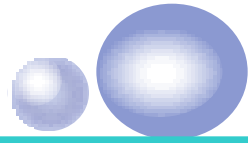
第7章 JDBC数据库访问



补充：MySQL数据库简介

7.1 JDBC技术概述

7.2 JDBC API介绍

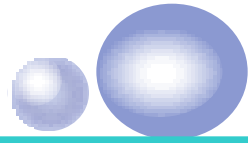
7.3 传统的数据库连接方法

7.4 连接池与数据源

7.5 DAO设计模式

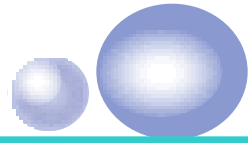
7.7 小结

补充： MySQL数据库简介



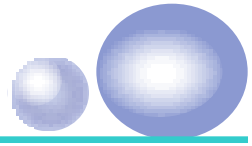
- MySQL是对象关系型数据库管理系统(ORDBMS).
- 它由MySQL AB开发、发布和支持，目前是是最受欢迎的开源SQL数据库管理系统，有着非常广泛的用户。
- MySQL是一个快速的、多线程、多用户和健壮的SQL数据库服务器,支持事务、子查询、多版本并发控制、数据完整性检查等特性，并且支持多语言的应用开发。
- 它能在包括Linux、FreeBSD和Windows等多种平台下运行。

MySQL的下载和安装



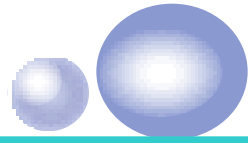
- MySQL官方站的下载页面为：
  http://dev.mysql.com/downloads/
- 在获得了MySQL后，接着所需要的就是安装。MySQL安装过程

Navicat for MySQL



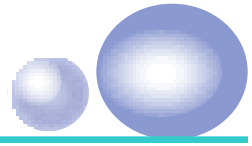
- Navicat for MySQL是一个强大的MySQL数据库服务器管理和开发工具。它可以与任何3.21或以上版本的MySQL一起工作，并支持大部分的MySQL最新功能。
- 其精心设计的图形用户界面(GUI) 可以让你用一种安全简便的方式快速并容易地创建，组织，访问和共享信息。
- Navicat for MySQL安装

使用Navicat for MySQL操作数据库



- 1. 创建用户并分配权限
- 2.新建连接
- 3. 创建数据库
- 4. 创建数据库表
- 5. 操作数据库表

7.1 JDBC 技术概述



- 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，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：
  - 建立与数据库的连接；
  - 发送SQL语句；
  - 处理数据库操作结果。

