

# J2EE与中间件技术

——JDBC/JTA

# JDBC技术

- ◆ JDBC——Java数据库连接（Java DataBase Connectivity）
- ◆ <http://download.oracle.com/javase/tutorial/jdbc/>

# JDBC技术

- ◆ ODBC——开放式数据库连接
- ◆ 使用针对于数据库的驱动程序，提供一组用于访问任何数据库的API

# JDBC驱动程序

- ◆ JDBC规范支持的产品：把基于JDBC的应用程序中的调用规则映射为针对于数据库的正确调用——JDBC驱动程序
- ◆ 访问特定类型的数据库：必须使用该数据库特定的JDBC驱动程序
- ◆ 由数据库供应商提供

# JDBC API

- ◆ Java环境提供了创建能与数据库交互作用的Java应用程序所必须的JDBC API，将Java命令转换为SQL语句
- ◆ JDBC API：由一组类和接口定义的方法构成，为Java开发人员提供了一个行业标准API，提供了数据库的调用层接口

# Java Database Connectivity API

- ◆ The JavaDatabase Connectivity (JDBC) API lets you invoke SQL commands from Java programming language methods.
- ◆ You use the JDBC API in an enterprise bean when you have a **session bean** access the database.
- ◆ You can also use the JDBC API from a **servlet** or a **JSP** page to access the database directly without going through an enterprise bean.

# Java Database Connectivity API

- ◆ The JDBC API has two parts: **an application-level interface** used by the application components to access a database, and **a service provider interface** to attach a JDBC driver to the Java EE platform.

# JDBC的作用

- ◆ 开发人员只需了解一组API，就可以访问任何关系数据库
- ◆ 不必为不同的数据库重新编写代码
- ◆ 不必了解数据库供应商的特定API
- ◆ 几乎每个数据库供应商都具有几种类型的JDBC驱动



# JDBC技术

## ◆ 实现Java程序与数据库之间的交互

- Java程序调用JDBC API的方法；
- JDBC 驱动程序将调用转换为数据库可以理解的形式；
- 执行操作，JDBC 驱动程序从数据库得到结果，将结果转换为JDBC API 类；
- Java程序得到结果。

