第5章 JDBC访问数据库

本章目标

- 学会MySQL数据库的下载与安装；
- 了解JDBC的体系结构；
- 熟悉常用的JDBC API；
- 掌握使用JDBC连接数据库的步骤；
- 学会数据源的配置和使用；
- 了解和掌握DAO设计模式。

Web应用程序需要访问数据库。Java使用JDBC访问数据库，JDBC是访问数据库的标准API。本章首先介绍MySQL数据库，然后介绍使用JDBC连接数据库的方法以及常用的JDBC API，接下来介绍数据源连接数据库的方法，最后讨论DAO设计模式。

5.1 MySQL数据库

MySQL是一种开放源代码的关系型数据库管理系统(RDBMS)，目前属于Oracle旗下产品。它使用SQL语言进行数据库管理。MySQL软件采用了双授权政策，它分为社区版和商业版，由于其体积小、速度快、总体成本低，尤其是开放源码这一特点，一般中小型网站的开发都选择MySQL作为网站数据库。

MySQL数据库

5.1.1 MySQL的下载与安装

可以到Oracle官方网站下载最新的MySQL软件，MySQL提供Windows下的安装程序，本书使用的是社区版(MySQL Community Server)，下载地址如下。

```
https://www.mysql.com/downloads/
```

MySQL的最新版本是MySQL 8.0，下载文件名为mysql-installer-community-8.0.11.0.msi，双击该文件即开始安装。安装过程中需要选择安装类型和安装路径。安装结束后需要配置MySQL，指定配置类型，这里选择Development Machine，还需要打开TCP/IP网络以及指定数据库的端口号，默认值为3306。单击Next按钮，在出现的页面中需要指定root账户的密码，这里输入123456。最后还需要指定Windows服务名，这里MySQL80。

5.1.2 使用 MySQL 命令行工具

选择“开始”→“所有程序”→MySQL→MySQL Server 8.0→MySQL 8.0 Command Line Client，打开命令行窗口，输入 root 账户密码，出现 mysql >提示符，如图 5.1 所示。

在 MySQL 命令提示符下可以通过命令操作数据库，使用 show databases；命令可以显示所有数据库信息。

```
mysql > show databases;
```

在对数据库操作之前，必须使用 use 命令打开数据库，下面命令打开 world 数据库。

```
mysql > use world;
```

使用 show tables 命令可以显示当前数据库中的表。

```
mysql > show tables;
```

使用 create database 命令可以建立数据库，使用 create table 语句可完成对表的创建，使用 alter table 语句可以对创建后的表进行修改，使用 describe 命令可查看已创建的表的详细信息，使用 insert 命令可以向表中插入数据、使用 delete 命令可以删除表中的数据，使用 update 命令可以修改表中的数据，使用 select 命令可以查询表中的数据。

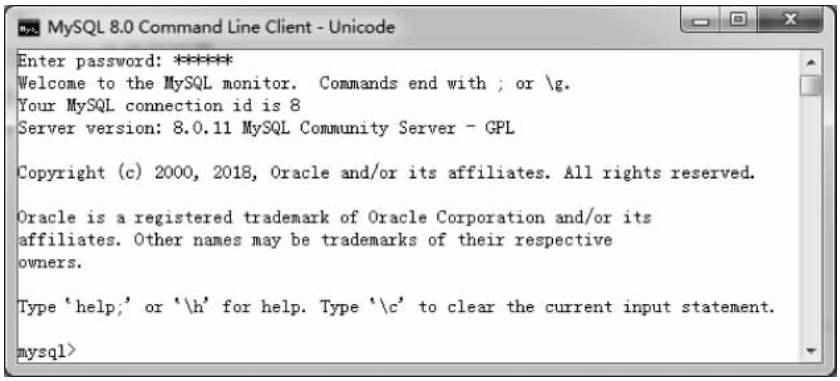

图 5.1 MySQL 命令行窗口

1. 创建数据库

创建数据库使用 create database 命令，格式如下：

```
create database <数据库名>
```

下面命令创建一个名为 webstore 的数据库。

```
mysql > create database webstore;
```

默认情况下，新建的数据库属于创建它的用户。也可以新建用户并把数据库上的操作权限授予新用户。

2. 创建用户

可以使用 CREATE USER 命令创建新用户，也可以使用 GRANT 命令在创建用户的

同时授予该用户特权。要允许用户从本地主机访问数据库,使用下面的命令:

```
mysql > grant all privileges on webstore. * to xiaozhang@localhost
      identified by '12345';
```


webstore 是数据库名,xiaozhang 是新用户名,@localhost 表示本地主机上的用户。12345 是密码。要准许用户从其他客户机访问数据库,使用下面的命令:

```
mysql > grant all privileges on database. * to xiaozhang@"%"
      identified by '12345';
```

其中@"%"的作用是通配符,表示任何客户机对数据库的访问。如果创建新用户时发生问题,请检查是否作为 root 用户启动 MySQL 服务器。

3. 使用 DDL 创建表

创建表使用 CREATE TABLE 命令,使用下面 SQL 语句创建 customers 客户表。

```
create table customers(
   id INTEGER primary key,
   name VARCHAR(20) not null,
   email VARCHAR(20),
   balance DOUBLE
);
```

4. 使用 DML 操纵表

可以使用 SQL 的 INSERT、DELETE 和 UPDATE 语句插入、删除和修改表中数据,使用 SELECT 语句查询表中数据。

使用下面语句向 customers 表中插入一行数据。

```
insert into customers
values(1001,'张大海','zhangda@163.com',35800.00);
```

使用下面语句可以查询 customers 表中所有信息。

```
select * from customers;
```

5.1.3 使用 Navicat 操作数据库

Navicat for MySQL 是一款专为 MySQL 设计的高性能数据库管理及开发工具,使用它可以简化数据库的管理及降低系统管理成本。它的设计符合数据库管理员、开发人员及中小企业的需要。Navicat 适用于 Microsoft Windows、Mac OS X 及 Linux 三种平台。它可以让用户连接到任何本地机或远程服务器、它提供一些实用的数据库工具如数据模型、数据传输、数据同步、结构同步、导入、导出、备份、还原、报表创建工具等。

Navicat for MySQL 可用于任何版本的 MySQL 数据库服务器,并支持大部分 MySQL 最新版本的功能,包括触发器、存储过程、函数、事件、视图、管理用户等。

用户可以到 http://www.formysql.com/xiazai_mysql.html 下载最新的 Navicat for MySQL 11 中文版。图 5.2 是 Navicat for MySQL 的运行界面。

图 5.2　Navcat for MySQL 的运行界面

5.2　JDBC API

JDBC 是 Java 程序访问数据库的标准，它是由一组 Java 语言编写的类和接口组成的，这些类和接口称为 JDBC API，它为 Java 程序提供一种通用的数据访问接口。

JDBC 的基本功能包括：①建立与数据库的连接。②发送 SQL 语句。③处理数据库操作结果。

5.2.1　JDBC 访问数据库

Java 应用程序访问数据库的一般过程如图 5.3 所示。应用程序通过 JDBC 驱动程序管理器加载相应的驱动程序，通过驱动程序与具体的数据库连接，然后访问数据库。