7.1.1数据库访问的两层和三层模型

- 两层模型即客户机/数据库服务器结构，也就是通常所说的C/S(Client/Server)结构

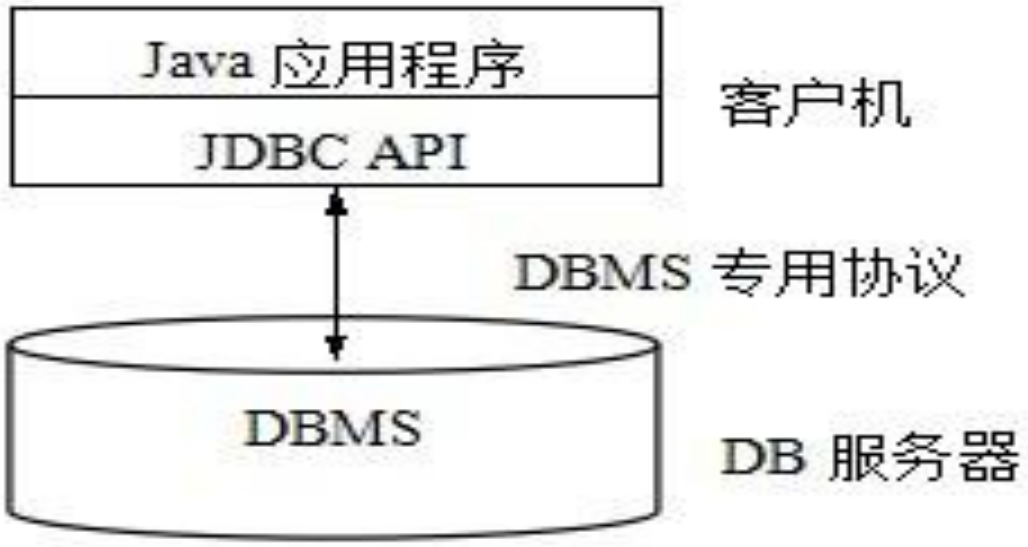


图 7-1　数据库访问的两层模型

7.1.1数据库访问的两层和三层模型

- 三层模型是指客户机/应用服务器/数据库服务器结构，也就是通常所说的B/S(Browser/Server)结构

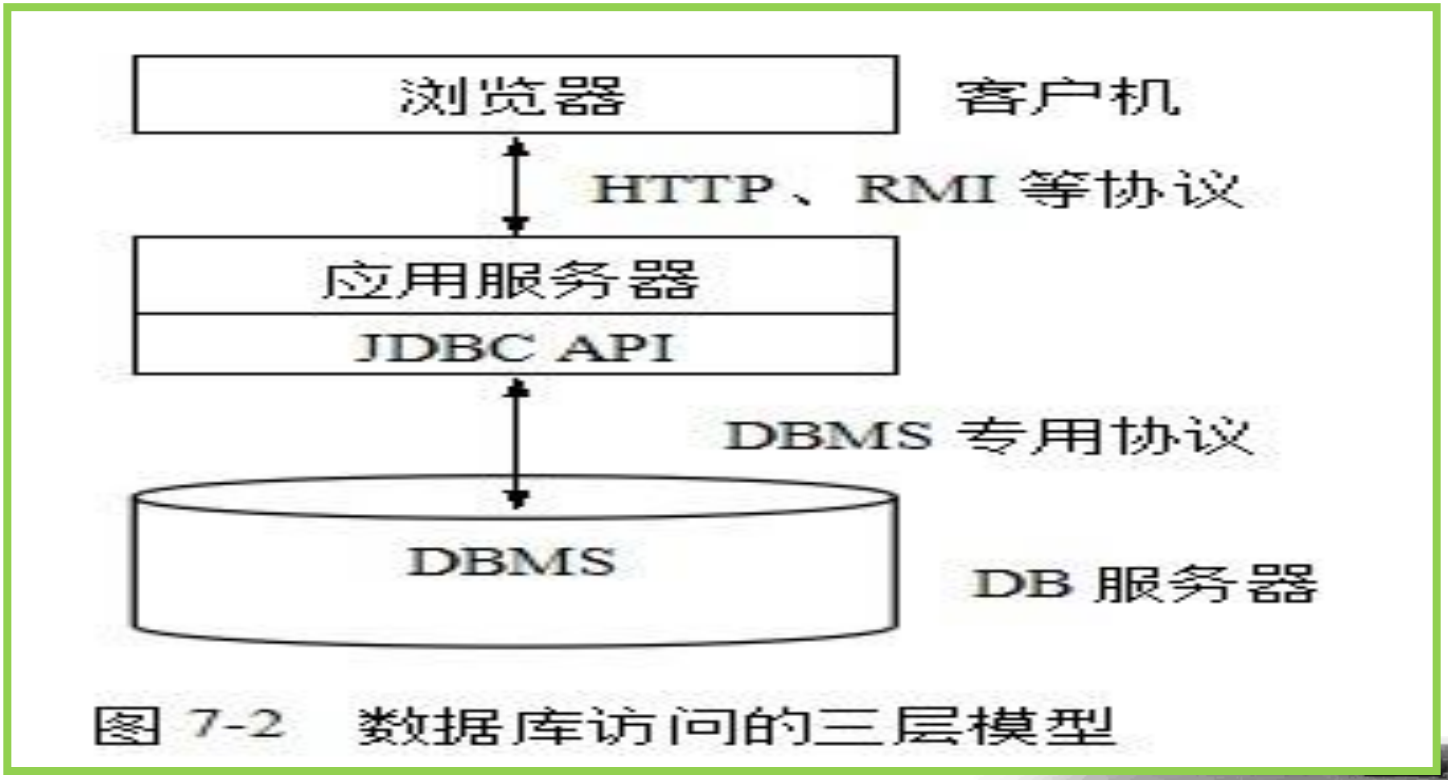
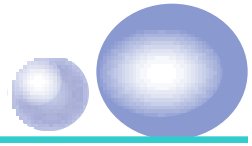


图 7-2　数据库访问的三层模型

7.1.2 JDBC驱动程序



- 目前有多种类型的数据库，每种数据库都定义了一套API，这些API一般是用C/C++语言实现的。
- 因此需要有在程序收到JDBC请求后，将其转换成适合于数据库系统的方法调用。
- 把完成这类转换工作的程序叫做数据库驱动程序。

7.1.2 JDBC驱动程序

❖在Java程序中可以使用的数据库驱动程序主要有四种类型，常用的有下面两种：

- **JDBC-ODBC桥驱动程序**
- **专为某种数据库而编写的驱动程序**

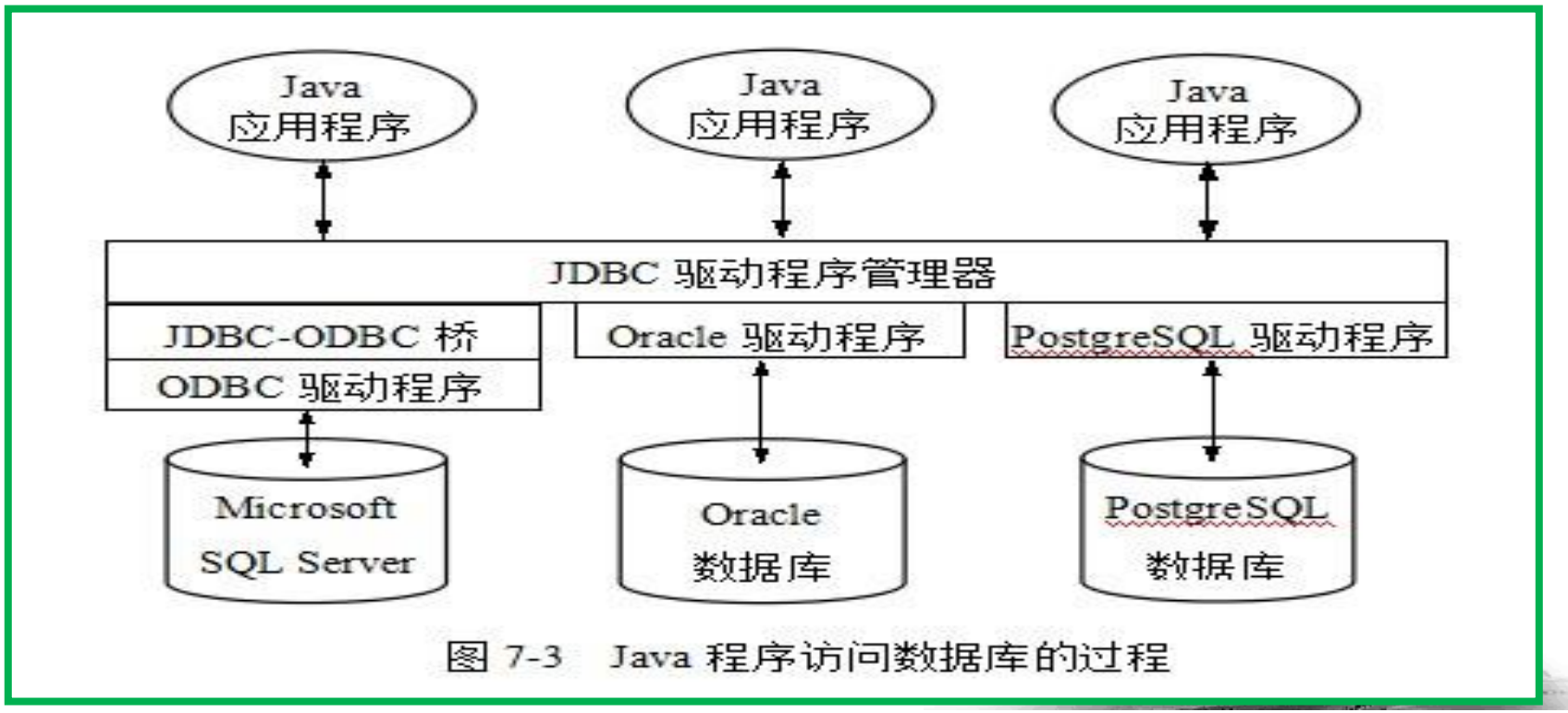
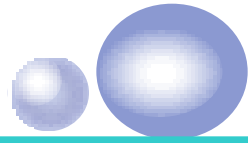