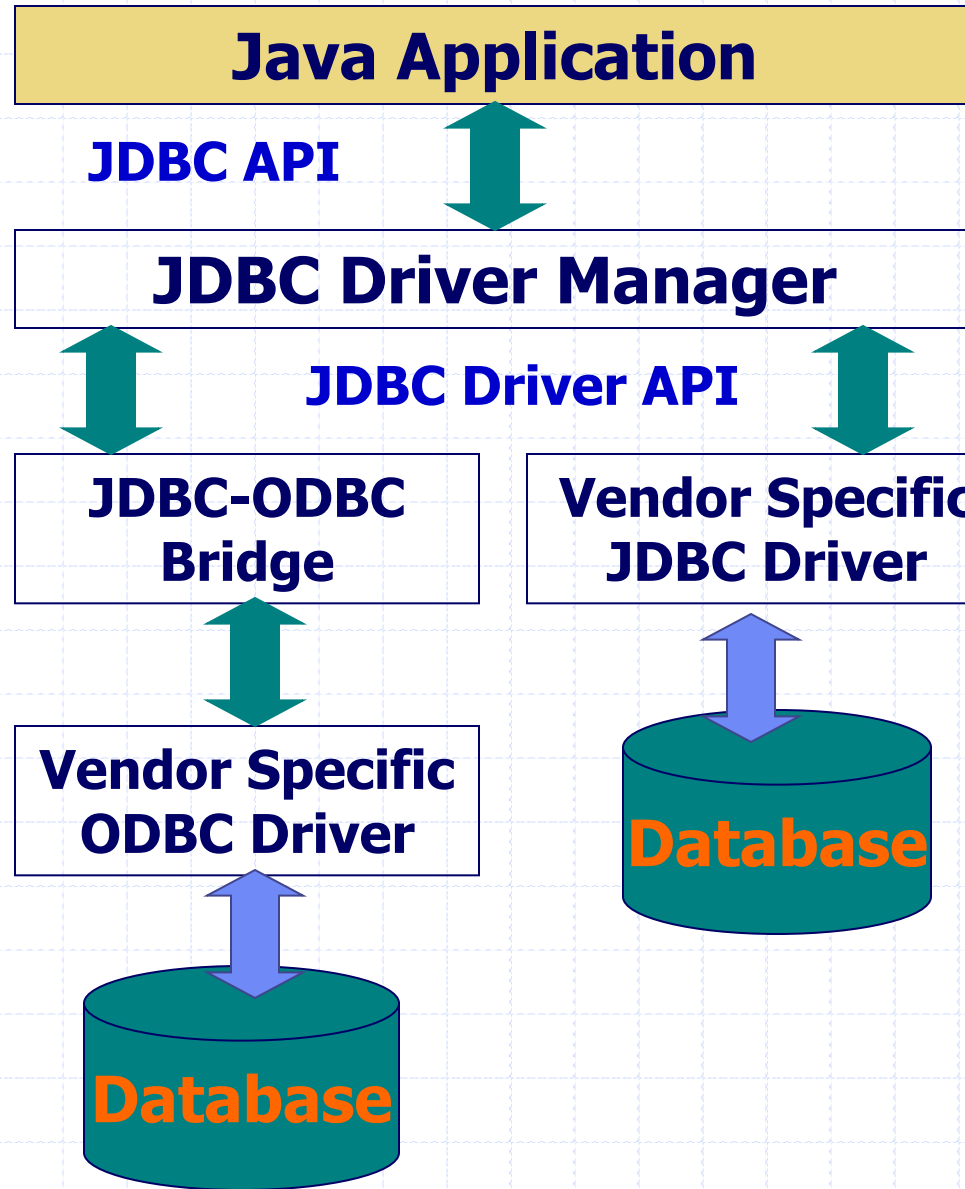
# 使用JDBC读取数据

- ◆ 建立数据库连接：使用DriverManager机制或DataSource机制
- ◆ 向数据库提交查询请求
- ◆ 读取查询结果
- ◆ 处理查询结果
- ◆ 释放连接

# JDBC API

## ◆ 两个程序包：

- java.sql 核心API——DriverManager 机制
- javax.sql 可扩展API——DataSource 机制



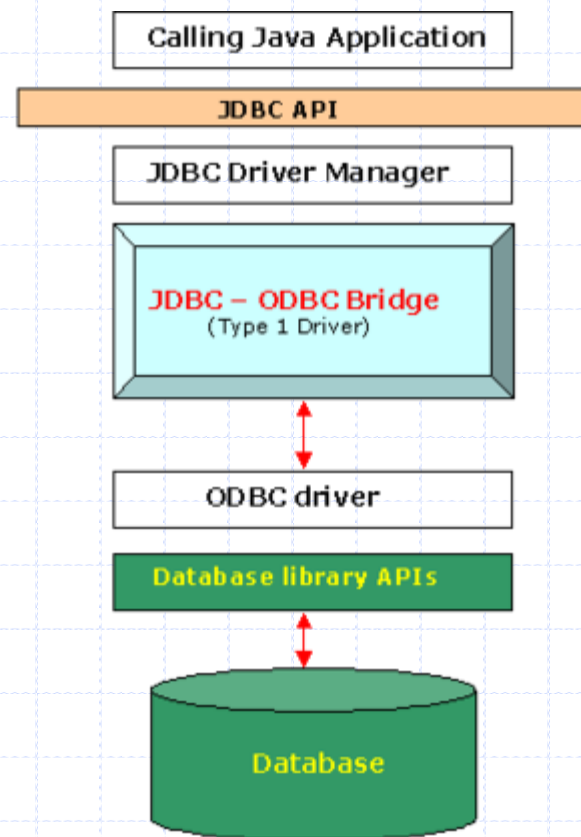
# JDBC驱动程序类型

## ◆ 四种类型

- Type1: JDBC-ODBC Bridge
- Type2: Native-API Partly Java
- Type3: Net-Protocol Fully Java
- Type4: Native-protocol Fully Java

