性包括类型，并发和光标保持性。

**ResultSet Types**

ResultSet对象的类型在两个方面确定其功能的级别：可以操纵游标的方式，以及ResultSet对象是否反映对底层数据源进行的并发更改。

ResultSet对象的敏感性由以下三个ResulteSet类型来决定：

* TYPE\_FORWARD\_ONLY：结果集不可滚动；它的游标只能向前移动，从第一行之前到最后一行。结果集中包含的行取决于底层数据库如何生成结果。也就是说，它包含在执行查询时或在检索行时满足查询的行。
* TYPE\_SCROLL\_INSENSITIVE：结果集可以滚动；其游标可以相对于当前位置向前和向后移动，并且它可以移动到一个任意的位置。但结果集在处于开启状态时，它对底层数据源的修改不敏感。它包含在执行查询时或在检索行时满足查询的行。
* TYPE\_SCROLL\_SENSITIVE：结果集可以滚动。其游标可以相对于当前位置向前和向后移动，并且它可以移动到一个任意的位置。在结果集保持开启状态时，它会反应底层数据源的修改。

默认的ResultSet类型为TYPE\_FORWARD\_ONLY。

**注意：**不是所有的数据库和JDBC驱动都支持ResultSet类型。如果支持指定的ResultSet类型，则DatabaseMetaData.supportsResultSetType的方法返回true，否则返回false。

**ResultSet Concurrency**

ResultSet对象的并发性决定了它支持的更新功能等级。

它有两个并发等级。

* CONCUR\_READ\_ONLY：ResultSet对象无法使用ResultSet接口进行更新。
* CONCUR\_UPDATABLE：可以使用ResultSet接口更新ResultSet对象。

默认的ResultSet并发性是CONCUR\_READ\_ONLY。

**注意**：不是所有的数据库和JDBC驱动都支持并发。如果驱动程序支持指定的并发级别，则DatabaseMetaData.supportsResultSetConcurrency方法返回true，否则返回false。

方法CoffeesTable.modifyPrices演示了如何使用并发级别为CONCUR\_UPDATABLE的ResultSet对象。

**Cursor Holdability**

调用Connection.commit方法可以关闭在当前事务中创建的ResultSet对象。然而，在某些情况下，这可能是不希望的行为。ResultSet的光标保持属性允许应用程序控制在调用commit时是否关闭ResultSet对象（游标）。

可以将以下ResultSet常量提供给Connection的createStatement，prepareStatement和prepareCall方法：

* HOLD\_CURSORS\_OVER\_COMMIT：ResultSet游标未关闭;它们是可以保留的：当调用方法commit时，它们被保持打开。可持续游标对只读ResultSet对象的应用程序是理想的。
* CLOSE\_CURSORS\_AT\_COMMIT：当调用commit方法时，ResultSet对象（游标）被关闭。调用此方法时关闭游标可以为某些应用程序带来更好的性能。

默认的游标保持性取决于您的DBMS。

**注意**：不是所有的数据库和JDBC驱动都支持游标的保持或非保持性。以下方法JDBCTutorialUtilities.cursorHoldabilitySupport输出ResultSet对象的默认游标可保持性，以及是否支持HOLD\_CURSORS\_OVER\_COMMIT以及CLOSE\_CURSORS\_AT\_COMMIT：。

public static void cursorHoldabilitySupport(Connection conn)

throws SQLException {

DatabaseMetaData dbMetaData = conn.getMetaData();

System.out.println("ResultSet.HOLD\_CURSORS\_OVER\_COMMIT = " +

ResultSet.HOLD\_CURSORS\_OVER\_COMMIT);

System.out.println("ResultSet.CLOSE\_CURSORS\_AT\_COMMIT = " +

ResultSet.CLOSE\_CURSORS\_AT\_COMMIT);

System.out.println("Default cursor holdability: " +

dbMetaData.getResultSetHoldability());

System.out.println("Supports HOLD\_CURSORS\_OVER\_COMMIT? " +

dbMetaData.supportsResultSetHoldability(

ResultSet.HOLD\_CURSORS\_OVER\_COMMIT));

System.out.println("Supports CLOSE\_CURSORS\_AT\_COMMIT? " +

dbMetaData.supportsResultSetHoldability(

ResultSet.CLOSE\_CURSORS\_AT\_COMMIT));

}

**Retrieving Column Values from Rows**

ResulteSet接口声明了getter方法（比如getBoolean和getLong）用来从当前行内获取列值。你可以使用列的索引号或者列的别名来获取值。使用列的索引通常更有效率。列从1开始编号。为了实现最大的可移植性，每行中的结果集列应以从左到右的顺序读取，每列应只读一次。

例如，以下CoffeesTable.alternateViewTable方法通过序号检索列值：

public static void alternateViewTable(Connection con)

throws SQLException {

Statement stmt = null;