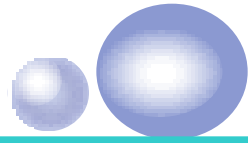
图 7-3　Java 程序访问数据库的过程

7.1.3 安装JDBC驱动程序



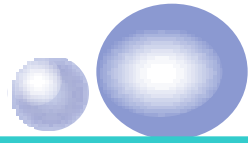
- 使用JDBC-ODBC桥驱动程序连接数据库，不需要安装驱动程序，因为在Java API中已经包含了该驱动程序。
- 使用专用驱动程序连接数据库，必须安装驱动程序。
  - 在开发Web应用程序中，需要将驱动程序打包文件复制到:
    - Tomcat安装目录的lib目录中
    - Web应用程序的WEB-INF\classes\lib目录中。

7.2 JDBC API介绍



- JDBC API是Java语言的标准API，它通过两个包提供的：java.sql包 javax.sql包。
- java.sql包提供了基本的数据库编程的类和接口，如驱动程序管理类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。

java.sql包中的类和接口介绍



7.2.1 Driver接口

7.2.2 DriverManager类

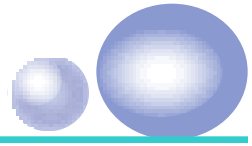
7.2.3 Connection接口

7.2.4 Statement接口

7.2.5 PreparedStatement接口

7.2.6 ResultSet接口

7.2.1 Driver接口



- 驱动程序是实现了Driver接口的类，它一般由数据库厂商提供。
- 加载驱动程序使用Class类的forName()静态方法，格式为：

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

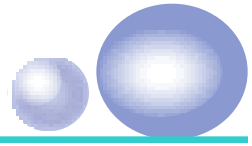
- 如果使用ODBC-JDBC桥驱动程序，语句为：

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

- 如果使用MySQL数据库，语句为：

```
Class.forName("com.mysql.jdbc.Driver");
```

7.2.2 DriverManager类



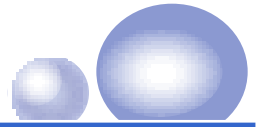
- DriverManager类维护一个注册的Driver类的列表。
- 驱动程序加载成功后应使用DriverManager类的getConnection()建立数据库连接对象
- getConnection()方法的声明格式为：

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

- 如果通过JDBC-ODBC桥驱动程序连接数据库，String dburl = "jdbc:odbc:Bookstore" ;
- 如果使用专门的驱动程序连接数据库，例如要连接MySQL数据库，它的JDBC URL为：

```
String dburl =
   "jdbc:mysql://localhost:3306/bookstore?
   user=root&password=123456";
Connection conn =
   DriverManager.getConnection(dburl);
```

常用的数据库JDBC 连接代码



数据库	连接代码
Oracle	Class.forName(“oracle.jdbc.driver.OracleDriver”); Connection conn=DriverManager.getConnection(“jdbc:oracle:thin:@dbServerIP:1521:ORCL”, user,password);
PostgreSQL	Class.forName(“org.postgresql.Driver”); Connection conn=DriverManager.getConnection(“jdbc: postgresql: //dbServerIP/dbName”,user, password);

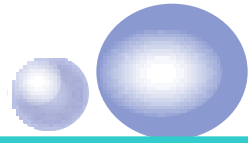
MySQL	Class.forName(“com.mysql.jdbc.Driver”);
	Connection conn=DriverManager.getConnection(
	“jdbc:mysql: //dbServerIP:3306/dbName?
	user=userName&password=password”);

常用的数据库JDBC 连接代码

数据库	连接代码
ODBC	Class.forName(“sun.jdbc.odbc.JdbcOdbcDriver”); Connection conn=DriverManager.getConnection (“jdbc:odbc:DSNName”,user,password);

SQL Server	Class.forName(“com.microsoft.jdbc.sqlserver.
	SQLServerDriver”);
	Connection conn=DriverManager.getConnection(
	“jdbc:microsoft:sqlserver: //dbServerIP:1433;
	databaseName=master”,user,password);