Java 应用程序要成功访问数据库，首先要加载相应的驱动程序。要使驱动程序加载成功，必须安装驱动程序。通常，JDBC 驱动程序由数据库厂商提供，可以到网上下载驱动程序，然后将驱动程序的 JAR 文件复制到 WEB-INF\lib 目录中，这样 Web 应用程序才能找到其中的驱动程序。

提示：*在 Java 8 中 JDBC-ODBC 桥驱动程序已被删除，所以不能再使用这种方法连接数据库。*

使用 JDBC API 可以访问从关系数据库到电子表格的任何数据源，它使开发人员可以用纯 Java 语言编写完整的数据库应用程序。JDBC API 已经成为 Java 语言的标准 API，在 Java 8 中的版本是 JDBC 4.2。在 JDK 中是通过 java.sql 和 javax.sql 两个包提供的。

java.sql 包提供了为基本的数据库编程服务的类和接口，如驱动程序管理的类

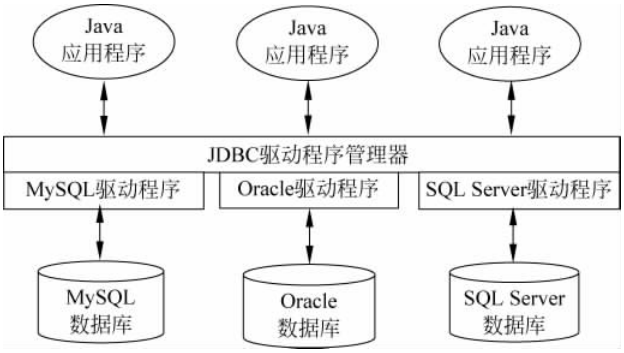

图 5.3　Java 应用程序访问数据库的过程

DriverManager、创建数据库连接 Connection 接口、执行 SQL 语句以及处理查询结果的类和接口等。

javax.sql 包主要提供了服务器端访问和处理数据源的类和接口，如 DataSource、RowSet、RowSetMetaData、PooledConnection 接口等，它们可以实现数据源管理、行集管理以及连接池管理等。

5.2.2　Connection 接口

通过调用 DriverManager 类的静态方法 getConnection()或数据源(DataSource)对象的 getConnection()都可以得到连接(Connection)对象。

1. DriverManager 类

DriverManager 类是 JDBC 的管理层，作用于应用程序和驱动程序之间。DriverManager 类跟踪可用的驱动程序，并在数据库和驱动程序之间建立连接。

DriverManager 类维护一个注册的 Driver 类的列表。建立数据库连接的方法是调用 DriverManager 类的静态方法 getConnection()，该方法的声明格式为：

- public static Connection getConnection(String dburl)
- public static Connection getConnection(String dburl,String user,String password)

这里字符串参数 dburl 表示 JDBC URL，user 表示数据库用户名，password 表示口令。调用该方法，DriverManager 类试图从注册的驱动程序中选择一个合适的驱动程序，然后建立到给定的 JDBC URL 的连接。如果不能建立连接将抛出 SQLException 异常。

2. 数据库 URL

数据库 URL 与一般的 URL 不同，它用来标识数据源，这样驱动程序就可以与它建立连接。下面是数据库 URL 的标准语法，它包括由冒号分隔的三个部分：

```
jdbc:<subprotocol>:<subname>
```

其中，jdbc 表示协议，数据库 URL 的协议总是 jdbc。subprotocol 表示子协议，它表示驱动程序或数据库连接机制的名称，子协议名通常为数据库厂商名，如 mysql、oracle、postgresql 等。subname 为子名称，它表示数据库标识符，该部分内容随数据库驱动程序的不同而不同。

5.2.3 Statement 接口

得到连接对象后就可以调用它的 createStatement()创建 SQL 语句(Statement)对象以及在连接对象上完成各种操作,下面是 Connection 接口创建 Statement 对象的方法。

- public Statement createStatement( ): 创建一个 Statement 对象。如果这个 Statement 对象用于查询,那么调用它的 executeQuery()返回的 ResultSet 是一个不可滚动、不可更新的 ResultSet。
- public Statement createStatement(int resultType, int concurrency): 创建一个 Statement 对象。如果这个 Statement 对象用于查询,那么这两个参数决定 executeQuery()返回的 ResultSet 是否是一个可滚动、可更新的 ResultSet。

一旦创建了 Statement 对象,就可以用它来向数据库发送 SQL 语句,实现对数据库的查询和更新操作等。

1. 执行查询语句

可以使用 Statement 接口的下列方法向数据库发送 SQL 查询语句。

```
public ResultSet executeQuery(String sql)
```

该方法用来执行 SQL 查询语句。参数 sql 为用字符串表示的 SQL 查询语句。查询结果以 ResultSet 对象返回,一般称为结果集对象。在 ResultSet 对象上可以逐行逐列地读取数据。

使用该方法创建的 ResultSet 对象是一个不可滚动的结果集,或者说是一个只能向前滚动的结果集,即只能从第一行向前移动直到最后一行为止,而不能向后访问结果集。

2. 执行非查询语句

可以使用 Statement 接口的下列方法向数据库发送非 SQL 查询语句。

```
public int executeUpdate(String sql)
```

该方法执行由字符串 sql 指定的 SQL 语句,该语句可以是 INSERT、DELETE、UPDATE 语句或者无返回值的 SQL 语句,如 SQL DDL 语句 CREATE TABLE。返回值是更新的行数,如果语句没有返回则返回值为 0。

- public boolean execute(String sql): 执行可能有多个结果集的 SQL 语句,sql 为任何的 SQL 语句。如果语句执行的第一个结果为 ResultSet 对象,该方法返回 true,否则返回 false。
- public int[] executeBatch(): 用于在一个操作中发送多条 SQL 语句。

3. 释放 Statement

与 Connection 对象一样,Statement 对象使用完毕应该用 close()将其关闭,释放其占用的资源。但这并不是说在执行了一条 SQL 语句后就立即释放这个 Statement 对象,可以用同一个 Statement 对象执行多个 SQL 语句。

5.2.4 ResultSet 接口

ResultSet 对象表示 SELECT 语句查询得到的记录集合,结果集一般是一个记录表,其中包含多个记录行和列标题,记录行从 1 开始,一个 Statement 对象一个时刻只能打开一个

ResultSet 对象。

如果需要对结果集的每行进行处理,需要移动结果集的游标。所谓游标(cursor)是结果集的一个标志或指针。对新产生的 ResultSet 对象,游标指向第一行的前面,可以调用 ResultSet 的 next(),使游标定位到下一条记录。next()的格式如下:

```
public boolean next() throws SQLException
```

将游标从当前位置向下移动一行。第一次调用 next()将使第一行成为当前行,以后调用游标依次向后移动。如果该方法返回 true,说明新行是有效的行,若返回 false,说明已无记录。

1. 检索字段值

ResultSet 接口提供了检索行的字段值的方法,由于结果集列的数据类型不同,所以应该使用不同的 getXxx()获得列值,例如若列值为字符型数据,可以使用下列方法检索列值:

- public String getString (int columnIndex)
- public String getString (String columnName)

