# 第3章 数据库编程

## 3.1 概述

业界存在许多数据库，且每种数据库所使用的协议和底层机制也各不相同。尽管从一开始 Java 开发人员就意识到 Java 在数据库应用方面潜力巨大，想通过扩展 Java 的标准类库就可以使用“纯”Java 语言与任何数据库进行通信，但这显然是一个无法完成的任务。所以，数据库供应商和开发商都认为，如果 Java 能够提供一套“纯”Java API，同时提供一个驱动管理器来允许第三方驱动程序可以连接到特定的数据库，数据库供应商就可以提供自己的驱动程序来插入注册到驱动管理器中。针对此，Java 提供了 JDBC(Java database connectivity)，用以与数据库连接。JDBC 是一个规范，提供了一整套接口，允许以一种可移植的方式访问底层数据库。

JDBC 通过标准的 SQL 语句，甚至是专门的 SQL 扩展访问数据库。数据库供应商和数据库工具提供商可以提供底层的驱动程序。本章将介绍具有广泛应用的关系型数据库，以及通过 JDBC 访问、操作数据库的方法。

## 3.2 数据库基础

日常生产、生活产生了海量数据，纸质存储已经远远不能满足要求。随着信息时代到来，计算机提供了另一种存储和处理数据的方法。那么计算机是如何存储与管理这些数据的呢？人们在计算机上建立了一套类似于现实生活中的仓库的数据仓库，把数据有组织地存储在数据仓库中，并提供相应的管理软件来管理这些数据，这个仓库就称为数据库。其中最重要的一种数据库被称为关系数据库。


扫一扫

## 3.2.1 关系数据库

关系数据库(relational database,RDB)是基于集合代数发展而来的数据种类,采用关系模型来组织数据。关系模型可以简单理解为二维表格模型,而关系型数据库就是由二维表及其之间的关系组成的一个数据组织。二维表格模型又称为关系。二维表格由列头和多行数据组成,列头定义了关系由哪些属性组成(这些属性称为字段),每行数据包含一个唯一的数据实体(称为记录),每个数据实体的每个字段(属性)有确定的值。

用来管理关系数据的软件被称为关系数据库管理系统(relational database management system,RDBMS),常见的 RDBMS 有 Oracle、IBM DB2、SQL Server、MySQL、Microsoft Access。MySQL 是一种开放源码,它是免费的关系数据库管理系统,具有体积小、速度快、总体拥有成本低等特点,广泛地被使用(注意:MySQL 的社区版免费,但商业版是收费的)。

关系数据库有以下常用概念。

(1) 表。

表(table)是以行和列表示的数据的集合,每个表在数据库中都有一个名称。一个数据库可以包含任意多个数据表。表又被称为关系,这就是关系数据库名称的来源。表中的行被称为记录,或者元组;表中的列被称为字段,或者属性。

(2) 记录或元组。

表中的每一行称为记录(record),也称为元组,它是表中存在的每个单独数据项。

(3) 列。

列(column)是表中的垂直实体,其中包含与表中特定字段关联的所有信息。

(4) 字段。

每个表都被分解为更小的实体,称为字段。字段是表中的列名称,用于维护有关表中每条记录的特定信息。

(5) 域。

表中属性的一组允许值。属性不能接收域外的值,例如一个整型字段不能接收浮点数类型或者字符类型的值。

(6) 空值。

表中的空值(null)是字段中显示为空的值,表示在创建记录时留空的字段,非常重要的一点是空值不同于零值或包含空格的字。

(7) 索引。

使用索引可快速访问数据库表中的特定信息。索引是对数据库表中一列或多列的值进行排序的一种结构。类似书籍的目录。

(8) 键。

键在 RDBMS 中起着重要作用,它可以在表中标识唯一行,还可以建立表之间的关系。以下列举常用的键。

① 主键(primary key):表中的一列或一组列,用于唯一标识该表中的元组(行)。其中主键包含多列,又称为超键,主键包含一列,又称为候选键。

② 外键(foreign key):外键是表的列指向另一个表的主键。它们充当表之间的交叉

引用。

比如，需要存储和处理学生以及课程信息。在关系数据库中，为学生和课程各建立一张表，表中的每一条记录代表一个学生或课程的具体信息，如表 3-1 和表 3-2 所示。

表 3-1　学生(student)表

| 学号(id) | 姓名(name) | 年龄(age) |
|---|---|---|
| 1001 | 张三 | 18 |
| 1002 | 李四 | 17 |

表 3-2　课程(course)表

| 课程编号(no) | 课程名称(name) | 学时(hours) |
|---|---|---|
| 001 | 数学 | 32 |
| 002 | 哲学 | 48 |

课程表中有两条记录，分别是数学课程信息和哲学课程信息，学生表中也有两条记录，分别是张三和李四的基本信息。现在要表示学生的选课信息，在关系数据库中如何表示呢？一般的做法是新建一张表，这张表用于描述学生表和课程表的关系，这张表称为关系表，关系表中的每条记录都要关联着关系的双方。比如在这个例子中，需要建立一张学生成绩表，用于表示学生的选课信息，表的结构如表 3-3 所示。

表 3-3　学生成绩(student_grade)表

| 学号(student_id) | 课程编号(course_no) | 成绩(grade) |
|---|---|---|
| 1001 | 001 | 80 |
| 1001 | 002 | 70 |
| 1002 | 002 | 90 |

表 3-3 中的每行记录有三个字段，分别用于表示学生的学号、课程编号以及学生的该门课成绩。其中学号即为外键，与学生表产生关联。通过这种关联，学生与课程之间就建立了关系。例如此例中，学号为 1001 的学生选择了编号为 001 和 002 的课程，学号为 1002 的学生只选择了编号为 002 的课程。

关系数据因简单灵活、安全健壮、高效率等特性得到了广泛应用，当今的主流数据库厂商生产的数据库仍然是以关系数据库为主，比如 MySQL、Oracle、DB2 等。

## 3.2.2　结构化查询语言

为了满足复杂的数据库操作，人们为关系数据库专门设计了一种数据处理语言，即结构化查询语言 SQL(structured query language)。最早提出结构化查询语言的是 IBM 公司，随后该语言得到广泛应用，并发布了相应的标准。现如今，虽然各种关系数据库的查询语言有一些差异，但大多数都遵从了 ANSI SQL 标准。