7.2.3 Connection接口



- Connection对象代表与数据库的连接，也就是在加载的驱动程序与数据库之间建立连接。
- 可以使用Connection接口的不同方法创建不同的语句对象。

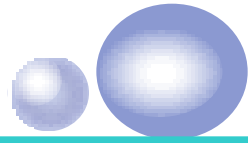
- 创建Statement对象：

  Statement stmt = conn.createStatement();
- 创建PreparedStatement对象：

  PreparedStatement pstmt=conn.prepareStatement();
- 创建CallableStatement对象：

  CallableStatement cstmt =conn.prepareCall() ;

7.2.4 Statement接口



- Statement接口对象主要用于执行一般的对数据库操作SQL语句.
- 调用Statement对象的方法可以执行SQL语句，返回一个ResultSet结果集对象。
- public ResultSet executeQuery(String sql) 该方法用来执行SQL查询语句。查询结果以ResultSet对象返回。

例如： String sql = "SELECT * FROM books" ;

- ResultSet rst = stmt.executeQuery(sql) ;

❖**public int executeUpdate(String sql)**

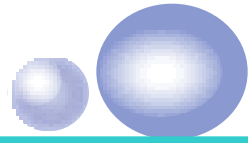
执行由字符串**sql**指定的**SQL**语句，该语句可以是**INSERT**、**DELETE**、**UPDATE**语句或者无返回的**SQL**语句，返回值是更新的行数，如果语句没有返回则返回值为**0**。

❖**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**。

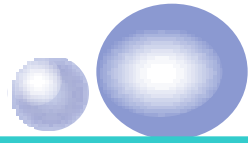
❖**public void close()**

7.2.5 PreparedStatement接口



- 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。

7.2.5 PreparedStatement接口



- 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。
- 创建PreparedStatement对象与创建Statement对象不同。需要给创建的PreparedStatement对象传递一个SQL命令。
- 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。

例：String sql = "INSERT INTO books(bookid, title, price) VALUES(?, ?, ?) ";
PreparedStatement pstmt = conn.prepareStatement(sql);

- SQL字符串中使用问号（?）作为占位符，在SQL语句执行时将被数据替换。
- 从SQL字符串左边开始，第一个占位符的序号为1，依次类推。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。

设置占位符

通过PreparedStatement接口中定义的setXXX()方法为占位符设置具体的值。

例如：

```
public void setInt(int parameterIndex,int x)
public void setString(int parameterIndex, String x)
```

例：
```
pstmt.setString(1,"801");
pstmt.setString(2,"Java EE 编程指南");
pstmt.setDouble(3, 49.00);
```

执行预处理语句

通设置好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() 。

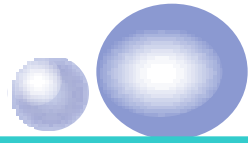
```
ResultSet result = pstmt.executeQuery();
```

- 对更新语句，应该调用executeUpdate() 。

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

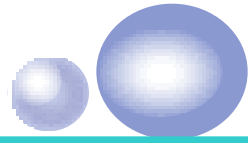
- 注意:对预处理语句，必须调用这些方法的无参数版本

7.2.6 ResultSet接口



- ResultSet对象表示SQL SELECT语句查询得到的记录集合，称为结果集。
- 结果集一般是一个记录表，其中包含多个记录行和列标题。通过调用ResultSet对象的方法，可以对查询结果处理。
- 每个结果集对象都有一个游标。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()方法，使游标定位到下一条记录。

7.2.6 ResultSet接口



游标，结果集的指针指向第一条数据之上。

ResultSet

next:+1

bookid	title	author	publisher	price
201	Servlets 与JSP核心教程	Marty Hall	清华大学出版社	90
202	Tomcat 与Java Web开发技	孙卫琴	机械工业出版社	45
203	JSP完全学习手册	张银鹤	清华大学出版社	50
204	Head First Servlets $ JSP	Bryan Basham	中国电力出版社	98
205	JAVA EE 5开发指南	Kevin Mukhar	机械工业出版社	69

- 如果游标指向一个具体的行，就通过调用ResultSet对象的方法，可以对查询结果处理。

1. 检索当前行字段值

- ResultSet接口提供了检索行的字段值的方法。

  String getString(int columnIndex)

  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结果行中的列名。

- 下面列出了返回其他数据类型的一些方法。

getShort(int columnIndex)

byte getByte(int columnIndex)

int getInt(int columnIndex)

long getLong(int columnIndex)

float getFloat(int columnIndex)

double getDouble(int columnIndex)

boolean getBoolean(int columnIndex)

Date getDate(int columnIndex)

Object getObject(int columnIndex)

Blob getBlob(int columnIndex)

Clob getClob(int columnIndex)

2. 对行操作的方法

- 将光标从当前位置向前移一行。

  boolean next() throws SQLException

- 将光标移动到ResultSet对象的最后一行。

  boolean last() throws SQLException

- 将光标移动到此 ResultSet 对象的上一行。

  boolean previous() throws SQLException

- 获取光标是否位于此 ResultSet 对象的第一行。

  boolean isFirst() throws SQLException

- 获取光标是否位于此 ResultSet 对象的最后一行。

  boolean isLast() throws SQLException

- 获取当前行编号。

  int getRow()

- 例如：

```
String sql = "SELECT * FROM books" ;
ResultSet rst = stmt.executeQuery(sql) ;
while(rst.next()){
System.out.print(rst.getString(1)+"\t") ;
```

- 调用executeQuery(String sql)方法，该方法的返回类型为**ResultSet**，
- 再通过调用ResultSet的方法可以对查询结果的每行进行处理。
- 本例依次输出 第一列bookid的值

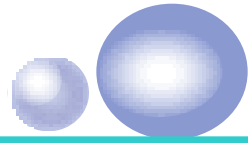
数据类型转换

- 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getXxx()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getXxx()方法就是把SQL数据类型转换为Java语言数据类型，