# JDBC-ODBC桥

- ◆ 把JDBC API映射成ODBC API，借用现成的ODBC驱动程序
- ◆ `sun.jdbc.odbc.JdbcOdbcDriver`



# JDBC-ODBC桥

## ◆ 优点：

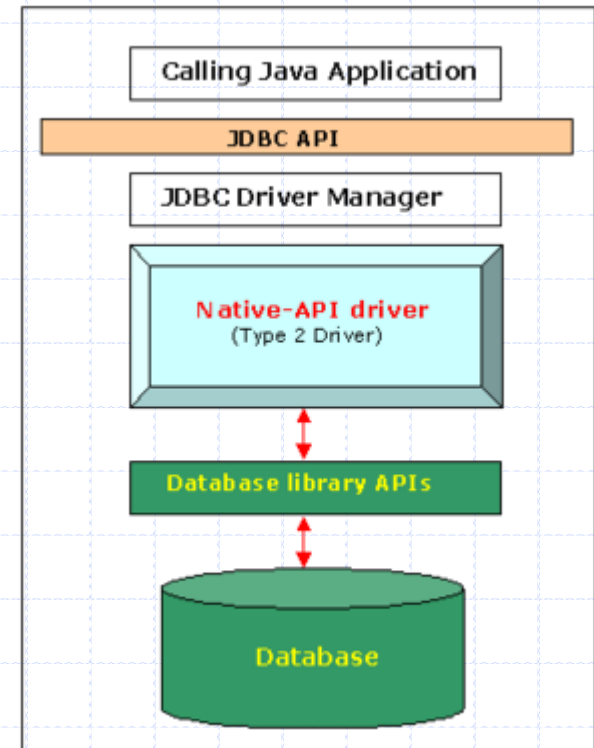
- 允许访问几乎任何数据库（特别是没有提供JDBC驱动程序的数据源）

## ◆ 缺点：

- 性能比单独的JDBC驱动慢；
- 丧失了Java的跨平台特性；不能用在Internet上（ODBC驱动必须安装在每个客户机上）；
- 不适合大型应用程序

# Native-API Partly Java

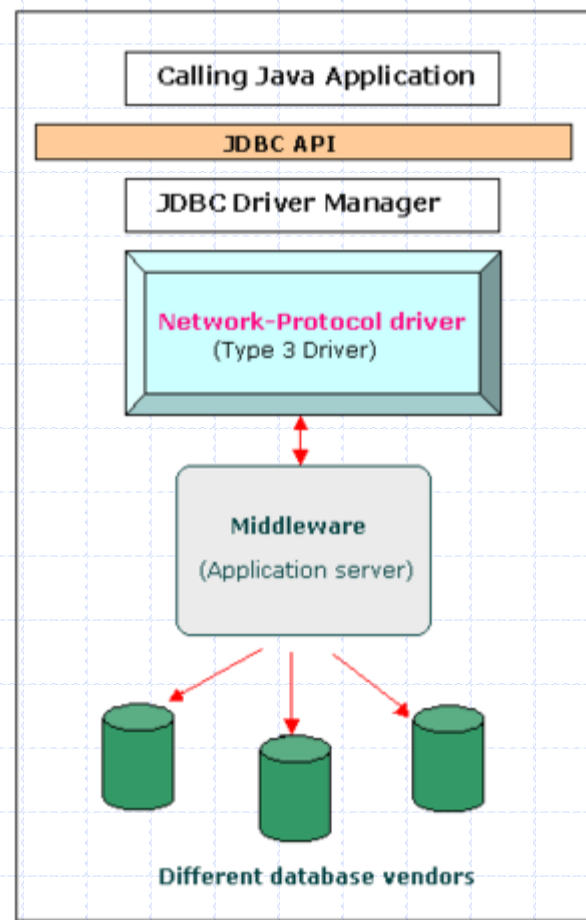
- ◆ 将JDBC API调用映射为针对数据库的客户端API
- ◆ 优点：
  - 性能比JDBC-ODBC桥好
- ◆ 缺点：
  - 需要为每台客户机上加载供应商数据库库文件/特殊协议（如Oracle SQLNet），不能用在Internet上