SQL 语言分成以下 4 部分。

（1）数据定义语言（DDL）：create、drop、alter 等语句。

（2）数据操作语言（DML）：insert、update、delete 等语句。

（3）数据查询语言（DQL）：select 等语句。

（4）数据控制语言（DCL）：grant、revoke、commit、rollback 等语句。

表 3-4 所示为 SQL 语言使用方法。

表 3-4　SQL 语言使用方法

| 操　作 | SQL 语 句 |
| --- | --- |
| 创建表 | create table student(id int,name varchar(50),age int);<br>create table course(no VARCHAR(50),name VARCHAR(50),hours int)<br>create table student_grade (student_id int,course_no VARCHAR(50),grade double) |
| 删除记录 | delete from student where id=1001 |
| 插入记录 | insert into student(id,name,age) values(100,'张三',18); |
| 修改记录 | update student set name='王五',age=24 where id=1001 |
| 查询数据 | select id,name,age from student;<br>select id,name,age from student where id=1001;<br>Select s.NAME,c.NAME,sg.grade |

以上列出了一些简单的 SQL 操作，SQL 是一种简单但功能强大的语言，有兴趣的同学请阅读 SQL 方面的专业书籍。

注意：在 Java 程序设计中，数据库 SQL 执行相对于 Java 来说非常耗费时间，因此不要滥用 SQL。可以通过关系数据的索引来提高 SQL 的执行效率。

### 3.2.3　MySQL 数据库

MySQL 是当下最流行的关系型数据库管理系统之一，在 Web 应用方面是最好的 RDBMS 应用软件之一。它采用双授权政策，分为社区版和商业版，由于其体积小、速度快、总体拥有成本低，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。随着 MySQL 的不断成熟，它也逐渐用于更多大规模网站和应用，比如维基百科、Google 和 Facebook 等网站。它的典型特性如下。

（1）使用 SQL 语言作为访问数据库的语言，优化的 SQL 查询算法有效提高了查询速度。

（2）为多种语言提供 API。这些编程语言包括 C、C ++ 、C ♯ 、VB.NET、Java、Perl、PHP、Python、Ruby 和 Tcl 等。

（3）支持多线程，充分利用 CPU 资源支持多用户。可以处理拥有上千万条记录的大型数据库。

（4）既能够作为一个单独的应用程序在客户—服务器网络环境中运行，也能够作为一个程序库而嵌入到其他的软件中。同时提供了用于管理、检查、优化数据库操作的管理工具。

（5）提供 TCP/IP、ODBC 和 JDBC 等多种数据库连接途径。

# 3.3 JDBC

本书使用的数据库是 MySQL 数据库，因此需要先安装 MySQL 数据库，下载 MySQL 数据库的驱动包，并把驱动包导入到项目中。

程序在运行的过程中会产生和处理大量的数据，因此数据库成为了多数程序必不可少的一部分。那么 Java 是如何对数据库进行操作的呢？这就是本节将要讨论的内容——JDBC。

## 3.3.1 数据库驱动


扫一扫

JDBC 是 Java 设计者为数据库编程提供的一组接口。如图 3-1 所示，对于开发者来说，这组接口是访问数据库的工具；对于数据库提供商来说，是驱动程序的编写规范，从而保证 Java 可以访问他们的数据库产品。因此使用 JDBC 后，开发者可以更加专注于业务开发，而不必为特定的数据库编写程序。

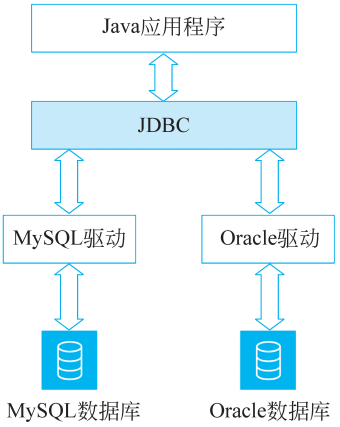

图 3-1 JDBC 驱动

JDBC 面向的是两个方向：开发者和数据库提供商。对于开发者来说，只要使用数据库提供商提供的驱动程序，就可以方便地访问数据库了；对于数据库提供商来说，他们的职责就是根据 JDBC 规范编写正确的驱动程序。那么 JDBC 如何让数据库操作运转起来呢？首先来看看 JDBC 的架构，如图 3-2 所示。

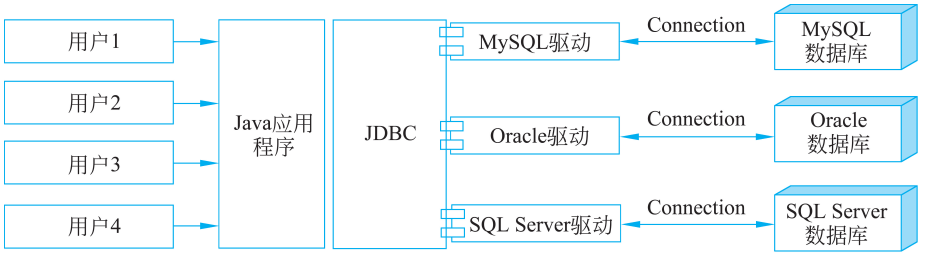

图 3-2 JDBC 架构

从图 3-2 可以看出，使用 JDBC 的用户很多，访问的数据库也各不相同，但 JDBC 使用这种可插拔的方式，让用户以同一种方式访问数据库。图 3-2 中只列出了 3 种数据库的驱动，其他的数据库厂商都可以根据 JDBC 规范来编写自己的驱动程序，从而让 Java 开发者可以访问他们的数据库产品。