SQL数据类型与Java数据类型的转换。

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
SMALLINT	Short	TIME	java.sql.Time
INTEGER	Int	TIMESTAMP	java.sql.Timestamp
REAL	float	CLOB	Clob
FLOAT	double	BLOB	Blob
BIGINT	Long	STRUCT	Struct

7.3 传统的数据库连接方法



1 加载驱动程序

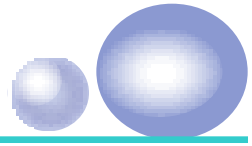
2 建立连接对象

3 创建语句对象

4 获得SQL语句的执行结果

5 关闭建立的对象，释放资源

简单的应用示例

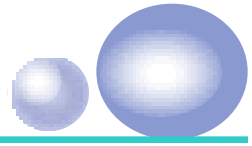


- 本示例程序可以根据用户输入的商品号从数据库中查询该商品信息，或者查询所有商品信息。
- 该例符合MVC设计模式
- 模型：Product.java
- 视图：queryProduct.jsp

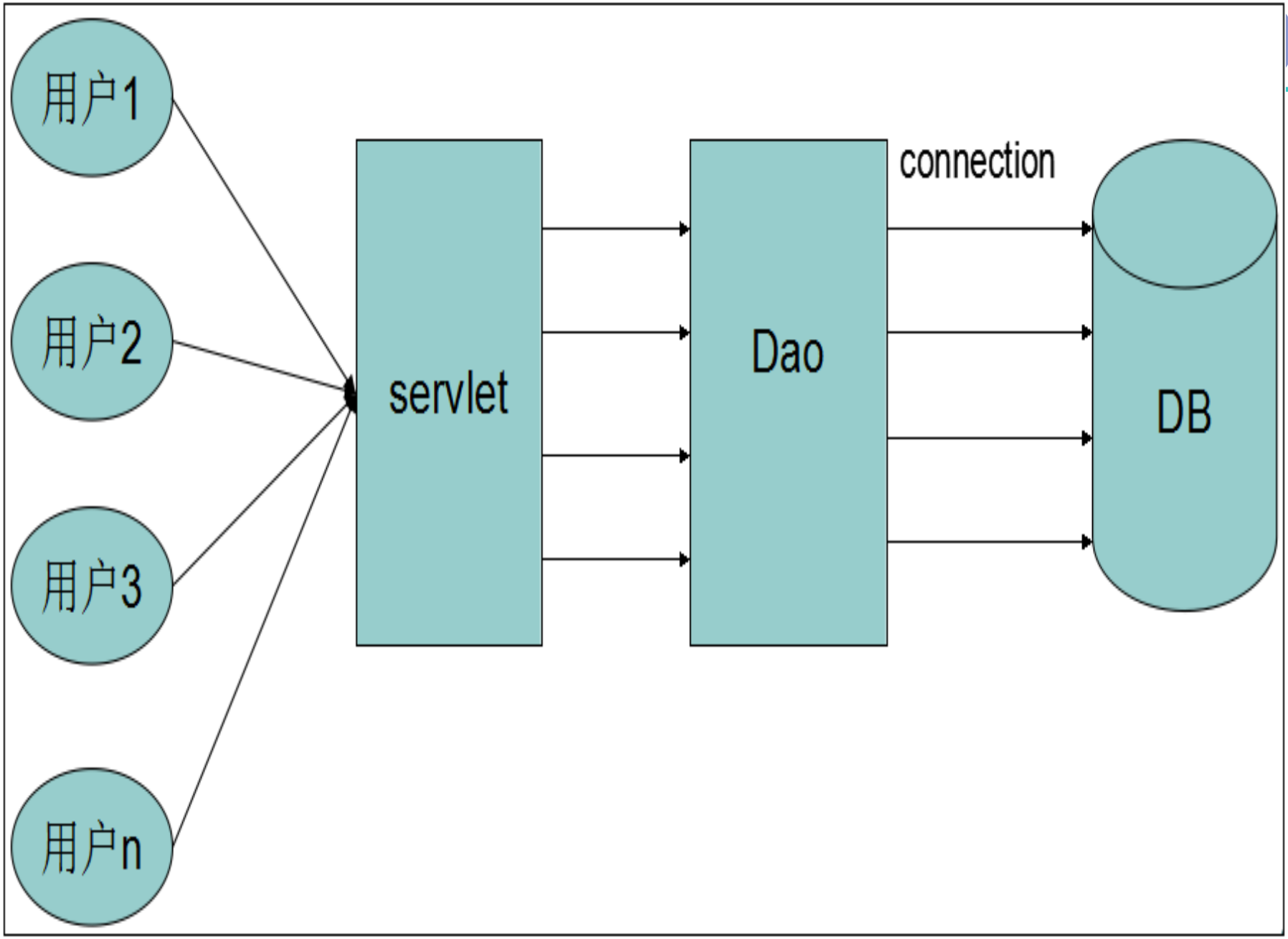
  displayProduct.jsp　　displayAllProduct.jsp

  error.jsp
- 控制器：QueryProductServlet.java
- 运行:http://localhost/chap07/queryProduct.jsp