String query =

"select COF\_NAME, SUP\_ID, PRICE, " +

"SALES, TOTAL from COFFEES";

try {

stmt = con.createStatement();

ResultSet rs = stmt.executeQuery(query);

while (rs.next()) {

String coffeeName = rs.getString(1);

int supplierID = rs.getInt(2);

float price = rs.getFloat(3);

int sales = rs.getInt(4);

int total = rs.getInt(5);

System.out.println(coffeeName + "\t" + supplierID +

"\t" + price + "\t" + sales +

"\t" + total);

}

} catch (SQLException e ) {

JDBCTutorialUtilities.printSQLException(e);

} finally {

if (stmt != null) { stmt.close(); }

}

}

字符串用作getter方法的输入时不区分大小写。当一个getter方法使用字符串作为输入被调用，并当有超过一列数据拥有相同字符串形式的别名或者名称时，只有第一个匹配的列的值被返回。使用字符串而不是整数的选项被设计为在生成结果集的SQL查询中使用列别名和名称时使用。对于查询中未明确命名的列（例如，select \* from COFFEES），最好使用列号。当使用列名时，开发人员应保证通过使用列别名来唯一地引用预期的列。列的别名可以有效地重命名结果集的列。要指定列别名，请使用SELECT语句中的SQL AS子句。

使用适当类型的getter方法来获取每列的值。比如，在方法CoffeeTables.viewTable中，在ResulteSet rs中每一行的第一列时COF\_NAME，它存储一个SQL类型VARCHAR的值。获取SQL类型VARCHAR的值的方法时getString。每一行的第二列存储一个SQL类型INTEGER的值，获取该值的方法时getInt。

请注意，尽管方法getString建议用于获取SQL类型CHAR和VARCHAR，但是可以使用它获取任何基本的SQL类型。使用getString获取所有值可能非常有用，但也有其局限性。例如，如果它用于检索数字类型，getString将数值转换为Java String对象，并且该值必须转换回数字类型才能作为数字操作。在值被视为字符串的情况下，它没有任何缺点。此外，如果您希望应用程序检获取SQL3类型之外的任何标准SQL类型的值，请使用getString方法。

**Cursors**

如前面提到，你可以通过一个游标访问在ResultSet对象中的数据，该游标指向ResultSet对象的一行。然而，当一个ResultSet对象首次被创建的时候，游标指向第一行之前。方法CoffeeTables.viewTable通过调用ResultSet.next方法移动光标。还有其他方法可以移动光标：

* next：将光标向前移动一行。如果光标现在位于行上，则返回true，如果光标位于最后一行之后，则返回false。
* previous：将光标向后移动一行。如果光标现在位于行上，则返回true，如果光标位于第一行之前，则返回false。
* first: 将游标移动到ResultSet对象的第一行。如果游标现在位于第一行，则返回true，如果ResultSet对象不包含任何行，则返回false。
* last：将游标移动到ResultSet对象中的最后一行。如果游标现在位于最后一行，则返回true，如果ResultSet对象不包含任何行，则返回false。
* beforFirst: 将游标放置在第一行之前的ResultSet对象的开头。如果ResultSet对象不包含任何行，则此方法不起作用。
* afterLast：将游标定位在ResultSet对象的最后一行之后。如果ResultSet对象不包含任何行，则此方法不起作用。
* relative（int rows）：相对于当前位置移动光标。
* absolute（int row）：将光标定位在由参数行指定的行上。

请注意，ResultSet的默认敏感度为TYPE\_FORWARD\_ONLY，这意味着它不能被滚动; 如果您的ResultSet无法滚动的话，除了next，你不能调用任何这些移动光标的方法；下面部分描述的CoffeesTable.modifyPrices方法演示了如何移动ResultSet的光标。

**Updating Rows in ResultSet Objects**

您不能更新一个默认级别的ResultSet对象，默认级别只能将其光标向前移动。但是，您可以创建可以滚动（光标可以向后移动或移动到绝对位置）并可以更新的ResultSet对象。

以下方法，CoffeesTable.modifyPrices，将每行的PRICE列乘以一个参数percentage：

public void modifyPrices(float percentage) throws SQLException {

Statement stmt = null;

try {

stmt = con.createStatement();

stmt = con.createStatement(ResultSet.TYPE\_SCROLL\_SENSITIVE,

ResultSet.CONCUR\_UPDATABLE);

ResultSet uprs = stmt.executeQuery(

"SELECT \* FROM " + dbName + ".COFFEES");

while (uprs.next()) {

float f = uprs.getFloat("PRICE");

uprs.updateFloat( "PRICE", f \* percentage);

uprs.updateRow();

}

} catch (SQLException e ) {

JDBCTutorialUtilities.printSQLException(e);