上面提到了可插拔的方式，那么 Java 是如何实现驱动程序的可插拔呢？让我们先来看一下 JDBC 的生命周期，图 3-3 显示了 JDBC 的生命周期。

第 1 步：注册驱动(作用：告诉 Java 程序即将要连接的是哪个品牌的数据库)。

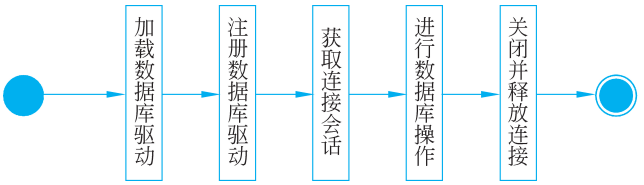


图 3-3　JDBC 的生命周期

第 2 步：获取连接(表示 JVM 的进程和数据库进程之间的通道打开了，这属于进程之间的通信，重量级的，使用完之后一定要关闭通道)。

第 3 步：获取数据库操作对象(专门执行 SQL 语句的对象)。

第 4 步：执行 SQL 语句(DQL DML…)。

第 5 步：处理查询结果集(只有当第 4 步执行的是 select 语句的时候，才有第 5 步处理查询结果集。)

第 6 步：释放资源(使用完资源之后一定要关闭资源。Java 和数据库属于进程之间的通信，开启之后一定要关闭)。

在 JDBC 的生命周期中，首先要加载需要用到的数据驱动，然后将加载的驱动注册到 JDBC 中，然后用户就可通过 JDBC 获取数据库连接会话了。获取会话后，用户就可以使用该会话进行数据库操作，操作完成后，即可关闭释放连接，一个 JDBC 的使用周期结束。

在 JDBC 的生命周期中，需要用到几个重要的类或接口，比如 Connection、Driver、DriverManager 以及一些具体的驱动类。下面来看看这些类或接口之间的关系，图 3-4 显示了 JDBC 核心类的架构。

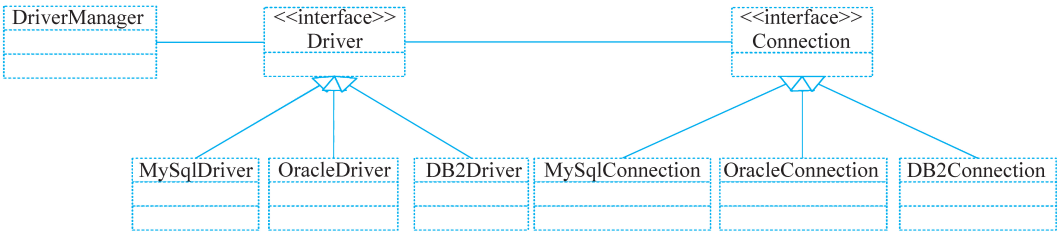


图 3-4　JDBC 核心类的架构

从图 3-4 可以看出，JDBC 使用了接口 Driver 和接口 Connection，JDBC 设计者并未对其进行实现，具体的实现留给数据库提供商，正是这种面向接口的编程使得 JDBC 的扩展更加灵活健壮，可以对 JDBC 进行插拔操作。

## 3.3.2　JDBC 核心组件

常见的 JDBC 组件 API 提供以下接口和类。

(1) DriverManager：此类管理数据库驱动程序。使用通信子协议将来自 Java 应用程序的连接请求与适当的数据库驱动程序进行匹配。在 JDBC 下识别某个子协议的第一个驱动程序将用于建立数据库连接。

(2) Driver：此接口处理与数据库服务器的通信。很少会直接与 Driver 对象进行交互。

但会使用 DriverManager 对象来管理这种类型的对象。它还提取与使用 Driver 对象相关的信息。

(3) Connection：此接口有连接数据库的所有方法。连接(Connection)对象表示通信上下文，即与数据库的所有通信仅通过连接对象。

(4) Statement：使用从此接口创建的对象将 SQL 语句提交到数据库。除了执行存储过程之外，一些派生接口还接收参数。

(5) ResultSet：在使用 Statement 对象执行 SQL 查询后，ResultSet 对象保存从数据库检索的数据。

(6) SQLException：此类处理数据库应用程序中发生的任何错误。

### 3.3.3 建立连接

Java.sql.Connection 接口的实现类负责维护 Java 开发者与数据库之间的会话。特定的数据库需要实现该接口，以便开发者能正确地操作数据库。开发者拥有该类的实例后，就可以访问数据库，并可以执行特定操作。下面列出了该接口中一些常用的重要方法。

(1) Statement createStatement() throws SQLException。

该方法返回一个用于执行 SQL 的 Statement 对象，通过该方法获得 Statement 实例后，即可通过该实例执行 SQL 语句，并获取返回结果。

(2) PreparedStatement prepareStatement(String sql) throws SQLException。

该方法返回一个 SQL 命令执行器 PreparedStatement，PreparedStatement 与 Statement 的区别是，该类在初始化时需要传入一个 SQL，SQL 需要的条件值可通过参数的方式设置，该类会预编译 SQL 命令，因此在执行效率上高于 Statement，但是它只能执行特定的 SQL 语句。

(3) void commit() throws SQLException。

提交数据库事务，默认情况下 connection 会自动提交事务，即执行每条 SQL 语句后都会自动提交，如果取消了自动提交，则必须使用此方法进行提交，否则对数据库的操作将无效。

(4) void rollback() throws SQLException。

取消当前事务的所有数据库操作，将已经修改的数据还原成初始状态。

想要进行数据库连接，首先需要建立与数据库之间的连接会话，所有操作都是基于这个会话基础上进行的。建立连接的代码如下。

```
01  public class MySqlDAO {
02      public static Connection getConnection()   throws Exception{
03          String driverName = "com.mysql.jdbc.Driver";
04          String url = "jdbc:mysql://localhost:3306/simple";
05          String userName = "root";
06          String password = "111111";
07          Class.forName(driverName);
08          Connection con = DriverManager.getConnection(url, userName, password);
09          return con;
10      }
11  }
```

这段代码的主要功能是建立与数据库之间的连接会话。下面来逐一分析这段代码，首先是连接参数定义，代码如下。

```
01  String driverName = "com.mysql.jdbc.Driver";
02  String url = "jdbc:mysql://localhost:3306/simple";
03  String userName = "root";
04  String password = "111111";
```