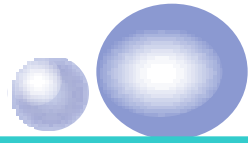
7.4 连接池与数据源



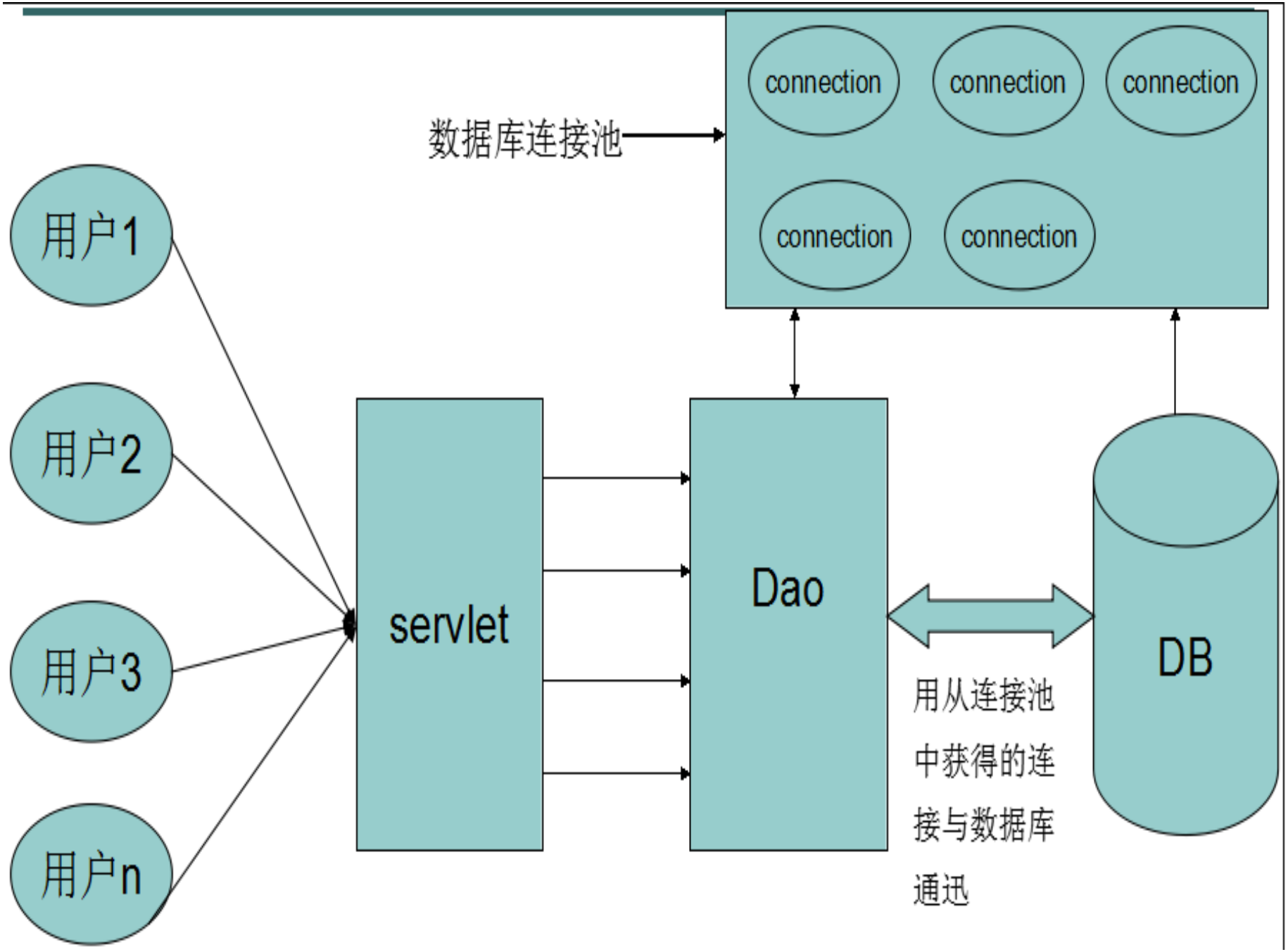
- 在设计需要访问数据库的Web应用程序时，需要考虑的一个主要问题是如何管理Web应用程序与数据库的通信。
- 一种方法是为每个HTTP请求创建一个连接对象，Servlet建立数据库连接、执行查询、处理结果集、请求结束关闭连接。
- 建立连接是比较耗费时间的操作，如果在客户每次请求时都要建立连接，这将导致增大请求的响应时间。