} finally {

if (stmt != null) { stmt.close(); }

}

}

ResultSet.TYPE\_SCROLL\_SENSITIVE字段创建一个光标可以相对于当前位置向前和向后移动，或者移动到一个绝对位置的ResultSet对象。

ResultSet.updateFloat方法更新指定的列（在本示例中，是在游标现在指向的行中带有指定float值的PRICE）。ResultSet包含各种更新方法，是你可以更新各种数据类型的类值。但是，这些更新方法都不会修改数据库;您必须调用ResultSet.updateRow方法来更新数据库。

**Using Statement Objects for Batch Updates**

Statement，PreparedStatement和CallableStatement对象有一系列与它们自身相关的命令列表。这些命令可能包括更新，插入或者删除行的语句。并且它也可以包含DDL语句，比如CREATE,TABLE以及DROP TABLE。但是，它不包含能够生成ResultSet对象的语句，比如SELECT语句。换句话说，这些命令列表只能包含产生更新计数的语句。

与Statement相关联的命令列表在它创建的时候，最初是空的。您可以使用addBatch方法向此列表添加SQL命令，并使用clearBatch方法将其清空。当您完成向列表添加语句时，调用executeBatch方法将它们全部发送到数据库，以作为一个单元或则批处理执行。

CoffeesTable.batchUpdate使用批量更新向COFFEES表添加了四行：

public void batchUpdate() throws SQLException {

Statement stmt = null;

try {

this.con.setAutoCommit(false);

stmt = this.con.createStatement();

stmt.addBatch(

"INSERT INTO COFFEES " +

"VALUES('Amaretto', 49, 9.99, 0, 0)");

stmt.addBatch(

"INSERT INTO COFFEES " +

"VALUES('Hazelnut', 49, 9.99, 0, 0)");

stmt.addBatch(

"INSERT INTO COFFEES " +

"VALUES('Amaretto\_decaf', 49, " +

"10.99, 0, 0)");

stmt.addBatch(

"INSERT INTO COFFEES " +

"VALUES('Hazelnut\_decaf', 49, " +

"10.99, 0, 0)");

int [] updateCounts = stmt.executeBatch();

this.con.commit();

} catch(BatchUpdateException b) {

JDBCTutorialUtilities.printBatchUpdateException(b);

} catch(SQLException ex) {

JDBCTutorialUtilities.printSQLException(ex);

} finally {

if (stmt != null) { stmt.close(); }

this.con.setAutoCommit(true);

}

}

下面一行禁用了Connection对象的自动提交模式，这样事务不会在executeBatch方法调用的时候自动提交或者回滚。

this.con.setAutoCommit(false);

为了能够正确的错误处理，您应该始终在开始批量更新之前禁用自动提交模式。

方法Statement.addBatch将一个命令添加到与Statement对象stmt相关联的命令列表中。在这个例子中，这些命令都是INSERT INTO语句，每个都添加一行包含五个列值。COF\_NAME和PRICE列的值分别是咖啡的名称及其价格。每一行的第二个值是49，因为这是供应商Superior Coffee的识别号码。最后两个值，列SALES和TOTAL的条目都从零开始，因为还没有销售。 （SALES是本周在本周销售的这排咖啡的磅数; TOTAL是该咖啡的累计销售总额。）

以下行将添加到其命令列表的四个SQL命令发送到数据库以作为批处理执行：

int [] updateCounts = stmt.executeBatch();

请注意，stmt使用executeBatch方法发送批量插入命令，而不是使用executeUpdate方法，它只会发送一条命令并且返回一个更新行数。DBMS按照命令的顺序执行这些命令，因此它将首先为Amaretto添加一行值，然后添加Hazelnut的行，然后添加Amaretto decaf的行，最后添加Hazelnut decaf。如果所有四个命令都成功执行，DBMS将按照执行的顺序返回每个命令的更新计数。存储在updateCounts数组中的更新计数表示了每条命令执行所影响的行数。

如果批中的所有四个命令都执行成功，则updateCounts将包含四个值，所有这些值都为1，因为一条插入命令只影响一行。与stmt相关联的命令列表现在将被清空，因为当stmt调用方法executeBatch时，先前添加的四个命令都已发送到数据库。您也可以随时使用clearBatch方法清空此命令列表。

Connection.commit方法使COFFEES表的批量更新成为永久性的。此方法需要显式调用，因为此连接的自动提交模式先前已禁用。

下面一行为当前的Connection对象重新设置启用自动提交。

this.con.setAutoCommit(true);

现在，示例中的每个语句将在执行后自动提交，并且不再需要调用方法commit。

**Performing Parameterized Batch Update**

也可以进行参数化的批量更新，如下面的代码片段所示，其中con是一个Connection对象：

con.setAutoCommit(false);