这段代码块首先定义了几个变量，分别是驱动类名称、数据库的连接字符串、数据库用户名以及数据库密码，请注意以下几点。

（1）驱动名称必须为全名，而且请确保路径正确。

（2）连接字符串分成以下 3 部分。

① 连接协议 jdbc:mysql://。

② 数据库地址 localhost:3306/，包含主机地址和数据库端口号。

③ 数据库名称 simple。

（3）请确保用户名和密码正确，否则会拒绝连接。

接着是加载数据库的驱动，代码如下。

```
01  Class.forName(driverName);
```

加载数据库驱动的时候，驱动程序会自动调用 DriverManager 中的 registerDriver (Driver driver)方法，将自身注册到管理器中。

接下来是创建并获取连接，代码如下。

```
01  Connection con = DriverManager.getConnection(url,userName,
02  password);
```

此处调用了驱动管理器的 getConnection 的三参数方法，驱动管理器对该方法有多个重载。该方法会根据传入的参数选择合适的连接会话返回给用户。至此已经获取了数据库的连接会话，可以在此会话基础上进行数据库操作了。注：执行这段代码前，请确保将数据库驱动 jar 包加入到了项目中。Connection 很昂贵，需要及时关闭。


扫一扫

## 3.3.4 执行数据查询语言

下面介绍如何从数据库中读取数据。

请看以下代码。

```
01  public class SelectTester {
02      public static void main(String[] args) throws Exception{
03      Connection con = MySqlDAO.getConnection();
04      Statement stmt = con.createStatement();
05      String sql = "select id, name, age from student where no >= 1001";
06      ResultSet rs = stmt.executeQuery(sql);
07      while(rs.next()){
```

```
08      System.out.print("学号:" + rs.getInt("no"));
09      System.out.println("   姓名:" + rs.getString(2));
10      }
11      stmt.close();
12      con.close();
13      }
14  }
```

这段代码的功能是读取表 student 中 no 大于或等于 1001 的数据，并打印出来。首先仍然是先获取数据库命令执行对象 Statement 的实例，接着定义一条查询 SQL 语句，执行这条语句会返回一个结果集，结果集中包含了 student 表中 no 大于等于 1001 的学生信息。

JDBC 使用游标来访问结果集中的数据。游标可以理解为指向结果集中一条数据的指针，指针可以向后移动指向下一条数据(游标也可以向前移动，但使用比较少)，调用结果集 ResultSet 的 next()方法就可以向后移动游标。

游标的工作原理如图 3-5 所示。游标最开始指向结果集的第 1 行数据之前，调用 ResultSet.next()方法就可以移动游标到第 1 行，并且 next()方法返回 true(如果移动后指向数据，就返回 true，否则返回 false)。同理，可以循环调用 next()方法，让游标指向结果集中的每一条数据。如果游标指向最后一条数据，再调用 next()，游标将指向最后一条数据之后，并且 next()方法返回 false，表示已经没有可以访问的数据了，上面代码第 7 行的循环将退出，结束数据集的访问。

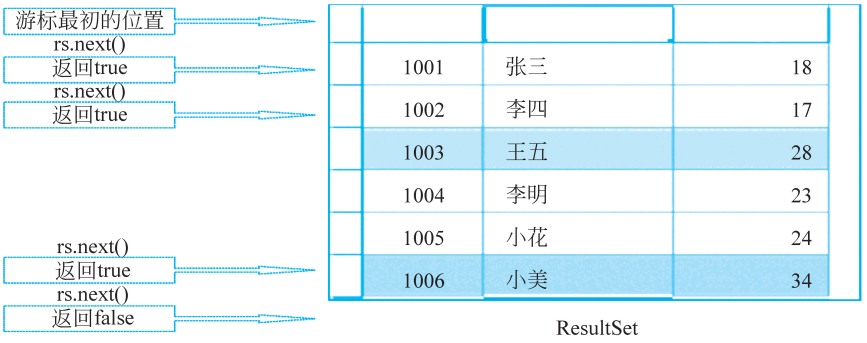


图 3-5　游标的工作原理

如果游标当前指向数据，可以使用 ResultSet 的方法访问当前行的数据。ResultSet 提供了一系列形如 getXXX()的方法访问当前行各个字段的值，例如用 getInt()方法访问 int 类型的字段。所有 getXXX()方法都有两个重载方法，一个传入一个字符串，表示字段的名称，一个是传入整数，表示字段的索引。例如第 8 行的 rs.getInt("no")访问名为 no 的字段，并返回整数；第 9 行的 rs.getString(2)访问第 2 个字段的值，即 name 字段的值，并返回字符串(请注意，字段的索引是根据 select 语句中列出的字段顺序，并且索引是从 1 开始计数的，因此前面的 getString(2)表示访问第 2 个字段 name，而不是第 3 个字段，这与 Java 通常从 0 开始计数不一样)。强烈建议通过字段的名称获取字段的值，不要通过索引。需要注意，字段的类型必须正确，不能通过 getInt 的方法获取字符串字段的值。

JDBC 驱动程序在将 Java 数据发送到数据库之前，会将其转换为相应的 JDBC 类型。对于大多数数据类型都采用了默认的映射关系。例如，一个 Java int 数据类型转换为 SQL INTEGER。通过默认的映射关系来提供驱动程序之间的一致性，如表 3-5 所示。