返回结果集中当前行指定的列号或列名的列值,结果作为字符串返回。columnIndex 为列在结果行中的序号,序号从 1 开始,columnName 为结果行中的列名。

下面列出了返回其他数据类型的方法,这些方法都可以使用这两种形式的参数:

- public short getShort(int columnIndex):返回指定列的 short 值。
- public byte getByte(int columnIndex):返回指定列的 byte 值。
- public int getInt(int columnIndex):返回指定列的 int 值。
- public long getLong(int columnIndex):返回指定列的 long 值。
- public float getFloat(int columnIndex):返回指定列的 float 值。
- public double getDouble(int columnIndex):返回指定列的 double 值。
- public boolean getBoolean(int columnIndex):返回指定列的 boolean 值。
- public Date getDate(int columnIndex):返回指定列的 Date 对象值。
- public Object getObject(int columnIndex):返回指定列的 Object 对象值。
- public Blob getBlob(int columnIndex):返回指定列的 Blob 对象值。
- public Clob getClob(int columnIndex):返回指定列的 Clob 对象值。

2. 数据类型转换

在 ResultSet 对象中的数据为从数据库中查询出的数据,调用 ResultSet 对象的 get*Xxx*()方法返回的是 Java 语言的数据类型,因此这里就有数据类型转换的问题。实际上调用 get*Xxx*()方法就是把 SQL 数据类型转换为 Java 语言数据类型,表 5-1 列出了 SQL 数据类型与 Java 数据类型的转换。

表 5-1 SQL 数据类型与 Java 数据类型之间的对应关系

SQL 数据类型	Java 数据类型	SQL 数据类型	Java 数据类型
CHAR	String	DOUBLE	double
VARCHAR	String	NUMERIC	java. math. BigDecimal
BIT	boolean	DECIMAL	java. math. BigDecimal
TINYINT	byte	DATE	java. sql. Date

续表

SQL 数据类型	Java 数据类型	SQL 数据类型	Java 数据类型
SMALLINT	short	TIME	java. sql. Time
INTEGER	int	TIMESTAMP	java. sql. Timestamp
REAL	float	CLOB	Clob
FLOAT	double	BLOB	Blob
BIGINT	long	STRUCT	Struct

5.2.5 预处理语句 PreparedStatement

Statement 对象在每次执行 SQL 语句时都将语句传给数据库，这样在多次执行同一个语句时效率较低，这时可以使用 PreparedStatement 对象。如果数据库支持预编译，它可以将 SQL 语句传给数据库作预编译，以后每次执行这个 SQL 语句时，速度就可以提高很多。

PreparedStatement 接口继承了 Statement 接口，因此它可以使用 Statement 接口中定义的方法。PreparedStatement 对象还可以创建带参数的 SQL 语句，在 SQL 语句中指出接收哪些参数，然后进行预编译。

创建 PreparedStatement 对象与创建 Statement 对象类似，唯一不同的是需要给创建的 PreparedStatement 对象传递一个 SQL 命令，即需要将执行的 SQL 命令传递给其构造方法而不是 execute()。用 Connection 的下列方法创建 PreparedStatement 对象。

- public PreparedStatement prepareStatement(String sql)：使用给定的 SQL 命令创建一个 PreparedStatement 对象，在该对象上返回的 ResultSet 是只能向前滚动的结果集。
- public PreparedStatement prepareStatement(String sql, int resultType, int concurrency)：使用给定的 SQL 命令创建一个 PreparedStatement 对象，在该对象上返回的 ResultSet 可以通过 resultType 和 concurrency 参数指定是否可滚动、是否可更新。

这些方法的第一个参数是 SQL 字符串。这些 SQL 字符串可以包含一些参数，这些参数在 SQL 中使用问号(?)作为占位符，在 SQL 语句执行时将用实际数据替换。

PreparedStatement 对象通常用来执行动态 SQL 语句，此时需要在 SQL 语句通过问号指定参数，每个问号为一个参数。通过使用带参数的 SQL 语句可以大大提高 SQL 语句的灵活性。例如：

```
String sql = "SELECT * FROM products WHERE id = ?";
String sql = "INSERT INTO products VALUES(?, ?, ?, ?) ";
PreparedStatement pstmt = conn.prepareStatement(sql);
```

SQL 命令中的每个占位符都是通过它们的序号被引用的，从 SQL 字符串左边开始，第一个占位符的序号为 1，以此类推。当把预处理语句的 SQL 发送到数据库时，数据库将对它进行编译。

1. 设置占位符

创建 PreparedStatement 对象之后，在执行该 SQL 语句之前，必须用数据替换每个占位

符。可以通过 PreparedStatement 接口中定义的 setXxx()为占位符设置具体的值。例如,下面方法分别为占位符设置整数值和字符串值。

- public void setInt(int parameterIndex,int x):这里 parameterIndex 为参数的序号,x 为一个整数值。
- public void setString(int parameterIndex, String x):为占位符设置一个字符串值。

每个 Java 基本类型都有一个对应的 setXxx(),此外,还有许多对象类型,如 BigDecimal 有相应的 setXxx()方法。关于这些方法的详细信息请参考 Java API 文档。

对前面的 INSERT 语句,可以使用下面的方法设置占位符的值。

```
pstmt.setString(1, 105);
pstmt.setString(2,"iPhone 5 手机");
pstmt.setDouble(3, 1490.00);
pstmt.setInt(4, 5);
```

使用预处理语句还有另外一个好处,每次执行这个 SQL 命令时已经设置的值不需要再重新设置,也就是说设置的值是可保持的。另外,还可以使用预处理语句执行批量更新。

2. 用复杂数据设置占位符

使用预处理语句可以对要插入到数据库的数据进行处理。对于日期、时间和时间戳的情况,只要简单地创建相应的 java.sql.Date 或 java.sql.Time 对象,然后把它传给预处理语句对象的 setDate()或 setTime()方法即可。在 Java 8 中,java.sql 包中的 Date、Time 和 Timestamp 类都提供了一些方法,可以与 java.time 包中对应的 LocalDate、LocalTime 和 LocalDateTime 类互相进行转换。例如,在 java.sql.Date 类中定义了下面方法。

- public static Date valueOf(LocalDate date):将 LocalDate 对象转换成 java.sql.Date 对象。
- public LocalDate toLocalDate():将 java.sql.Date 对象转换成 LocalDate 对象。

下面代码将 LocalDate 对象转换成 java.sql.Date 对象并设置为预编译语句的参数。

```
LocalDate localDate = LocalDate.of(2022, Month.NOVEMBER, 20);
java.sql.Date d = java.sql.Date.valueOf(localDate);
pstmt.setDate(1, d);          //将第一个参数设置为 d
```

3. 设置空值

如果需要为某个占位符设置空值,需要使用 PreparedStatement 对象的 setNull()方法,该方法有下面两种格式:

- public void setNull(int parameterIndex, int sqlType)
- public void setNull(int parameterIndex, int sqlType, String typeName)