PreparedStatement pstmt = con.prepareStatement(

"INSERT INTO COFFEES VALUES( " +

"?, ?, ?, ?, ?)");

pstmt.setString(1, "Amaretto");

pstmt.setInt(2, 49);

pstmt.setFloat(3, 9.99);

pstmt.setInt(4, 0);

pstmt.setInt(5, 0);

pstmt.addBatch();

pstmt.setString(1, "Hazelnut");

pstmt.setInt(2, 49);

pstmt.setFloat(3, 9.99);

pstmt.setInt(4, 0);

pstmt.setInt(5, 0);

pstmt.addBatch();

// ... and so on for each new

// type of coffee

int [] updateCounts = pstmt.executeBatch();

con.commit();

con.setAutoCommit(true);

**Handling Batch Update Exceptions**

当您调用方法executeBatch时，如果（1）添加到批处理的其中一条SQL语句生成一个结果集（通常是一个查询）或（2）批处理中的一个SQL语句由于其他一些原因不能成功执行，则会得到一个BatchUpdateException 。

您不应该将一个查询（一个SELECT语句）添加到一批SQL命令中，因为方法executeBatch返回一个更新计数的数组，它希望从执行成功的每个SQL语句中得到一个更新计数。这意味着只有返回更新计数的命令（诸如INSERT INTO，UPDATE，DELETE等命令）或返回0的命令（如CREATE TABLE，DROP TABLE，ALTER TABLE）才能以executeBatch方法成功执行。

BatchUpdateException包含与方法executeBatch返回的数组类似的更新计数数组。在这两种情况下，更新计数与生成它们的命令的顺序相同。这告诉您批处理中有多少个命令成功执行并且哪些是成功执行的命令。例如，如果五个命令成功执行，该数组将包含五个数字：第一个是第一个命令的更新计数，第二个是第二个命令的更新计数，等等。

BatchUpdateException派生自SQLException。这意味着您可以使用SQLException对象所有可用的所有方法。以下方法，JDBCTutorialUtilities.printBatchUpdateException打印所有SQLException信息以及BatchUpdateException对象中包含的更新计数。因为BatchUpdateException.getUpdateCounts返回int数组，所以代码使用for循环来打印每个更新计数：

public static void printBatchUpdateException(BatchUpdateException b) {

System.err.println("----BatchUpdateException----");

System.err.println("SQLState: " + b.getSQLState());

System.err.println("Message: " + b.getMessage());

System.err.println("Vendor: " + b.getErrorCode());

System.err.print("Update counts: ");

int [] updateCounts = b.getUpdateCounts();

for (int i = 0; i < updateCounts.length; i++) {

System.err.print(updateCounts[i] + " ");

}

}

**Inserting Rows in ResultSet Objects**

**注意**：并非所有JDBC驱动程序都支持使用ResultSet接口插入新行。如果您尝试插入新行，并且JDBC驱动程序数据库不支持此功能，则抛出SQLFeatureNotSupportedException异常。

以下方法CoffeesTable.insertRow通过ResultSet对象将一行插入到COFFEES中：

public void insertRow(String coffeeName, int supplierID,

float price, int sales, int total)

throws SQLException {

Statement stmt = null;

try {

stmt = con.createStatement(

ResultSet.TYPE\_SCROLL\_SENSITIVE

ResultSet.CONCUR\_UPDATABLE);

ResultSet uprs = stmt.executeQuery(

"SELECT \* FROM " + dbName +

".COFFEES");

uprs.moveToInsertRow();

uprs.updateString("COF\_NAME", coffeeName);

uprs.updateInt("SUP\_ID", supplierID);

uprs.updateFloat("PRICE", price);

uprs.updateInt("SALES", sales);

uprs.updateInt("TOTAL", total);

uprs.insertRow();

uprs.beforeFirst();

} catch (SQLException e ) {

JDBCTutorialUtilities.printSQLException(e);

} finally {

if (stmt != null) { stmt.close(); }

}

}

此示例使用两个参数ResultSet.TYPE\_SCROLL\_SENSITIVE和ResultSet.CONCUR\_UPDATABLE调用Connection.createStatement方法。第一个值使得ResultSet对象的光标向前和向后移动。如果要将行插入到ResultSet对象中，则需要第二个值ResultSet.CONCUR\_UPDATABLE; 它指定结果集可以更新。

在getter方法中使用字符串的相同规定也适用于updater方法。

ResultSet.moveToInsertRow方法将光标移动到插入行。插入行是与可更新结果集相关联的特殊行。它本质上是一个缓冲区，其中可以通过在将行插入结果集之前调用updater方法来构建新行。例如，该方法调用ResultSet.updateString方法将插入行的COF\_NAME列更新为Kona。

ResultSet.insertRow方法将插入行的内容插入到ResultSet对象中并进入数据库。

**注意**：在使用ResultSet.insertRow插入一行后，应将光标移动到其他行而不是插入行。例如，此示例使用ResultSet.beforeFirst方法将其移动到结果集中的第一行之前。如果应用程序的其他部分使用相同的结果集，并且光标仍然指向插入行，则可能会发生意外的结果。