表 3-5　JDBC 字段类型映射表

| SQL 类型 | JDBC/Java 类型 | setXXX | getXXX |
|---|---|---|---|
| VARCHAR | java.lang.String | setString | getString |
| CHAR | java.lang.String | setString | getString |
| LONGVARCHAR | java.lang.String | setString | getString |
| BIT | boolean | setBoolean | getBoolean |
| NUMERIC | java.math.BigDecimal | setBigDecimal | getBigDecimal |
| TINYINT | byte | setByte | getByte |
| SMALLINT | short | setShort | getShort |
| INTEGER | int | setInt | getInt |
| BIGINT | long | setLong | getLong |
| REAL | float | setFloat | getFloat |
| FLOAT | float | setFloat | getFloat |
| DOUBLE | double | setDouble | getDouble |
| VARBINARY | byte[ ] | setBytes | getBytes |
| BINARY | byte[ ] | setBytes | getBytes |
| DATE | java.sql.Date | setDate | getDate |
| TIME | java.sql.Time | setTime | getTime |
| TIMESTAMP | java.sql.Timestamp | setTimestamp | getTimestamp |
| CLOB | java.sql.Clob | setClob | getClob |
| BLOB | java.sql.Blob | setBlob | getBlob |
| ARRAY | java.sql.Array | setARRAY | getARRAY |
| REF | java.sql.Ref | SetRef | getRef |
| STRUCT | java.sql.Struct | SetStruct | getStruct |

JDBC 为什么要通过游标来访问数据集中的数据，而不是像访问数组中的数据那样，通过索引来访问呢？这是因为 JDBC 通过 Statement 执行 SQL 语句，创建了 ResultSet 对象，JDBC 并不是把 select 查询到的所有数据都检索（fetch）出来，并存储在 ResultSet 中，而是只返回查询结果的前面一部分数据和查询结果的数据总数。没有返回的数据在可能需要访问的时候再从数据库中检索出来，并存入 ResultSet 供访问。JDBC 能根据访问的情况异步检索后面的数据，例如访问到结果集的第 50 行时，JDBC 可能会把 100～200 行的数据都异步检索出来，供后面访问。如果使用游标方式顺序访问结果集中的数据，JDBC 可以预测将

要访问的数据，并预先加载，如果采用索引方式，可以随机访问结果集中的数据，这种预测就不可能实现了，这是采用游标方式的主要原因。采用游标方式还有其他原因，例如如果查询的结果集比较大(数万条数据)，Java 内存可能不足以存放这么多数据，导致内存溢出，采用游标方式可以从内存中移除访问过的数据，以存放后面的数据。

## 3.3.5 处理 null 值

SQL 使用 null 值和 Java 使用 null 是不同的概念。那么，可以使用三种策略来处理 Java 中的 SQL null 值。

(1) 避免使用返回基本数据类型的 getXXX()方法。

(2) 使用包装类的基本数据类型，并使用 ResultSet 对象的 wasNull()方法来测试收到 getXXX()方法返回的值是否为 null，如果是 null，该包装类变量则被设置为 null。

(3) 使用基本数据类型和 ResultSet 对象的 wasNull()方法来测试通过 getXXX()方法返回的值，如果是 null，则基本变量应设置为可接收的值来代表 NULL。

下面是一个处理 NULL 值的示例，代码如下。

```
01  Connection con = MysqlDAO.getConnection();
02  Statement stmt = conn.createStatement();
03  String sql = "SELECT id, first, last, age FROM Employees";
04  ResultSet rs = stmt.executeQuery(sql);
05  int id = rs.getInt(1);
06  if(rs.wasNull()) {
07     id = 0;
08  }
```

## 3.3.6 执行数据操作语句


扫一扫

本小节将介绍获取连接会话后，如何通过 Statement 对数据库进行增、删、改、查操作。在对数据库进行操作前，首先需要获取 Statement 对象，用于执行数据操作命令。

下面代码演示了在数据库中插入一条记录的操作。

```
01      public class InsertTester {
02            public static void main(String[] args) throws Exception{
03                Connection con = MySqlDAO.getConnection();
04                Statement stmt = con.createStatement();
05                String sql = "insert into student(id,name,age) values(1007,
06  '小亮',28)";
07                int count = stmt.executeUpdate(sql);
08                System.out.println("成功插入了 "+count+" 条数据");
09                stmt.close();
10                con.close();
11            }
12        }
```

代码第 7 行通过 executeUpdate()执行 insert 语句。executeUpdate()返回 SQL 语句

影响的行数，如果是 insert，表示新增的行数，如果执行的是 update、delete 语句，分别返回修改的行数和删除的行数。修改（update）、删除（delete）语句与插入（insert）操作类似，在此就不赘述了。

### 3.3.7 执行数据定义语句

通过 JDBC 可以操作数据，还可以执行定义数据库结构的数据定义语言（DDL）。以下代码演示了通过 JDBC 在数据库中创建一张数据表，代码如下。

```
01      public class CreateTableTester {
02          public static void main(String[] args) throws Exception{
03              Connection con = MySqlDAO.getConnection();
04              Statement stmt = con.createStatement();
05              String sql = "create table student(no int primary key, name
06  varchar(50),age int)";
07              stmt.execute(sql);
08              stmt.close();
09              con.close();
10          }
11  }
```

第 03 行代码首先通过 MySqlDAO 获取用于数据库连接的 Connection 实例，第 04 行代码获取了数据库命令执行对象 Statement 的实例，然后定义了一条数据库语句。该语句的作用是创建一张 student 表，表中有 3 个字段：整型的 no（学号）、字符串类型的 name（姓名）、整型的 age（年龄）。接着调用 Statement 对象的 execute(String sql)方法，执行这条数据库语句后，将在数据库中创建一张 student 表。


扫一扫

### 3.3.8 预编译 Statement

上面的 Statement 可以用来执行 SQL 语句，JDBC 为了提高执行效率，提供了 PrepareStatement 类来执行需要多次重复执行的语句。例如前面的 insert 语句，如果要一次性插入 10 名学生，除了后面的值不一样，整个语句的样式是一样的。PrepareStatement 类能提高这种语句的总体执行效率。

与 Statement 类似，需要获取一个 PreparedStatement 对象，并为其指定 SQL 语句。

```
01  PreparedStatement ps =
02  MySqlDAO.getConnection().prepareStatement(sql);
```

下面的代码演示了如何使用 PreparedStatement。

```
01      public class PrepareStatement Tester {
02            public static void main(String[] args) throws Exception{
03               Connection con = MySqlDAO.getConnection();
04               String sql = "select * from student where no = ?";
05               PreparedStatement ps = con.prepareStatement(sql);
```

```
            ps.setInt(1, 1);
            ResultSet rs = ps.executeQuery();
            while(rs.next()){
               System.out.print("学号" + rs.getInt("no"));
               System.out.println("   姓名" + rs.getString(2));
            }
            ps.close();
            con.close();
        }
    }
```