参数 parameterIndex 是占位符的索引,sqlType 参数是指定 SQL 类型,它的取值为 java.sql.Types 类中的常量。在 java.sql.Types 类中,每个 JDBC 类型都对应一个 int 常量,例如,如果想把 String 列设置为空,应该使用 Types.VARCHAR,这里 VARCHAR 是 SQL 的字符类型。如果要把一个 Date 列设置为空,应该使用 Types.DATE。

typeName 参数用来指定用户定义类型名或 REF 类型,用户定义类型包括 STRUCT、DISTINCT、Java 对象类型及命名数组类型等。

4. 执行预处理语句

设置好 PreparedStatement 对象的全部参数后，调用它的有关方法执行语句。对查询语句应该调用 executeQuery()，如下所示：

```
ResultSet result = pstmt.executeQuery();
```

对更新语句，应该调用 executeUpdate()，如下所示：

```
int n = pstmt.executeUpdate();
```

对其他类型的语句，应该调用 execute()，如下所示：

```
boolean b = pstmt.execute();
```

注意，对于预处理语句，必须调用这些方法的无参数版本，如 executeQuery()等。如果调用 executeQuery(String)、executeUpdate(String)或者 execute(String)，将抛出 SQLException 异常。如果在执行 SQL 语句之前没有设置好全部参数，也会抛出一个 SQLException 异常。

5.3 数据库连接步骤

下面介绍使用 JDBC API 访问数据库的基本步骤。

5.3.1 加载驱动程序

要使应用程序能够访问数据库，首先必须加载驱动程序。驱动程序是实现了 Driver 接口的类，它一般由数据库厂商提供。加载 JDBC 驱动程序最常用的方法是使用 Class 类的 forName()静态方法，该方法的声明格式为：

```
public static Class<?> forName(String className)
                        throws ClassNotFoundException
```

参数 className 为字符串表示的完整的驱动程序类名。如果找不到驱动程序将抛出 ClassNotFoundException 异常。该方法返回一个 Class 类的对象。

对不同的数据库，驱动程序的类名不同。下面几行代码分别是加载 MySQL 数据库、Oracle 数据库和 PostgreSQL 数据库驱动程序。

```
//加载 MySQL 数据库驱动程序
Class.forName("com.mysql.cj.jdbc.Driver");
//加载 Oracle 数据库驱动程序
Class.forName("oracle.jdbc.driver.OracleDriver");
//加载 PostgreSQL 数据库驱动程序
Class.forName("org.postgresql.Driver");
```

5.3.2 建立连接对象

驱动程序加载成功后应使用 DriverManager 类的 getConnection()建立数据库连接对象。下面代码建立一个到 MySQL 数据库的连接。

```
String dburl = "jdbc:mysql://127.0.0.1:3306/webstore?useSSL=true";
Connection conn = DriverManager.getConnection(
                        dburl, "root", "123456");
```


上述代码中 127.0.0.1 为本机 IP 地址,也可以使用 localhost,3306 为 MySQL 数据库服务器使用的端口号,数据库名为 webstore,用户名为 root,口令为 12345。

下面代码建立一个到 PostgreSQL 数据库的连接。

```
String dburl = "jdbc:postgresql://127.0.0.1:5432/webstore"
Connection conn = DriverManager.getConnection(
                dburl, "automan", "hacker");
```

上述代码中 5432 为数据库服务器使用的端口号,数据库名为 webstore,用户名为 automan,口令为 hacker。

表 5-2 列出了常用数据库 JDBC 连接代码。

表 5-2 常用数据库的 JDBC 连接代码

数据库	连接代码
MySQL	Class.forName("com.mysql.cj.jdbc.Driver"); Connection conn = DriverManager.getConnection("jdbc:mysql://dbServerIP:3306/dbName? user = userName&password = password");
Oracle	Class.forName("oracle.jdbc.driver.OracleDriver"); Connection conn = DriverManager.getConnection("jdbc:oracle:thin:@dbServerIP:1521:ORCL",user,password);
SQL Server	Class.forName("com.micrsoft.jdbc.sqlserver.SQLServerDriver"); Connection conn = DriverManager.getConnection("jdbc:microsoft:sqlserver://dbServerIP:1433;databaseName=master", user, password);
PostgreSQL	Class.forName("org.postgresql.Driver"); Connection conn = DriverManager.getConnection("jdbc:postgresql://dbServerIP/dbName", user, password);

表中 forName()方法中的字符串为驱动程序名,getConnection()方法中的字符串即为数据库 URL,其中 dbServerIP 为数据库服务器的主机名或 IP 地址,端口号为相应数据库的默认端口。

5.3.3 创建语句对象

通过 Connection 对象,可以创建语句(Statement)对象。对于不同的语句对象,可以使用 Connection 接口的不同方法创建。例如,要创建一个简单的 Statement 对象可以使用 createStatement(),创建 PreparedStatement 对象应该使用 prepareStatement(),创建 CallableStatement 对象应该使用 prepareCall()。下面代码创建一个简单的 Statement 对象。

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

下面代码创建一个预编译的 PreparedStatement 对象。

```
String sql = "SELECT * FROM products";
PreparedStatement pstmt = dbconn.prepareStatement(sql);
```

5.3.4 执行 SQL 语句并处理结果

执行 SQL 语句使用 Statement 对象的方法。对于查询语句，调用 executeQuery(String sql)返回 ResultSet。ResultSet 对象保存查询的结果集，再调用 ResultSet 的方法可以对查询结果的每行进行处理。

```
String sql = "SELECT * FROM products" ;
Statement stmt = conn.createStatement();
ResultSet rst = stmt.executeQuery(sql) ;
while(rst.next()){
    out.print(rst.getString(1) + "\t") ;
}
```

对于 DDL 语句如 CREATE、ALTER、DROP 和 DML 语句如 INSERT、UPDATE、DELETE 等须使用语句对象的 executeUpdate(String sql)。该方法返回值为整数，用来指示被影响的行数。

5.3.5 关闭建立的对象

在 Connection 接口、Statement 接口和 ResultSet 接口中都定义了 close()。当这些对象使用完毕后应使用 close()关闭。如果使用 Java 7 的 try-with-resources 语句，则可以自动关闭这些对象。

5.3.6 实例：Servlet 访问数据库

本示例程序可以根据用户输入的商品号从数据库中查询该商品信息，或者查询所有商品信息。本应用的设计遵循了 MVC 设计模式，其中视图有 queryProduct.jsp、displayProduct.jsp、displayAllProduct.jsp 和 error.jsp 几个页面，Product 类实现模型，ProductQueryServlet 类实现控制器。

该应用需要访问数据库表 products 中的数据，该表的定义如下：

```
CREATE TABLE products (
   id INTEGER  NOT NULL PRIMARY KEY,          -- 商品号
   pname VARCHAR(20) NOT NULL,                -- 商品名
   brand VARCHAR(20) NOT NULL,                -- 品牌
   price FLOAT,                               -- 价格
   stock SMALLINT                             -- 库存量
);
```

根据表的定义，设计下面的 Product 类存放商品信息，这里 Product 类的成员变量与表的字段对应。

程序 5.1　Product.java

```
package com.model;
import java.io.Serializable;
public class Product implements Serializable {
    private int id;
    private String pname;
    private String brand;
    private float price;
    private int stock;

    public Product() { }
    public Product(int id, String pname, String brand,
                   float price, int stock) {
      this.id = id;
      this.pname = pname;
      this.brand = brand;
      this.price = price;
      this.stock = stock;
    }
    //这里省略属性的 getter 和 setter 方法
}
```