**Using Prepared Statements**

This page covers the following topics:

* [Overview of Prepared Statements](http://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html#overview_ps)
* [Creating a PreparedStatement Object](http://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html#create_ps)
* [Supplying Values for PreparedStatement Parameters](http://docs.oracle.com/javase/tutorial/jdbc/basics/prepared.html#supply_values_ps)

## Overview of Prepared Statements

有时使用PreparedStatement对象发送SQL语句到数据库更加方便。这个特殊类型的语句派生自你已经知道的更通用的Statement类。

如果你想执行一个Statement对多次，使用PreparedStatement对象代替会减少执行所需的时间。

PreparedStatement对象的主要特点是它不像Statement对象在创建的时候就给定一个SQL语句。这样做的好处是，在大多数情况下，这个SQL语句立即被发送到DBMS编译。因此，PreparedStatement对象不仅包含SQL语句，还包含已预编译的SQL语句。这意味着，当PreparedStatement被执行时，DBMS可以仅仅运行PrepraredStatement的SQL语句而不用首先编译它。

尽管PreparedStatement对象可以用于没有参数的SQL语句，但您可能最常用于的还是使用参数的SQL语句。使用采用参数的SQL语句的优点是您可以使用相同的语句，但在每次执行时提供不同的值。这方面的例子如下。

以下方法，CoffeesTable.updateCoffeeSales，为每种类型的咖啡在SALES列中存储它在本周售出的磅数，并在TOTAL列中更新它的售出总磅数。

public void updateCoffeeSales(HashMap<String, Integer> salesForWeek)

throws SQLException {

PreparedStatement updateSales = null;

PreparedStatement updateTotal = null;

String updateString =

"update " + dbName + ".COFFEES " +

"set SALES = ? where COF\_NAME = ?";

String updateStatement =

"update " + dbName + ".COFFEES " +

"set TOTAL = TOTAL + ? " +

"where COF\_NAME = ?";

try {

con.setAutoCommit(false);

updateSales = con.prepareStatement(updateString);

updateTotal = con.prepareStatement(updateStatement);

for (Map.Entry<String, Integer> e : salesForWeek.entrySet()) {

updateSales.setInt(1, e.getValue().intValue());

updateSales.setString(2, e.getKey());

updateSales.executeUpdate();

updateTotal.setInt(1, e.getValue().intValue());

updateTotal.setString(2, e.getKey());

updateTotal.executeUpdate();

con.commit();

}

} catch (SQLException e ) {

JDBCTutorialUtilities.printSQLException(e);

if (con != null) {

try {

System.err.print("Transaction is being rolled back");

con.rollback();

} catch(SQLException excep) {

JDBCTutorialUtilities.printSQLException(excep);

}

}

} finally {

if (updateSales != null) {

updateSales.close();

}

if (updateTotal != null) {

updateTotal.close();

}

con.setAutoCommit(true);

}

}

**Creating a PreparedStatement Object**

以下创建一个需要两个输入参数的PreparedStatement对象：

String updateString =

"update " + dbName + ".COFFEES " +

"set SALES = ? where COF\_NAME = ?";

updateSales = con.prepareStatement(updateString);

**Supplying Values for PreparedStatement Parameters**

在执行PreparedStatement对象之前，必须提供值来替代问号占位符（如果有的话）。通过调用PreparedStatement类中定义的setter方法之一来执行此操作。以下语句在名为updateSales的PreparedStatement中提供两个问号占位符：

updateSales.setInt(1, e.getValue().intValue());

updateSales.setString(2, e.getKey());

每个这些setter方法的第一个参数指定问号占位符。在此示例中，setInt指定第一个占位符，setString指定第二个占位符。

在一个参数设置了一个值之后，它保留该值，直到它被重置为另一个值，或者调用clearParameters方法。使用PreparedStatement对象updateSales，以下代码片段说明在重置其中一个参数的值并使另一个参数相同的值之后重新使用预预编译语句：

**// changes SALES column of French Roast**

**//row to 100**

updateSales.setInt(1, 100);

updateSales.setString(2, "French\_Roast");

updateSales.executeUpdate();

**// changes SALES column of Espresso row to 100**

**// (the first parameter stayed 100, and the second**

**// parameter was reset to "Espresso")**

updateSales.setString(2, "Espresso");

updateSales.executeUpdate();

**Using Loops to Set Values**

您可以通过使用for循环或while循环来设置输入参数的值来使代码编写更容易。

CoffeesTable.updateCoffeeSales方法使用for-each循环重复设置PreparedStatement对象updateSales和updateTotal中的值：

for (Map.Entry<String, Integer> e : salesForWeek.entrySet()) {

updateSales.setInt(1, e.getValue().intValue());

updateSales.setString(2, e.getKey());

// ...