与 3.3.6 节通过 Statement 对象操作数据库的代码类似，这段代码仍然是在数据库连接实例上获取 PreparedStatement 实例，并指定一条 SQL 语句，但这条语句不同的是它采用"?"代替了具体的值，PreparedStatement 对象允许在执行 SQL 语句时才进行参数指定。PreparedStatement 对象有多个设置参数值的方法，设置参数时，需要知道参数的值类型，比如 int 类型可以使用 setInt(int paramIndex,int value)进行参数值设定，前一个参数表示参数的索引，即它代表着第几个问号，后面的参数为属性值。

PreparedStatement 对象会预先编译 SQL 语句，因此它的执行效率高于 Statement，但它只能执行预先设定的 SQL，因此多用于指定特定 SQL 的场景中。PreparedStatement 的其他数据库操作与此类似，因此不再赘述。

下面代码展示了一个批量插入的例子，可以看出 PreparedStatement 对象与 Statement 对象两者在执行效率上面的差别。

```
01 public class CompareStatement {
02         static final int SIZE = 5000;
03         public static void main(String[] args) {
04             Connection con = null;
05             try {
06                 //Statement
07                 String sql = "INSERT student(id,name)
08 VALUES(1,'test')";
09                 con = MySqlDAO.getConnection();
10                 System.out.println(con.toString());
11                 long startTime=System.currentTimeMillis();
12                 Statement stmt = con.createStatement();
13                 for (int j = 0; j < SIZE; j++) {
14                     stmt.execute(sql);
15                 }
16                 stmt.close();
17                 System.out.println("Statement 运行时间:
18 "+(System.currentTimeMillis() - startTime)+"ms");
19                 //PreparedStatement
20                 String psql =  "INSERT student(id,name)
```

```
21  VALUES(?,?)";
22                  startTime=System.currentTimeMillis();
23                  PreparedStatement ps =
24  con.prepareStatement(psql);
25                  for (int i=0; i < SIZE; i++) {
26                      ps.setInt(1, 1);
27                      ps.setString(2, "test");
28                      ps.execute();
29                  }
30                  ps.close();
31                  System.out.println("PreparedStatement 运行时
32  间: "+(System.currentTimeMillis() - startTime)+"ms");
33              } catch (Exception e) {
34                  e.printStackTrace();
35              }finally {
36                  if(con != null)
37                      try {
38                          con.close();
39                      } catch (SQLException e) {
40                          e.printStackTrace();
41                      }
42              }
43          }
44      }
```

运行结果如下：

```
01  Statement 运行时间: 1942ms
02  PreparedStatement 运行时间: 1783ms
```

从上面的结果可以看出，PreparedStatement 要比 Statement 快。在数据库模型更加复杂的实际生产应用中，PreparedStatement 的优势更加明显。

通过 JDBC 执行一条 SQL 语句，通常需要数毫秒、数百毫秒，比较复杂的语句可能用数秒。需要注意，对于一次请求，数毫秒并不能被认为是微不足道的，相对于普通的 Java 代码，数毫秒已经是很长的时间了。因此，使用 JDBC 访问数据库的时候，要有效率意识，也就是尽量减少 SQL 语句的执行时间，例如创建数据库索引、使用 PreparedStatement、批量更新等。

值得注意的是，MySQL 本身是具有预编译功能的，但是 PreparedStatement 默认不会帮我们开启。只有使用了 useServerPrepStmts＝true 才能开启 MySQL 的预编译，其他代码不变，修改连接数据库的 URL 的代码如下。

```
01  String url =
02  "jdbc:mysql://localhost:3306/simple?useServerPrepStmts=
03  true";
```

如果 useServerPrepStmts＝true，Connection 实例在 PrepareStatement 处会产生一个

ServerPrepareStatement 对象，在此对象开始构造时首先会把当前 SQL 语句发送到 MySQL，进行预编译，然后将返回的结果缓存起来。其中包括预编译的名称（可以看作是当前 SQL 语句编译后的函数名）、签名（参数列表）。执行时会直接把参数传递给这个函数，请求 MySQL 执行这个函数。

此外，如果在代码中使用不同的 PrepareStatement 句柄，就会发现同一个 SQL 语句发生了两次预编译，这不是我们想要的效果。若想要对同一 SQL 语句多次执行，而不是每次都预编译，就要使用 cachePrepStmts=true。这个选项可以让 JVM 端缓存每个 SQL 语句的预编译结果。简单说就是以 SQL 语句为 key，将预编译结果缓存起来。下次遇到相同的 SQL 语句时，作为 key 去获得结果。修改连接数据库的 URL 的代码如下。

```
01  String url =
02  "jdbc:mysql://localhost:3306/simple?useServerPrepStmts=true
03  &cachePrepStmts=true";
```

当然，配置参数不止这两个，此处不赘述。更多说明可以参阅 MySQL 官方文档的 Connector/J 部分。

使用 PreparedStatement 的优势如下。

- 防止注入攻击。
- 防止烦琐的字符串拼接和错误。
- 直接设置对象，而不需要转换为字符串。
- PreparedStatement 使用预编译速度相对 Statement 较快。

通常，一条 SQL 语句在数据库中从接收到最终执行完毕返回，可以分为下面 3 个过程。

（1）词法和语义解析。

（2）优化 SQL 语句，制订执行计划。

（3）执行并返回结果。

这种普通语句被称作 Immediate Statements。

但是在很多情况，一条 SQL 语句可能会反复执行，或者每次执行的时候只有个别的值不同（比如 query 的 where 子句值不同，update 的 set 子句值不同，insert 的 values 值不同）。

如果每次都需要经过上面的词法语义解析、语句优化、制定执行计划等，则效率就明显不高了。

所谓预编译语句，就是将这类语句中的值用占位符替代，可以视为将 SQL 语句模板化或参数化，一般称这类语句为 Prepared Statements 或 Parameterized Statements。