下面是 queryProduct.jsp 页面代码。

程序 5.2　queryProduct.jsp

```
<%@ page contentType="text/html;charset=UTF-8"
                          pageEncoding="UTF-8" %>
<html>
<head><title>商品查询</title></head>
<body>
<p><a href="query-product">查询所有商品</a></p>
<form action = "query-product" method="post">
  请输入商品号:
  <input type = "text" name="productid" size="15">
  <input type = "submit" value = "确定">
</form>
</body>
</html>
```

页面运行结果如图 5.4 所示。

图 5.4　queryProduct.jsp 页面的运行结果

下面的 Servlet 连接数据库，当用户在文本框中输入商品号，单击“确定”按钮，将执行 doPost()，当用户单击“查询所有商品”链接时，将执行 doGet()。

程序 5.3　ProductQueryServlet.java

```
package com.demo;
import java.io.*;
import java.sql.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.annotation.WebServlet;
2import com.model.Product;

@WebServlet("/query-product")
public class ProductQueryServlet extends HttpServlet{
   private static final long serialVersionUID = 1L;
   Connection dbconn = null;
   public void init() {
      String driver = "com.mysql.cj.jdbc.Driver";
      String dburl = "jdbc:mysql://127.0.0.1:3306/webstore?useSSL=true";
      String username = "root";
      String password = "123456";
      try{
          Class.forName(driver); //加载驱动程序
          //创建连接对象
          dbconn = DriverManager.getConnection(
                          dburl,username,password);
      }catch(ClassNotFoundException e1){
          System.out.println(e1);
          getServletContext().log("驱动程序类找不到!");
      }catch(SQLException e2){
          System.out.println(e2);
      }
   }
   public void doGet(HttpServletRequest request,
                     HttpServletResponse response)
                     throws ServletException,IOException{
      ArrayList<Product> productList = null;
      productList = new ArrayList<Product>();
      try{
         String sql="SELECT * FROM products";
         PreparedStatement pstmt = dbconn.prepareStatement(sql);
         ResultSet result = pstmt.executeQuery();
         while(result.next()){
            Product product = new Product();
            product.setId(result.getInt("id"));
            product.setPname(result.getString("pname"));
            product.setBrand(result.getString("brand"));
            product.setPrice(result.getFloat("price"));
            product.setStock(result.getInt("stock"));
```

```
                productList.add(product);
            }
            if(!productList.isEmpty()){
                request.getSession().setAttribute("productList",productList);
                response.sendRedirect("/chapter05/displayAllProduct.jsp");
            }else{
                response.sendRedirect("/chapter05/error.jsp");
            }
        }catch(SQLException e){
            e.printStackTrace();
        }
    }

    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
                  throws ServletException,IOException{
        String productid = request.getParameter("productid");
        try{
          String sql = "SELECT * FROM products WHERE id = ?";
          PreparedStatement pstmt = dbconn.prepareStatement(sql);
          pstmt.setString(1,productid);
          ResultSet rst = pstmt.executeQuery();
          if(rst.next()){
             Product product = new Product();
             product.setId(rst.getInt("id"));
             product.setPname(rst.getString("pname"));
             product.setBrand(rst.getString("brand"));
             product.setPrice(rst.getFloat("price"));
             product.setStock(rst.getInt("stock"));
             request.getSession().setAttribute("product", product);
             response.sendRedirect("/chapter05/displayProduct.jsp");
          }else{
             response.sendRedirect("/chapter05/error.jsp");
          }
        }catch(SQLException e){
             e.printStackTrace();
        }
    }
    public void destroy(){
      try {
        dbconn.close();
      }catch(Exception e){
          e.printStackTrace();
       }
    }
}
```

程序在 init()中建立数据库连接对象,在 doPost()中根据商品号查询商品信息,并将其存储到会话对象中。在 doGet()方法中查询数据库中所有商品信息并将结果存储到 ArrayList 中并将其存储到会话对象中。destroy()关闭数据库的连接。

下面的 JSP 页面 displayProduct.jsp 和 displayAllProduct.jsp 分别显示查询一件商品和所有商品信息。

程序 5.4　displayProduct.jsp

```
<%@ page contentType = "text/html; charset = utf - 8" %>
<jsp:useBean id = "product" type = "com.model.Product"
             scope = "session"></jsp:useBean>
<html>
<head><title>商品信息</title></head>
<body>
<table border = "0">
<tr><td>商品号: </td>
    <td><jsp:getProperty name = "product" property = "id" /></td>
</tr>
<tr><td>商品名: </td>
    <td><jsp:getProperty name = "product" property = "pname" /></td>
</tr>
<tr><td>品牌: </td>
    <td><jsp:getProperty name = "product" property = "brand" /></td>
</tr>
<tr><td>价格: </td>
    <td><jsp:getProperty name = "product" property = "price" /></td>
</tr>
<tr><td>库存量: </td>
    <td><jsp:getProperty name = "product" property = "stock" /></td>
</tr>
</table>
</body></html>
```

当单击图 5.6 中的“查询所有商品”链接时，将执行 Servlet 的 doGet()方法，查询所有商品，最后控制将转到 displayAllProduct.jsp 页面，如下所示。

程序 5.5　displayAllProduct.jsp

```
<%@ page contentType = "text/html; charset = UTF - 8"
                              pageEncoding = "UTF - 8" %>
<%@ page import = "java.util. * ,com.model.Product" %>
<html>
<head><title>显示所有商品</title></head>
<body>
<table border = "1">
<tr><td>商品号</td><td>商品名</td><td>品牌</td>
        <td>价格</td><td>数量</td></tr>
<% ArrayList<Product> productList =
      (ArrayList<Product>)session.getAttribute("productList");
   for(Product product:productList){
%>
   <tr><td><% = product.getId() %></td>
       <td><% = product.getPname() %></td>
       <td><% = product.getBrand() %></td>
       <td><% = product.getPrice() %></td>
```

```
        <td><% = product.getStock() %></td></tr>
<%
   }
%>
</table>
</body></html>
```

当查询的商品不存在时,显示下面的页面。

程序 5.6　error.jsp

```
<%@ page contentType = "text/html; charset = UTF - 8" %>
<html><body>
  该商品不存在.<a href = "/chapter05/queryProduct.jsp">返回</a>
</body></html>
```

在图 5.4 的页面中输入商品号,单击“确定”按钮,则显示如图 5.5 所示的页面。在图 5.4 中单击“查询所有商品”链接,则显示如图 5.6 所示的页面。

图 5.5　显示指定商品

图 5.6　显示所有商品

5.4　使用数据源

使用数据源

在设计需要访问数据库的 Web 应用程序时,需要考虑的一个主要问题是如何管理 Web 应用程序与数据库的通信。一种方法是为每个 HTTP 请求创建一个连接对象,Servlet 建立数据库连接、执行查询、处理结果集、请求结束关闭连接。建立连接是比较耗费时间的操作,如果在客户每次请求时都要建立连接,这将导致增大请求的响应时间。此外,有些数据库支持同时连接的数量要比 Web 服务器少,这种方法限制了应用程序的可缩放性。