用户1
用户2
用户3
用户n
servlet
Dao
connection
DB

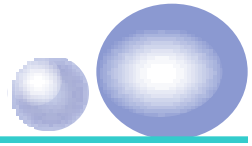
7.4 连接池与数据源



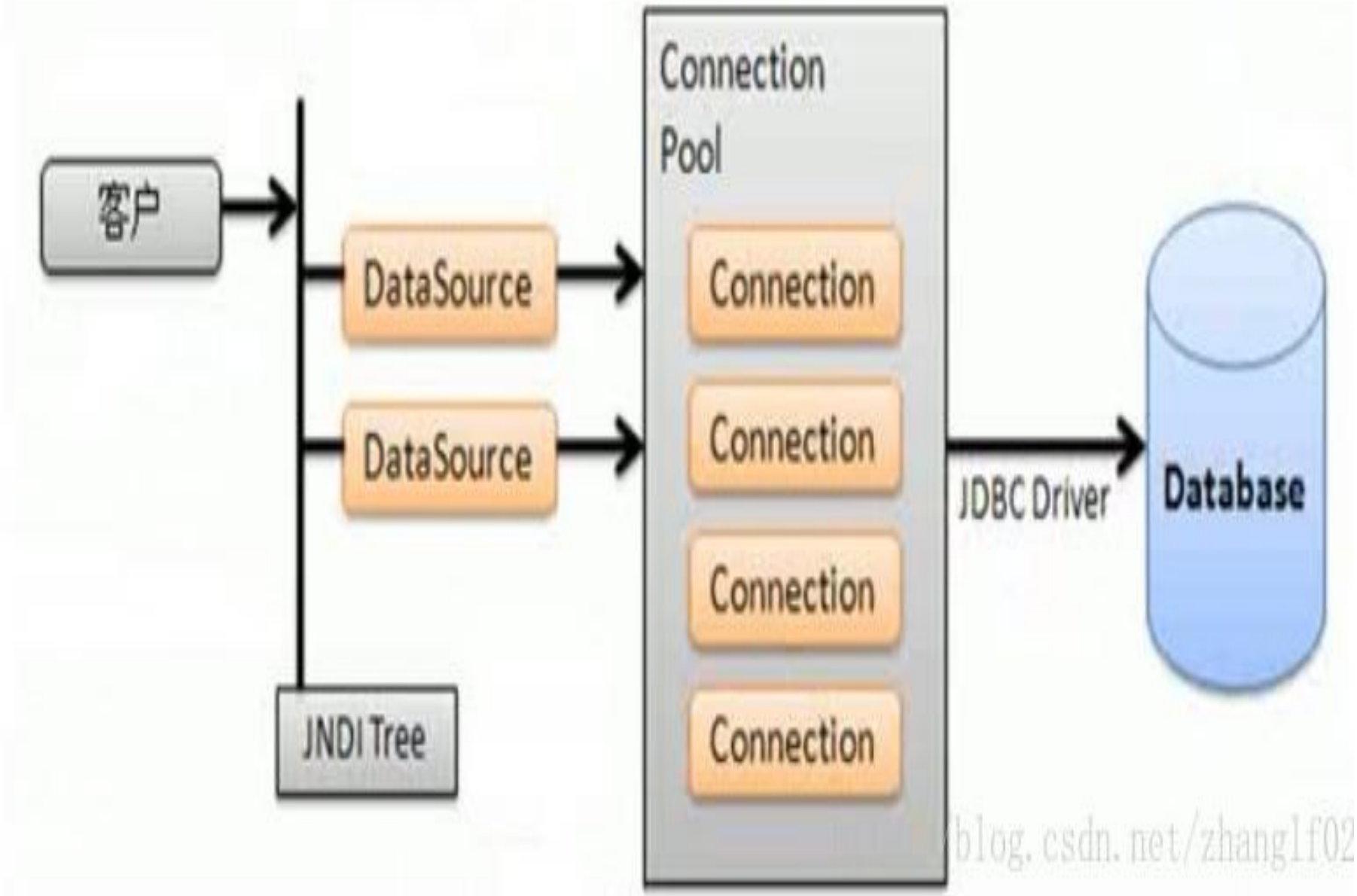
- 为了提高数据库访问效率，从JDBC 2.0开始提供了一种更好的方法建立数据库连接对象，即使用连接池和数据源的技术访问数据库。
- 这种方法是事先建立若干连接对象，将它们存放在数据库连接池（connection pooling）中供数据访问组件共享。
- 使用这种技术，应用程序在启动时就不需要为每个HTTP请求都创建一个连接对象，大大降低请求的响应时间。


connection
connection
connection
connection
connection
数据库连接池
用户1
用户2
用户3
用户n
servlet
Dao
DB
用从连接池
中获得的连
接与数据库
通迅

7.4.1 连接池与数据源介绍

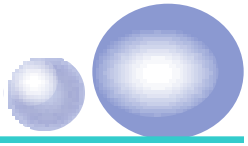


- 连接池预定义了一些连接，当应用程序需要连接对象时就从连接池中取出一个，当连接对象使用完毕将其放回连接池，从而可以避免在每次请求连接时都要创建连接对象。
- 通常DataSource对象是从连接池中获得连接对象。
- 数据源是通过javax.sql.DataSource接口对象实现的，通过它可以获得数据库连接，因此它是对DriverManager工具的一个替代。



- 用DataSource对象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，简称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。

使用数据源访问数据库的步骤



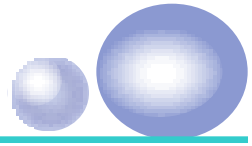
1 配置数据源(局部数据源或全局数据源)；

2 通过JNDI机制查找命名数据源；

3 通过数据源对象创建连接对象；

4 其他与传统方法一致。

7.4.2 配置数据源



❖在Tomcat中可以配置两种数据源：

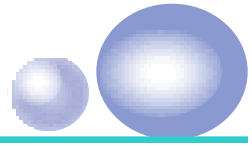
- 局部数据源：局部数据源只能被定义数据源的应用程序使用
- 全局数据源：全局数据源可被所有的应用程序使用。

□ 注意 不管配置哪种数据源，都需要将JDBC驱动程序。

复制到Tomcat安装目录的lib目录

并且需要重新启动Tomcat服务器。

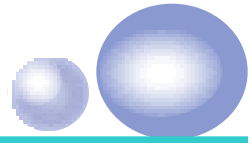
1. 配置局部数据源



- 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：

```
<?xml version="1.0" encoding="utf-8"?>
<Context reloadable = "true">
<Resource
    name="jdbc/sampleDS"
    type="javax.sql.DataSource"
    maxActive="4"
    maxIdle="2"
    username="root"
    maxWait="5000"
    driverClassName=" com.mysql.jdbc.Driver "
    password="123456"
    url=" jdbc:mysql://localhost:3306/company "/>
</Context>
```

2. 在应用程序中访问数据源



- 使用javax.naming.Context接口的lookup()方法查找JNDI数据源

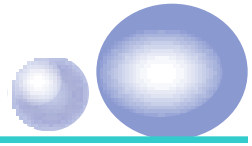
```
Context context = new InitialContext();
DataSource ds = (DataSource)context.lookup(
    "java:comp/env/jdbc/sampleDS");
```