}

方法CoffeesTable.updateCoffeeSales有一个参数HashMap。HashMap参数中的每个元素都包含一种咖啡类型的名称，以及在本周内销售的咖啡类型的磅数。for-each循环遍历HashMap参数的每个元素，并在updateSales和updateTotal中设置相应的问号占位符。

**Executing PreparedStatement Objects**

与Statement对象一样，要执行PreparedStatement对象，请调用execute语句：executeQuery，如果查询只返回一个ResultSet（如SELECT SQL语句），调用executeUpdate如果查询不返回ResultSet（如UPDATE SQL语句），或者调用execute，如果查询可能返回多个ResultSet对象。CoffeesTable.updateCoffeeSales中的两个PreparedStatement对象都包含UPDATE SQL语句，因此它们都通过调用executeUpdate来执行：

updateSales.setInt(1, e.getValue().intValue());

updateSales.setString(2, e.getKey());

updateSales.executeUpdate();

updateTotal.setInt(1, e.getValue().intValue());

updateTotal.setString(2, e.getKey());

updateTotal.executeUpdate();

con.commit();

当执行updateSales和updateTotals时不会提供给executeUpdate方法参数; PreparedStatement对象都已经包含要执行的SQL语句。

**注意**：在CoffeesTable.updateCoffeeSales开头，自动提交模式设置为false：

con.setAutoCommit(false);

因此，在调用方法commit之前，不会提交任何SQL语句。有关自动提交模式的详细信息，请参阅Transactions.。

**Return Values for the executeUpdate Method**

executeQuery返回一个包含发送到DBMS的查询结果的ResultSet对象，而executeUpdate的返回值是一个int值，表示更新了的表的行数。例如，以下代码显示分配给变量n的executeUpdate的返回值：

updateSales.setInt(1, 50);

updateSales.setString(2, "Espresso");

int n = updateSales.executeUpdate();

**// n = 1 because one row had a change in it**

表COFFEES被更新了;用50替换Espresso行中列SALES中的值。该更新影响表中的一行，因此n等于1。

当方法executeUpdate用于执行DDL（数据定义语言）语句（如创建表）时，返回int值0。因此，在执行用于创建表COFFEES的DDL语句的以下代码片段中，n被赋值为0：

**// n = 0**

int n = executeUpdate(createTableCoffees);

请注意，当executeUpdate的返回值为0时，可以表示以下两种之一：

* 执行的语句是一个影响零行的更新语句。
* 执行的语句是一个DDL语句。

**Using Transactions**

有时你不希望一个语句生效，除非另外一个语句完成。例如，当“The Coffee Break”的所有者更新每周销售的咖啡量时，所有者也想要更新到目前为止销售的总金额。但是，每周销售的金额和销售总额应同时更新; 否则，数据将不一致。要想确保两个操作要么都发生要么都不发生的方法是采用事务。事务是一个或多个语句集合被作为一个单元一起执行，因此要么所有的语句都执行，要么没有一个语句被执行。

本页包含以下主题

* Disabling Auto-Commit Mode
* Committing Transactions
* Using Transactions to Preserve Data Integrity
* Setting and Rolling Back to Savepoints
* Releasing Savepoints
* When to Call Method rollback

**Disabling Auto-Commit Mode**

当一个连接创建的时候，它是处于自动提交模式。这意味着每个单独的SQL语句都被视为一个事务，并在执行后理解提交。（更准确地说，默认是在SQL语句完成时提交的，而不是执行时。一个语句在所有它的结果集和更新计数被获取时完成。然而，在几乎所有情况下，一个语句在执行完毕之后就已经完成，因此提交。）

允许将两个或多个语句分组到事务中的方法是禁用自动提交模式。这在以下代码中演示，其中con是可用连接：

con.setAutoCommit(false);

**Committing Transactions**

在禁用自动提交模式后，除非你明确调用commit方法，则不会有任何SQL语句被提交。在先调用commit方法后，所有的语句被包含在一个事务中并作为一个单元一起被提交执行。以下方法，CoffeesTable.updateCoffeeSales，其中con是可用连接，演示了一个事务：

public void updateCoffeeSales(HashMap<String, Integer> salesForWeek)

throws SQLException {

PreparedStatement updateSales = null;

PreparedStatement updateTotal = null;

String updateString =

"update " + dbName + ".COFFEES " +

"set SALES = ? where COF\_NAME = ?";

String updateStatement =

"update " + dbName + ".COFFEES " +

"set TOTAL = TOTAL + ? " +

"where COF\_NAME = ?";

try {

con.setAutoCommit(false);

updateSales = con.prepareStatement(updateString);

updateTotal = con.prepareStatement(updateStatement);

for (Map.Entry<String, Integer> e : salesForWeek.entrySet()) {

updateSales.setInt(1, e.getValue().intValue());

updateSales.setString(2, e.getKey());

updateSales.executeUpdate();

updateTotal.setInt(1, e.getValue().intValue());

updateTotal.setString(2, e.getKey());

updateTotal.executeUpdate();

con.commit();