为了提高数据库访问效率,从 JDBC 2.0 开始提供了一种更好的方法建立数据库连接对象,即使用连接池和数据源的技术访问数据库。

5.4.1　数据源概述

数据源(DataSource)的概念是在 JDBC 2.0 中引入的,是目前 Web 应用开发中获取数据库连接的首选方法。这种方法是事先建立若干连接对象,将它们存放在数据库连接池(connection pooling)中供数据访问组件共享。使用这种技术,应用程序在启动时只需创建

少量的连接对象即可。这样就不需要为每个 HTTP 请求都创建一个连接对象，这会大大降低请求的响应时间。

数据源是通过 javax.sql.DataSource 接口对象实现的，通过它可以获得数据库连接，因此它是对 DriverManager 工具的一个替代。通常 DataSource 对象是从连接池中获得连接对象。连接池预定义了一些连接，当应用程序需要连接对象时就从连接池中取出一个，当连接对象使用完毕将其放回连接池，从而可以避免在每次请求连接时都要创建连接对象。

通过数据源获得数据库连接对象不能直接在应用程序中通过创建一个实例的方法来生成 DataSource 对象，而是需要采用 Java 命名与目录接口(Java Naming and Directory Interface，JNDI)技术来获得 DataSource 对象的引用。

可以简单地把 JNDI 理解为一种将名字和对象绑定的技术，对象工厂负责创建对象，这些对象都和唯一的名字绑定，外部程序可以通过名字来获得某个对象的访问。

在 javax.naming 包中提供了 Context 接口，该接口提供了将名字和对象绑定，通过名字检索对象的方法。可以通过该接口的一个实现类 InitialContext 获得上下文对象。

下面讨论在 Tomcat 中如何配置使用 DataSource 建立数据库连接。

5.4.2 配置数据源

在 Tomcat 中可以配置两种数据源：局部数据源和全局数据源。局部数据源只能被定义数据源的应用程序使用，全局数据源可被所有的应用程序使用。

1. 配置局部数据源

建立局部数据源非常简单，首先在 Web 应用程序的 META-INF 目录中建立一个 context.xml 文件，下面代码配置了连接 MySQL 数据库的数据源，内容如下。

程序 5.7 context.xml

```
<?xml version = "1.0" encoding = "utf - 8"?>
< Context reloadable = "true">
< Resource
    name = "jdbc/webstoreDS"
    type = "javax.sql.DataSource"
    maxTotal = "4"
    maxIdle = "2"
    driverClassName = "com.mysql.cj.jdbc.Driver"
    url = "jdbc:mysql://127.0.0.1:3306/webstore?useSSL = true"
    username = "root"
    password = "12345"
    maxWaitMillis = "5000" />
</Context >
```

上述代码中< Resource >元素各属性的含义如下。

- name：数据源名，这里是 jdbc/webstoreDS。
- driverClassName：使用的 JDBC 驱动程序的完整类名。
- url：传递给 JDBC 驱动程序的数据库 URL。
- username：数据库用户名。
- password：数据库用户口令。

- type：指定该资源的类型，这里为 DataSource 类型。
- maxTotal：指定数据源最多连接数。
- maxIdle：连接池中可空闲的连接数。
- maxWaitMillis：在没有可用连接的情况下，连接池在抛出异常前等待的最大毫秒数。

通过上面的设置后，不用在 Web 应用程序的 web.xml 文件中声明资源的引用就可以直接使用局部数据源。

2. 配置全局数据源

全局数据源可被所有应用程序使用，它是通过< *tomcat-install* >/conf/server.xml 文件的< GlobalNamingResources >元素定义的，定义后就可在任何的应用程序中使用。假设要配置一个名为 jdbc/webstoreDS 的数据源，应该按下列步骤操作。

(1) 首先在 server.xml 文件的< GlobalNamingResources >元素内增加下面的代码。

```
< Resource
       name = "jdbc/webstoreDS"
       type = "javax.sql.DataSource"
       maxTotal = "4"
       maxIdle = "2"
       username = "root"
       maxWaitMillis = "5000"
       driverClassName = "com.mysql.cj.jdbc.Driver"
       password = "12345"
       url = "jdbc:mysql://127.0.0.1:3306/webstore?useSSL = true "
/>
```

这里的 name 属性值是指全局数据源名称，其他属性与局部数据源属性含义相同。

(2) 在 Web 应用程序中建立一个 META-INF 目录，在其中建立一个 context.xml 文件，内容如下。

程序 5.8　context.xml

```
<?xml version = "1.0" encoding = "utf - 8"?>
< Context reloadable = "true">
  < ResourceLink
    global = "jdbc/webstoreDS"
    name = "jdbc/sampleDS"
    type = "javax.sql.DataSource"/>
< WatchedResource > WEB - INF/web.xml </WatchedResource >
</Context >
```

上述文件中< ResourceLink >元素用来创建到全局 JNDI 资源的链接，该元素有三个属性。

- global：指定在全局 JNDI 环境中所定义的全局资源名。
- name：指定数据源名，该名相对于 java:comp/env 命名空间前缀。
- type：指定该资源的类型的完整类名。

配置了全局数据源后，需重新启动 Tomcat 服务器才能生效。使用全局数据源访问数

据库与局部数据源相同。

5.4.3 在应用程序中使用数据源

配置了数据源后，就可以使用 javax.naming.Context 接口的 lookup()查找 JNDI 数据源，如下面代码可以获得 jdbc/webstoreDS 数据源的引用。

```
Context context = new InitialContext();
DataSource dataSource =
      (DataSource)context.lookup("java:comp/env/jdbc/webstoreDS");
```

查找数据源对象的 lookup()的参数是数据源名字符串，但要加上“java:comp/env”前缀，它是 JNDI 命名空间的一部分。得到了 DataSource 对象的引用后，就可以通过它的 getConnection()获得数据库连接对象 Connection。

对程序 5.3 的数据库连接程序，如果使用数据源获得数据库连接对象，修改后的程序如下。

程序 5.9　ProductQueryServlet.java

```
package com.demo;
import java.io.*;
import java.sql.*;
import javax.sql.DataSource;
import javax.servlet.*;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.*;
import com.model.Product;
import java.util.*;
import javax.naming.*;

@WebServlet("/query-product")
public class ProductQueryServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;
    Connection dbconn = null;
    DataSource dataSource;              //声明一个数据源变量

    public void init() {
        try {
            Context context = new InitialContext();
            //查找数据源
            dataSource =
                (DataSource)context.lookup("java:comp/env/jdbc/webstoreDS");
            //通过数据源返回连接对象，即从连接池中取出一个连接
            dbconn = dataSource.getConnection();
        }catch(NamingException ne){
            System.out.println("Exception:" + ne);
        }catch(SQLException se){
            System.out.println("Exception:" + se);
        }
    }
```

```
    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
                       throws ServletException, IOException {
        //代码同程序 5.3 的 doPost()
    }
    public void doGet(HttpServletRequest request,
                       HttpServletResponse response)
                       throws ServletException, IOException {
        //代码同程序 5.3 的 doGet()
    }
}
```