预编译语句的优势可归纳为：一次编译、多次运行，省去了解析优化等过程。此外，预编译语句能防止 SQL 注入。

当然，就优化来说，很多时候最优的执行计划不是光靠知道 SQL 语句的模板就能决定了，往往就是需要通过具体值来预估出成本代价。

为了防止 SQL 注入，采用预编译的方法，先将 SQL 语句中可被客户端控制的参数集进行编译，生成对应的临时变量集，再使用对应的设置方法，为临时变量集里面的元素赋值，赋值函数为 setString()，对传入的参数进行强制类型检查和安全检查，避免了 SQL 注入的产生。下面具体分析。

（1）为什么 Statement 会被 SQL 注入？

Statement 之所以会被 SQL 注入，是因为 SQL 语句结构发生了变化。比如：

```
"select * from user where username='"+uesrname+
"'and password='"+password+"'"
```

用户输入'or true or'之后，SQL 语句结构改变，代码如下。

```
select * from user where username=''or true or'' and
password=''
```

上面的 SQL 语句本意是查询用户名和密码匹配的人员信息，但如果用户注入 or true or 之后，原来查询条件中就包含了 or true 的部分，这就意味着表中的所有数据记录都会满足条件，会返回到程序中。如果这个语句用于登录，就意味着所有的用户都可以登录，这是严重的安全漏洞。

（2）为什么 Preparement 可以防止 SQL 注入？

```
select * from user where username=? and password=?
```

该 SQL 语句会在得到用户的输入之前先用数据库进行预编译，这样不管用户输入什么用户名和密码，判断始终都是“并”的逻辑关系，防止了 SQL 注入。

简单总结，参数化能防止注入的原因在于：语句是语句，参数是参数，参数的值并不是语句的一部分，数据库只按语句的语义执行。

### 3.3.9 批量更新

批量执行 SQL 操作，建议使用 addBatch()方法，此方法将若干 SQL 语句装载到一起，然后一次送到数据库执行，执行只需要很短的时间。上一小节的批量操作是一条一条发往数据库执行的，部分时间消耗在数据库连接的传输上面。数据量越大，addBatch()方法的优势越明显。

Statement 接口中包括如下两个方法。

（1）void addBatch(String sql) throws SQLException；

将给定的 SQL 命令添加到此 Statement 对象的命令列表中，通过调用 executeBatch()可以批量执行此列表中的命令。

（2）int[] executeBatch() throws SQLException；

将一批命令提交给数据库来执行，返回一个整形数组 int[]，数组中每个数字对应一条命令的影响行数。

以下展示的是用 addBatch()方法进行批量更新的例子，代码如下。

```
    public class AddBatchTester {
        static final int SIZE = 5000;
        public static void main(String[] args) {
            Connection con  = null;
            try {
```

```
06                  con = MySqlDAO.getConnection();
07                  con.setAutoCommit(false);
08                  Statement stmt = con.createStatement();
09                  for (int i = 0; i < SIZE; i++) {
10                      stmt.addBatch("INSERT student(no,name)
11  VALUES(1,'test')");
12                  }
13                  stmt.executeBatch();
14                  con.commit();
15              } catch (Exception e) {
16                  e.printStackTrace();
17              }finally {
18                  if(con != null){
19                      try {
20                          con.close();
21                      } catch (SQLException e) {
22                          e.printStackTrace();
23                      }
24                  }
25              }
26          }
27      }
```

在默认环境下，SQL 操作会被自动提交到数据库，无法回滚事务。为了完成在一次数据库连接中完成所有 SQL 语句的执行，第 7 行代码中首先设置 con.setAutoCommit(false)，表示 SQL 命令的提交由应用程序负责，程序必须调用 con.commit()，将先前执行的语句一起提交到数据库，或者调用 con.rollback()方法，取消在当前事务中进行的更改，并且释放 Connection 对象持有的所有数据库锁。关于事务操作等 JDBC 高级操作，将在 3.4 节详细说明。

值得一提的是，跟 JDBC 连接中设置参数类似，添加 rewriteBatchedStatements＝true 参数，可以很大程度地提高操作效率。通过 tcpdump 抓包，并在 wireshark 下做分析发现，若不使用参数，SQL 语句在一次连接中被逐条提交到服务器，该操作一共执行了 5000 次。而加上参数后，JDBC 将 5000 条 insert 语句分成若干条报文，将 SQL 语句分批发送到 MySQL 服务器。每发送一次报文，便插入一批数据进入数据库，实现了批量操作。

## 3.4 JDBC 进阶

本节将讨论一些 JDBC 高级操作，这些操作包括事务、存储过程、数据库连接池、元数据、分页等等。

### 3.4.1 事务


扫一扫

数据库的事务(transaction)是保证数据库完整性、一致性的一种机制，即把多个相关联

的数据库操作当作一个原子性的操作对待，要么同时成功，要么同时失败，不允许同一个事务中的部分操作成功，这有点类似 Java 的 synchronized 机制。

数据库系统保证在一个事务中的所有 SQL 操作要么全部执行成功，要么全部不执行，事务具有 ACID 特性。

- 原子性(atomicity)：表示事务包含的操作要么全部成功，要么失败回滚(恢复到初始状态)。
- 一致性(consistency)：事务执行前后必须保持一致，比如在转账时，转出的数量需要等于转入的数量。
- 隔离性(isolation)：当多个事务并发性访问数据库时，每个事务不能被其他事务干扰。比如数据库中有两个事务，记为 T1、T2，两个事务同时更新一份共享数据，T1 事务应该等待 T2 事务结束后执行，反之亦可。
- 持久性(durability)：表示一个事务一旦被提交了，对数据库的改变是永久的，即使数据库发生了故障，也可以恢复。

事务的通常工作过程如下。

(1) 开启事务(Begin Transaction)。

(2) 执行数据库操作 1。

(3) 执行数据库操作 2，以及其他操作。

(4) 如果发生错误，回滚事务(rollback transaction)，即把数据库恢复到事务开启前的状态。

(5) 如果没有错误，提交事务(commit transaction)，即持久化前面的所有操作。

数据库事务可以并发执行。并发执行会带来一系列的问题：脏读(dirty read)、不可重复读(non repeatable read)、幻读(phantom read)。下面举例说明这些问题。