}

} catch (SQLException e ) {

JDBCTutorialUtilities.printSQLException(e);

if (con != null) {

try {

System.err.print("Transaction is being rolled back");

con.rollback();

} catch(SQLException excep) {

JDBCTutorialUtilities.printSQLException(excep);

}

}

} finally {

if (updateSales != null) {

updateSales.close();

}

if (updateTotal != null) {

updateTotal.close();

}

con.setAutoCommit(true);

}

}

在这个方法中，连接con的自动提交模式被禁用，这意味着当调用commit方法时，两个预编译语句updateSales和updateTotal会一起提交。无论何时调用commit方法（自动启用自动提交模式或禁用自动提交模式时），所有由事务中的语句导致的更改都将是永久性的。在这个示例中，这意味着Colombian coffee的SALES和TOTAL列已更改为50（如果TOTAL以前为0），并将保持此值，直到与其他更新语句更改为止。

con.setAutoCommit（true）语句重新启用自动提交模式，这意味着再次每个语句在完成后自动提交。然后，以设置返回了默认的状态这样你不用自己去调用commit方法来提交。建议仅在事务模式下禁用自动提交模式。这样，您避免为多个语句持有数据库锁，从而增加与其他用户冲突的可能性。

**Using Transactions to Preserve Data Integrity**

除了将语句成组为一个单元执行外，事务还可以帮助保持表中数据的完整性。例如，假设一个员工应该在COFFEES表中输入新的咖啡价格，但延迟了几天。同时，价格上涨，今天的所有者正在输入一个更高的价格。在所有者试图更新表的同时，员工终于绕过输入了一个过时的价格。插入过时的价格后，员工意识到他们不再有效，并调用了Connection的rollback方法来撤销它们的操作（rollback方法中止一个事务并将值恢复到尝试更新之前）。同时，所有者正在执行SELECT语句并打印新的价格。在这种情况下，所有者可能会打印已经回滚到之前的价格的价格，使打印价格不正确。

这种情景可以通过使用事务来避免，为两个用户同时访问数据时产生的冲突提供一定程度的保护。

为了避免事务中的冲突，DBMS使用锁机制阻止他人访问事务正在访问的数据（请注意，在自动提交模式下，每个语句都是一个事务，锁只持有一个语句。）锁锁定定后，它将一直保持有效，直到事务提交或回滚。例如，DBMS可以锁定表的一行，直到它的更新被提交。此锁的作用是防止用户获得脏读，也就是在持久化之前读取了该值（访问一个还没被提交的更新值被认为是脏读，因为这个值可能被回滚到原来的值。如果你读取的值稍后被回滚，则读取的是一个无效值）。

锁的设置方式由事务隔离级别决定，隔离级别范围可以从不支持事务到支持执行非常严格的访问规则的事务。

事务隔离级别的一个示例是TRANSACTION\_READ\_COMMITTED，直到该值被提交前，它不允许该值被访问。换而言之，如果事务的隔离级别被设置为TRANSACTION\_READ\_COMMITTED，DBMS不会允许脏读发生。Connection接口包含五个值，它们表示您可以在JDBC中使用的事务隔离级别：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **隔离级别** | **事务** | **脏读** | **不可重复读** | **幻读** |
| TRANSACTION\_NONE | 不支持 | 不适用 | 不适用 | 不适用 |
| TRANSACTION\_READ\_COMMITTED | 支持 | 阻止 | 允许 | 允许 |
| TRANSACTION\_READ\_UNCOMMITTED | 支持 | 允许 | 允许 | 允许 |
| TRANSACTION\_REPEATABLE\_READ | 支持 | 阻止 | 阻止 | 允许 |
| TRANSACTION\_SERIALIZABLE | 支持 | 阻止 | 阻止 | 阻止 |

不可重复读发生在当事务A检索一行数据时，事务B随后更新该行，并且事务A之后再次检索该行。这样事务A两次检索相同的行但是看到了不同的数据。

幻读发生在当事务A检索满足给定条件的一系列行，事务B随后插入或者更新一行，这样使得该行现在满足事务A中的条件，并且事务A后号重复该条件的检索。这样事务A现在看到多了一行。这一行被称为一个幻数据。

通常，您不需要对事务隔离级别做任何事情;你可以只使用你的DBMS默认值。事务隔离级别由你的DBMS决定。比如对于Java DB，它的隔离级别是TRANSACTION\_READ\_COMMITTED。JDBC允许您查出DBMS设置的事务隔离级别（使用Connection的getTransactionIsolation方法）并且还允许您将其设置为另一个级别（使用Connection的setTransactionIsolation方法）。