代码首先通过 InitialContext 类创建一个上下文对象 context,然后通过它的 lookup()查找数据源对象,最后通过数据源对象从连接池中返回一个数据库连接对象。当程序结束数据库访问后,应该调用 Connection 的 close()将连接对象返回到数据库连接池。这样,就避免了每次使用数据库连接对象都要重新创建,从而可以提高应用程序的效率。

5.5 DAO 设计模式

DAO(Data Access Object)称为数据访问对象。DAO 设计模式可以在使用数据库的应用程序中实现业务逻辑和数据访问逻辑分离,从而使应用的维护变得简单。它通过将数据访问实现(通常使用 JDBC 技术)封装在 DAO 类中,提高应用程序的灵活性。

DAO 模式有很多变体,这里介绍一种比较简单的形式。首先定义一个 DAO 接口,它负责建立数据库连接。然后为每种实体的持久化操作定义一个接口,如 ProductDao 接口负责 Product 对象的持久化,CustomerDao 接口负责 Customer 对象的持久化,最后定义这些接口的实现类。图 5.7 给出 Dao 接口、ProductDao 接口和 CustomerDao 接口的关系。

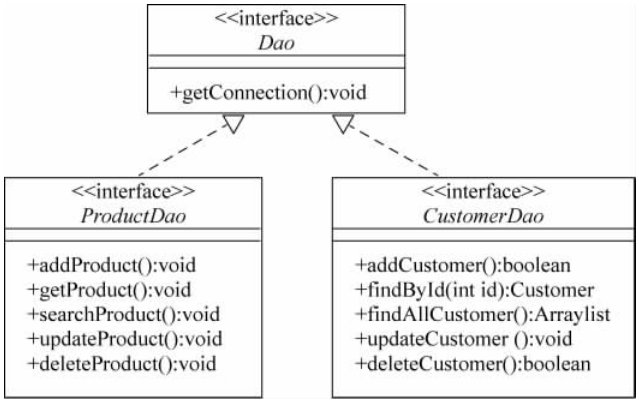

图 5.7 Dao 接口及其子接口

在 DAO 模式中,通常要为需要持久存储的每种实体类型编写一个相应的类。如要存储 Customer 信息就需要编写一个类。实现类应该提供以下功能:添加、删除、修改、检索、查找等功能。

5.5.1 设计实体类

实体类是用来存储要与数据库交互的数据。实体类通常不包含任何业务逻辑，业务逻辑由业务对象实现，因此实体类有时也叫普通的 Java 对象（Plain Old Java Object，POJO）。实体类必须是可序列化的，也就是它必须实现 java.io.Serializable 接口，下面的 Customer 类就是实体类。

程序 5.10　Customer.java

```
package com.model;
import java.io.Serializable;
public class Customer implements Serializable{
   private int id;
   private String name;
   private String email;
   private double balance;
   public int getId() {
      return id;
   }
   public void setId(int id) {
      this.id = id;
   }
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
   public String getEmail() {
      return email;
   }
   public void setEmail(String email) {
      this.email = email;
   }
   public double getBalance() {
      return balance;
   }
   public void setBalance(double balance) {
      this.balance = balance;
   }
}
```

该持久化对象用于在程序中保存应用数据，并可实现对象与关系数据的映射，它实际上是一个可序列化的 JavaBeans。

5.5.2 设计 DAO 对象

本示例的数据访问对象组件包含下面的接口和类：

- Dao 接口是所有接口的根接口，其中定义了默认方法建立到数据库的连接。

- DaoException 类是一个异常类,当 Dao 方法发生运行时异常时抛出。
- CustomerDao 接口和 CustomerDaoImpl 实现类提供了对 Customer 对象持久化的各种方法。

异常类 DaoException 如程序 5.11 所示,Dao 接口如程序 5.12 所示,CustomerDao 接口如程序 5.13 所示,CustomerDaoImpl 类如程序 5.14 所示。

程序 5.11　DaoException.java

```
package com.dao;
public class DaoException extends Exception{
   private static final long serialVersionUID = 19192L;
   private String message;
   public DaoException() {}
   public DaoException(String message){
      this.message = message;
   }
   public String getMessage(){
      return message;
   }
   public void setMessage(String message) {
      this.message = message;
   }
   public String toString(){
      return message;
   }
}
```

程序 5.12　Dao.java

```
package com.dao;
import java.sql.*;
import javax.sql.DataSource;
import javax.naming.*;

public interface Dao {
    //查找并返回数据源对象
    public static DataSource getDataSource(){
        DataSource dataSource = null;
        try {
            Context context = new InitialContext();
            dataSource =
                (DataSource)context.lookup("java:comp/env/jdbc/webstoreDS");
        }catch(NamingException ne){
            System.out.println("异常:" + ne);
        }
        return dataSource;
    }
    //返回连接对象方法
    public default Connection getConnection() throws DaoException {
        DataSource dataSource = getDataSource();
        Connection conn = null;
```

```
        try{
            conn = dataSource.getConnection();
        }catch(SQLException sqle){
            System.out.println("异常:" + sqle);
        }
        return conn;
    }
}
```

该接口的 getDataSource()静态方法用于查找并返回数据源对象,getConnection()方法是接口的默认方法,它通过一个数据源对象创建并返回数据库连接对象,该方法将被子接口或实现类继承。

程序 5.13　CustomerDao.java

```
package com.dao;
import java.util.ArrayList;
import com.model.Customer;
public interface CustomerDao extends Dao{
    //添加客户方法
    public boolean addCustomer (Customer customer) throws DaoException;
    //按 id 查询客户方法
    public Customer findById (int id) throws DaoException;
    //查询所有客户方法
    public ArrayList<Customer> findAllCustomer ()throws DaoException;
}
```

为了简单,该接口只定义了三个对 Customer 的操作方法。addCustomer()用来插入一个客户记录、findById()用来查询一个客户,findAllCustomer()返回所有客户信息。

程序 5.14　CustomerDaoImpl.java

```
package com.dao;
import java.sql.*;
import java.util.ArrayList;
import com.model.Customer;
public class CustomerDaoImpl implements CustomerDao{
    //插入一条客户记录
    public boolean addCustomer(Customer customer)
            throws DaoException{
      String sql = "INSERT INTO customers VALUES(?,?,?,?)";
      try(
          Connection conn = getConnection();
          PreparedStatement pstmt = conn.prepareStatement(sql))
      {
        pstmt.setInt(1,customer.getId());
        pstmt.setString(2,customer.getName());
        pstmt.setString(3,customer.getEmail());
        pstmt.setDouble(4,customer.getBalance());
        pstmt.executeUpdate();
        return true;
      }catch(SQLException se){
```

```
            se.printStackTrace();
            return false;
        }
    }
    //按 id 查询客户记录
    public Customer findById(int id) throws DaoException{
        String sql = "SELECT id,name,email,balance" +
                  " FROM customers WHERE id = ?";
        Customer customer = new Customer();
        try(
          Connection conn = getConnection();
          PreparedStatement pstmt = conn.prepareStatement(sql)){
          pstmt.setInt(1,id);
          try(ResultSet rst = pstmt.executeQuery()){
             if(rst.next()){
              customer.setId(rst.getInt("id"));
              customer.setName(rst.getString("name"));
              customer.setEmail(rst.getString("email"));
              customer.setBalance(rst.getDouble("balance"));
            }
          }
        }catch(SQLException se){
          return null;
         }
         return customer;
    }
    //查询所有客户信息
    public ArrayList<Customer> findAllCustomer()throws DaoException{
        Customer customer = new Customer();
        ArrayList<Customer> custList = new ArrayList<Customer>();
        String sql = "SELECT * FROM customers";
        try(
            Connection conn = getConnection();
            PreparedStatement pstmt = conn.prepareStatement(sql);
            ResultSet rst = pstmt.executeQuery()){
            while(rst.next()){
              customer.setId(rst.getInt("id"));
              customer.setName(rst.getString("name"));
              customer.setEmail(rst.getString("email"));
              customer.setBalance(rst.getDouble("balance"));
              custList.add(customer);
        }
          return custList;
      }catch(SQLException e){
          e.printStackTrace();
          return null;
      }
    }
}
```