脏读(dirty read)。例如有两个事务 T1、T2，T1 开启事务，然后修改学生张三的年龄为 19 岁，这时 T2 开启事务，读取学生张三的年龄，T2 读取的年龄应该是原来的 18 岁，还是 T1 修改后的 19 岁呢？从时效性来说，应该读取到最新的数据 19 岁，但是如果事务 T1 没有正常提交，而是回滚了，即 T1 把数据恢复到开启事务前的 18 岁，那么 T2 读出的 19 岁就是错误的(这个错误的数据被称为脏数据)。

不可重复读(non repeatable read)。例如 T1 读取年龄小于 20 岁的学生，读取到张三和李四的信息，此时 T2 修改张三的年龄为 21 岁，如果此时 T1 重新读取年龄小于 20 岁的学生，只能读取到李四，这种在一个事务中两次读到的数据不一样的问题称为不可重复读。

幻读(phantom read)。例如 T1 读取年龄大于等于 18 岁的学生，将读取到张三，此时 T2 修改李四的年龄为 21 岁。如果此时 T1 重新读取大于等于 18 岁的学生，将读取到张三和李四。这种在事务执行过程中，当两个完全相同的查询语句执行得到不同的结果集的现象被称为幻读。

导致上面 3 种问题的原因是事务之间没有有效的隔离，即同时执行的两个事务相互影响。为了解决以上 3 种问题，数据库采用了事务隔离机制。根据事务隔离的程度可以分为不同的事务界别(isolation level)，高级别的隔离能解决以上的所有问题，但是数据库的执行效率将降低。SQL 标准定义了 4 种隔离级别，分别对应可能出现的数据不一致的情况，如表 3-6 所示。

表 3-6 事务隔离

| 隔离级别(isolation level)<br>存在的问题 | 脏读<br>(dirty read) | 不可重复读<br>(non repeatable read) | 幻读<br>(phantom read) |
|---|---|---|---|
| 未提交读(read uncommitted) | 存在 | 存在 | 存在 |
| 提交读(read committed) | — | 存在 | 存在 |
| 可重复读(repeatable read) | — | — | 存在 |
| 可序列化(serializable) | — | — | — |

对应用程序来说,数据库事务非常重要,很多运行着关键任务的应用程序都必须依赖数据库事务,保证程序的结果正常。举个例子:假设小明准备给小红支付 100 元,两人在数据库中的记录主键分别是 123 和 456,那么用两条 SQL 语句的操作代码如下。

```
01  UPDATE accounts SET balance = balance - 100 WHERE id=123;
03  UPDATE accounts SET balance = balance + 100 WHERE id=456;
```

这两条语句必须以事务方式执行才能保证业务的正确性。因为一旦第一条 SQL 执行成功而第二条 SQL 失败,系统的钱就会凭空减少 100 元,而有了事务,要么这笔转账成功,要么转账失败,双方账户的钱都不变。

要在 JDBC 中执行事务,本质上就是如何把多条 SQL 包裹在一个数据库事务中执行。

JDBC 中使用事务机制的代码结构如下所示。

```
01  Connection con = openConnection();
02  try {
03      //关闭自动提交:
04      con.setAutoCommit(false);
05      //执行多条 SQL 语句:
06      insert(); update(); delete();
07      //提交事务:
08      con.commit();
09  } catch (SQLException e) {
10      //回滚事务:
11      con.rollback();
12  } finally {
13      con.setAutoCommit(true);
14      con.close();
15  }
```

其中,开启事务的关键代码是 con.setAutoCommit(false),表示关闭自动提交(JDBC 默认是执行每条 SQL 语句自动开启事务,并自动提交事务)。提交事务的代码在执行完指定的若干条 SQL 语句后调用 con.commit()提交事务。要注意,事务不是总能成功,如果事务提交失败,会抛出 SQL 异常(也可能在执行 SQL 语句的时候就抛出了),此时必须捕获并调用 con.rollback()回滚事务。最后,在 finally 中通过 con.setAutoCommit(true)把 Connection

对象的状态恢复到初始值。

实际上，默认情况下，我们获取到 Connection 连接后，总是处于“自动提交”模式，也就是每执行一条 SQL 都是作为事务自动执行的，这也是为什么前面几节更新操作总能成功的原因：因为默认有这种“隐式事务”。只要关闭了 Connection 的 autoCommit，就可以在一个事务中执行多条语句，事务以 commit()方法结束。

如果要设定事务的隔离级别，可以使用如下代码。

```
01  //设定隔离级别为 READ COMMITTED:
02      con.setTransactionIsolation(Connection.TRANSACTION_READ_
03  COMM ITTED);
```

如果没有调用上述方法，会使用数据库的默认隔离级别。MySQL 的默认隔离级别是 REPEATABLE READ。

JDBC 对事务的操作主要通过 Connection 类的 rollback()、commit()以及 setAutoCommit(boolean autoCommit)等方法进行事务回滚、提交操作。

下面为一个事务实例的代码。

```
01  public class DBTest7 {
02            public static void main(String[] args){
03               Connection con  = null;
04               try{
05                  con = MySqlDAO.getConnection();
06                  con.setAutoCommit(false);
07                  Statement stmt = con.createStatement();
08                  String sql1 = "select max(no) from student";
09                  ResultSet rs = stmt.executeQuery(sql1);
10                  int no = 0;
11                  while(rs.next()){
12                     no = rs.getInt(1) + 1;
13                  }
14                  String sql2 = "insert into student values(" + no + ",
15  'wahaha')";
16                  stmt.execute(sql2);
17                  con.commit();
18                  stmt.close();
19                  con.close();
20               }catch(Exception e){
21                  try {
22                     con.rollback();
23                     con.close();
24                  } catch (SQLException e1) {
25                     e1.printStackTrace();
26                  }
```