**注意**：JDBC驱动程序可能不支持所有事务隔离级别。如果驱动程序不支持在调用setTransactionIsolation中指定的隔离级别，则驱动程序可以使用一个更高，更严格的事务隔离级别替代。如果驱动程序无法使用较高的事务级别进行替代，则会抛出一个SQLException。使用DatabaseMetaData.supportsTransactionIsolationLevel方法来确定驱动是否支持给定级别。

**Setting and Rolling Back to Savepoints**

Connectioin.setSavepoint方法在当前事务内设置一个Savepoint对象。Connection.rollback方法被重载来接收一个Savepoint参数。

以下方法，CoffeesTable.modifyPricesByPercentage，将特定咖啡的价格提高一个百分比priceModifier。但是，如果新价格大于指定价格，则最高价格，则价格将恢复为原始价格：

public void modifyPricesByPercentage(

String coffeeName,

float priceModifier,

float maximumPrice)

throws SQLException {

con.setAutoCommit(false);

Statement getPrice = null;

Statement updatePrice = null;

ResultSet rs = null;

String query =

"SELECT COF\_NAME, PRICE FROM COFFEES " +

"WHERE COF\_NAME = '" + coffeeName + "'";

try {

Savepoint save1 = con.setSavepoint();

getPrice = con.createStatement(

ResultSet.TYPE\_SCROLL\_INSENSITIVE,

ResultSet.CONCUR\_READ\_ONLY);

updatePrice = con.createStatement();

if (!getPrice.execute(query)) {

System.out.println(

"Could not find entry " +

"for coffee named " +

coffeeName);

} else {

rs = getPrice.getResultSet();

rs.first();

float oldPrice = rs.getFloat("PRICE");

float newPrice = oldPrice + (oldPrice \* priceModifier);

System.out.println(

"Old price of " + coffeeName +

" is " + oldPrice);

System.out.println(

"New price of " + coffeeName +

" is " + newPrice);

System.out.println(

"Performing update...");

updatePrice.executeUpdate(

"UPDATE COFFEES SET PRICE = " +

newPrice +

" WHERE COF\_NAME = '" +

coffeeName + "'");

System.out.println(

"\nCOFFEES table after " +

"update:");

CoffeesTable.viewTable(con);

if (newPrice > maximumPrice) {

System.out.println(

"\nThe new price, " +

newPrice +

", is greater than the " +

"maximum price, " +

maximumPrice +

". Rolling back the " +

"transaction...");

con.rollback(save1);

System.out.println(

"\nCOFFEES table " +

"after rollback:");

CoffeesTable.viewTable(con);

}

con.commit();

}

} catch (SQLException e) {

JDBCTutorialUtilities.printSQLException(e);

} finally {

if (getPrice != null) { getPrice.close(); }

if (updatePrice != null) {

updatePrice.close();

}

con.setAutoCommit(true);

}

}

以下语句指定当调用commit方法时，从getPrice查询生成的ResultSet对象的游标将被关闭。请注意，如果您的DBMS不支持ResultSet.CLOSE\_CURSORS\_AT\_COMMIT，则该常量将被忽略：

getPrice = con.prepareStatement(query, ResultSet.CLOSE\_CURSORS\_AT\_COMMIT);

该方法从以下语句创建一个Savepoint开始：

Savepoint save1 = con.setSavepoint();

该方法检查新价格是否大于maximumPrice值。如果是这样，该方法使用以下语句回滚事务：

con.rollback(save1);

因此，当方法通过调用Connection.commit方法提交事务时，它不会提交其关联的Savepoint已经回滚的任何行; 它将提交所有其他更新的行。

**Releasing Savepoints**

Connection.releaseSavepoint方法将Savepoint对象作为参数，并将其从当前事务中删除。

在一个保存点被释放后，尝试在回滚操作中再次引用它会导致一个SQLException被抛出。任何在事务中创建的保存点都会在事务提交，或者回滚后被自动释放而变得失效。回滚一个事务到一个保存点也会使得其他任何在这个保存点之后创建的保存点被自动释放并失效。

**When to Call Method rollback**

如前面提到的，调用rollback方法会终止一个事务，并且将任何修改的值返回到它们之前的值。如果你尝试在一个事务中执行一个或多个语句，并且获得一个SQLExcepiton，调用rollback方法来终止这个事务，并且重新开始事务。这是能够知道事务是否被提交的唯一方式。捕捉一个SQLException会告诉你有什么问题，但是它并没有告诉你哪些已经被提交。因为你不能去指望并没有提交，所以调用rollback方法是能唯一确保的方式。

方法CoffeesTable.updateCoffeeSales演示了一个事务，并且在catch块中包含了调用rollback方法。如果应用程序继续并使用事务操作的结果，在catch块中rollback方法的调用会防止这些可能不正确数据的使用。