- lookup()方法的参数是数据源名字符串，但要加上“java:comp/env”前缀。
- 得到了DataSource对象的引用后，就可以通过它的getConnection()方法获得数据库连接对象Connection。

```
public void init() {
    try {
      Context context = new InitialContext();
      DataSource ds = (DataSource)context.lookup(
          "java:comp/env/jdbc/sampleDS");
      dbConnection = ds.getConnection();
    }catch(NamingException ne){
       log("Exception:"+ne);
    }catch(SQLException se){
       log("Exception:"+se);
     }
  }
```

- 代码修改:productSearch.jsp ProductSearchServlet.java
- 运行:http://localhost/chap07/productSearch.jsp

3. 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
  - <CATALINA_HOME>/conf/server.xml文件的<GlobalNamingResources>标签定义的。
- 定义后就可在任何的应用程序中使用。
- 假设我们要配置一个名为jdbc/bookstore的数据源，应该按下列步骤操作：

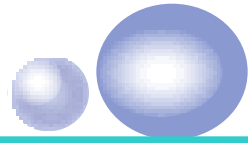
1)首先在server.xml文件的<GlobalNamingResources>标签内增加下面代码：

```
<Resource
    name="jdbc/company"
    type="javax.sql.DataSource"
    maxActive="40"
    maxIdle="2"
    username=“root"
    maxWait="5000"
    driverClassName=" com.mysql.jdbc.Driver"
    password=“123456"
    url="jdbc:mysql://localhost:3306/company"
/>
```

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：

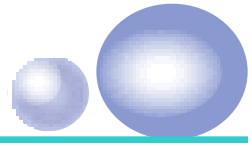
```
<?xml version="1.0" encoding="utf-8"?>
<Context reloadable="true">
  <ResourceLink
    global = "jdbc/company"
    name = "jdbc/sampleDS"
    type = "javax.sql.DataSource"/>
  <WatchedResource>WEB-
   INF/web.xml</WatchedResource>
</Context>
```

数据源的优点



- 数据源(DataSource)即数据来源，调用DataSource.getConnection()即可获取一个连接，而无需关心连到哪个数据库，用户名密码是什么。
- 这 比 DriverManager.getConnection(url, user, password)要先进多了。

DataSource有两种实现方式



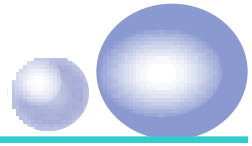
1.直连数据库方式

当调用**DataSource.getConnection()**时，其实它调用的是**DriverManager.getConnection(url, user, password)**来获取一个Connection，Connection使用完后被close,断开与数据库的连接,我们称这总方式是直连数据库,因为每次都需要重新建立与数据库之间的连接,而并没有把之前的Connection保留供下次使用。

2.池化连接方式

- DataSource内部封装了一个连接池，当你获取DataSource的时候，它已经悄悄的与数据库建立了多个Connection，并将这些Connection放入了连接池，此时调用DataSource.getConnection()，它从连接池里取一个Connection。
- Connection使用完后被close，但这个close并不是真正的与数据库断开连接，而是告诉连接池“我”已经被使用完，“你”可以把我分配给其它“人”使用了。
- 就这样连接池里的Connection被循环利用，避免了每次获取Connection时重新去连接数据库。

DataSource与连接池的关系



- DataSource利用连接池缓存Connection，以达到系统效率的提升,资源的重复利用。
- 而连接池它可以单独存在，不需要依靠DataSource来获取连接，你可以直接调用连接池提供的方法来获取连接。
- 目前大多数应用服务器都支持池化连接方式的DataSource。

7.5 DAO设计模式介绍

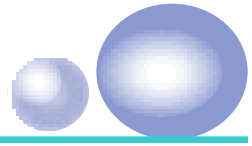
- DAO(Data Access Object)称为数据访问对象。
- DAO设计模式可以在使用数据库的应用程序中实现业务逻辑和数据访问逻辑分离，从而使应用的维护变得简单。
- 它通过将数据访问实现（通常使用JDBC技术）封装在DAO类中，提高应用程序的灵活性。

DAO的使用
- 1)设计持久对象
- 2)设计DAO对象
- 3)使用DAO对象

7.5.1设计持久对象

- 在分布式Web应用中，数据对象可能需要跨层传输，例如可以把数据从表示层持久到业务层，或者反之。
- 跨层持久数据最好的方法是使用持久对象（Persistent Object）。
- 持久对象只包含数据元素，不包含任何业务逻辑，业务逻辑由业务对象（Business Object）实现。
- 持久对象必须是可序列化的，也就是说它的类必须实现java.io.Serializable接口。
- 下面程序定义的Customer类就是持久对象。该持久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。 Customer.java

7.5.2 设计DAO对象



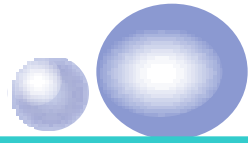
- DAO设计模式有许多种变体，这里介绍一种简单的。先定义一个基类**BaseDao**连接数据库，通过该类可以获得一个连接对象。
- BaseDao.java
- 然后定义**CustomerDao**类，其中定义了添加客户、查找客户、查找所有客户的方法。
- CustomerDao.java
- 下面的**addCustomer.jsp**页面通过一个表单提供向数据库中插入的数据。
- 程序7.13 addCustomer.jsp

- 下面的AddCustomerServlet使用了DAO对象和传输对象，通过JDBC API实现将数据插入到数据库中。
- 程序7.14 AddCustomerServlet.java
- 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insertCustomer()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addCustomer.jsp页面。
- 运行：http://localhost/chap07/addCustomer.jsp

7.6 小 结

- Java程序是通过JDBC API访问数据库。
- 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。

7.6 小 结



- 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。
- 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。