该类没有给出修改记录和删除记录的方法,读者可自行补充完整。

5.5.3 使用 DAO 对象

下面的 addCustomer.jsp 页面通过一个表单提供向数据库中插入的数据。

程序 5.15 addCustomer.jsp

```
<%@ page contentType="text/html; charset=UTF-8" %>
<html><head><title>添加客户</title></head>
<body>
<font color=red><%=request.result%></font>
<p>请输入一条客户记录</p>
<form action = "addCustomer.do" method = "post">
  <table>
    <tr><td>客户号:</td><td><input type="text" name="id"></td></tr>
    <tr><td>客户名:</td><td><input type="text" name="cname"></td></tr>
    <tr><td>Email:</td><td><input type="text" name="email"></td></tr>
    <tr><td>余额:</td><td><input type="text" name="balance"></td></tr>
    <tr><td><input type="submit" value="确定"></td>
      <td><input type="reset" value="重置"></td>
  </tr>
</table>
</form>
</body></html>
```

下面的 AddCustomerServlet 使用了 DAO 对象和持久化对象,通过 JDBC API 实现将数据插入到数据库中。

程序 5.16 AddCustomerServlet.java

```
package com.demo;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import com.model.Customer;
import com.dao.CustomerDao;
import com.dao.CustomerDaoImpl;
import javax.servlet.annotation.WebServlet;

@WebServlet("/addCustomer.do")
public class AddCustomerServlet extends HttpServlet{
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
                   throws ServletException,IOException{
    CustomerDao dao = new CustomerDaoImpl();
    Customer customer = new Customer();
    String message = null;
    try{
        customer.setId(Integer.parseInt(request.getParameter("id")));
        //将传递来的字符串重新使用 utf-8 编码,以免产生乱码
        customer.setName(new String(request.getParameter("cname")
```

```
                .getBytes("iso-8859-1"),"UTF-8"));
        customer.setEmail(new String(request.getParameter("email")
                .getBytes("iso-8859-1"),"UTF-8"));
        customer.setBalance(
            Double.parseDouble(request.getParameter("balance")));
        boolean success = dao.addCustomer(customer);
        if(success){
            message = "<li>成功插入一条记录!</li>";
        }else{
          message = "<li>插入记录错误!</li>";
        }
      }catch(Exception e){
        System.out.println(e);
        message = "<li>插入记录错误!</li>" + e;
      }
      request.setAttribute("result", message);
      RequestDispatcher rd =
        getServletContext().getRequestDispatcher("/addCustomer.jsp");
      rd.forward (request,response);
    }
}
```

该程序首先从请求对象中获得请求参数并进行编码转换,创建一个 Customer 对象,然后调用 CustomerDao 对象的 addCustomer()方法将客户对象插入数据库中,最后根据该方法执行结果将请求再转发到 JSP 页面。

访问 addCustomer.jsp 页面,输入客户信息,单击“确定”按钮可将客户信息插入数据库,如图 5.8 所示。

图 5.8　addCustomer.jsp 的运行结果

本 章 小 结

Java 程序是通过 JDBC API 访问数据库的。JDBC API 定义了 Java 程序访问数据库的接口。访问数据库首先应该建立到数据库的连接。传统的方法是通过 DriverManager 类的 getConnection()建立连接对象。使用这种方法很容易产生性能问题。因此,从 JDBC 2.0 开始提供了通过数据源建立连接对象的机制。

通过 PreparedStatement 对象可以创建预处理语句对象，它可以执行动态 SQL 语句。通过数据源连接数据库，首先需要建立数据源，然后通过 JNDI 查找数据源对象，建立连接对象，最后通过 JDBC API 操作数据库。

DAO 设计模式是数据库访问的标准方法，它是一种面向接口的设计方法，实现数据访问逻辑，通常为每个实体类设计一个接口和一个实现类。

思考与练习

1. Web 应用程序需要访问数据库，数据库驱动程序应该安装在哪个目录中？（　　）

A. 文档根目录　　B. WEB-INF\lib

C. WEB-INF　　D. WEB-INF\classes

2. 使用 Class 类的 forName()加载驱动程序需要捕获什么异常？（　　）

A. SQLException　　B. IOException

C. ClassNotFoundException　　D. DBException

3. 程序若要连接 Oracle 数据库，请给出连接代码。数据库驱动程序名是什么？数据库 JDBC URL 串的内容是什么？

4. 试说明使用数据源对象连接数据库的优点是什么？通过数据源对象如何获得连接对象？

5. 编写一个 Servlet，查询 books 表中所有图书的信息并在浏览器中通过表格的形式显示出来。

6. 请为本章的 CustomerDao.java 程序增加两个方法实现删除和修改客户信息，这两个方法的格式为：

```
public boolean deleteCustomer(String custName)
public boolean updateCustomer(Customer customer)
```

7. 编写一个名为 SelectCustomerServlet 的 Servlet，在其中使用 CustomerDao 类的 findById()，实现客户查询功能，然后将请求转发到 displayCustomer.jsp 页面，显示查询结果。

8. 改写 4.7 节购物车应用，使用数据库存放商品信息，实现商品的显示、删除等操作。

9. C3P0 是一个开源的 JDBC 连接池，它实现了数据源和 JNDI 绑定，支持 JDBC 4 规范的标准扩展。目前使用它的开源项目有 Hibernate，Spring 等。在 Java Web 应用中可以使用它来建立数据源而不需要使用 JNDI。可到网上下载 C3P0 或从 Hibernate 的打包文件中获得，最新版本是 0.9.5.2。将 c3p0-0.9.5.2.jar 和 mchange-commons-java-0.2.11.jar 两个文件复制到 WEB-INF\lib 目录中。请改写本章 5.9 程序使用 C3P0 创建数据源对象。

10. 开发一个如图 5.9 所示的应用程序，其功能是插入和删除数据库表中学生记录。当插入一条学生记录后，程序应显示表中所有记录。要求数据库连接使用数据源，数据库操作通过 DAO 设计模式实现。

图 5.9　添加学生记录页面