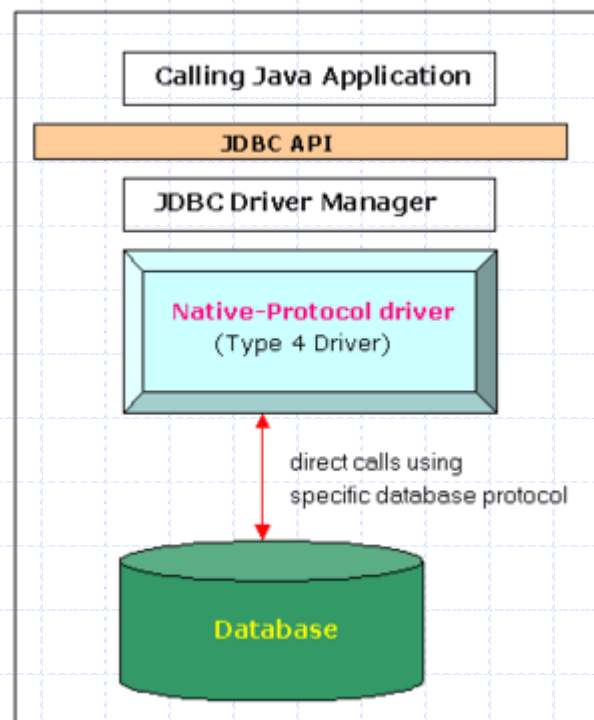
# Net-Protocol Fully Java

- ◆ 支持三层结构的JDBC访问方式，将JDBC调用转化为与DBMS无关的网络协议（如HTTP），再由中间层服务器转化为DBMS使用的协议
- ◆ 不需要在客户机上安装任何供应商数据库文件；最优的可移植性、性能和可缩放性；用在因特网应用程序中
- ◆ 数据来自后端服务器，查看记录集所需的时间长



# Native-Protocol Fully Java

- ◆ 纯JAVA：直接将JDBC调用转化为DBMS使用的网络协议，允许从客户机上直接调用DBMS服务器
- ◆ 比类型3快，用在因特网应用程序中
- ◆ 与平台无关



# DriverManager机制

## ◆ 使用DriverManager、Driver和Connection连接到数据库上：

- java.sql.DriverManager类：了解驱动的信息，维护驱动实现的列表，向应用程序提供一个驱动实现
- java.sql.Driver接口：DBMS供应商提供的驱动必须实现Driver接口，处理JDBC语句，把包含的SQL参数发送给数据库引擎
- java.sql.Connection接口：把一系列SQL语句发送给数据库，管理这些语句的提交或中断

# DriverManager机制

## ◆ 1. 注册驱动程序

- 隐式注册：加载数据库驱动程序类（把驱动加载到内存中），自动向DriverManager注册
  - ◆ `Class.forName("JDBCDriverName");`
- 显示注册：
  - ◆ `DriverManager.registerDriver (new JDBCDriverName());`

# DriverManager机制

## ◆ 2. 建立数据库连接:

- Connection  
con=DriverManager.getConnection(*URL, username, password*);
- //按照注册顺序, 找到第一个可以成功连接到给定URL的驱动程序, 返回一个Connection对象
- JDBC URL的语法:
  - ◆ jdbc:driver:databasename

## ◆ 3. 使用连接进行查询、插入、删除的操作

# DriverManager机制

## ◆ DriverManager机制的弊端：

- 是一个同步的类，一次只有一个线程可以运行
- 与数据库相关的连接信息都包含在类中，如果用户更换另一台计算机作数据库服务器，就需要重新修改URL变量、重新编译、部署；
- 用户的用户名、口令也包含在类中，丧失了安全性

# DataSource机制——JNDI

- ◆ 注册到JNDI，使用JNDI服务向程序隐藏了登录细节

<http://download.oracle.com/javase/jndi/tutorial/index.html>

- ◆ JNDI：Java命名和目录接口 (Java Naming and Directory Interface)，为开发人员提供了查找和访问各种命名和目录服务的通用、统一的方式。（中央注册中心，储存了各种对象、用户和应用的变量及其值，开发大型的分布式应用，使分布式的Java程序找到分布式的对象）

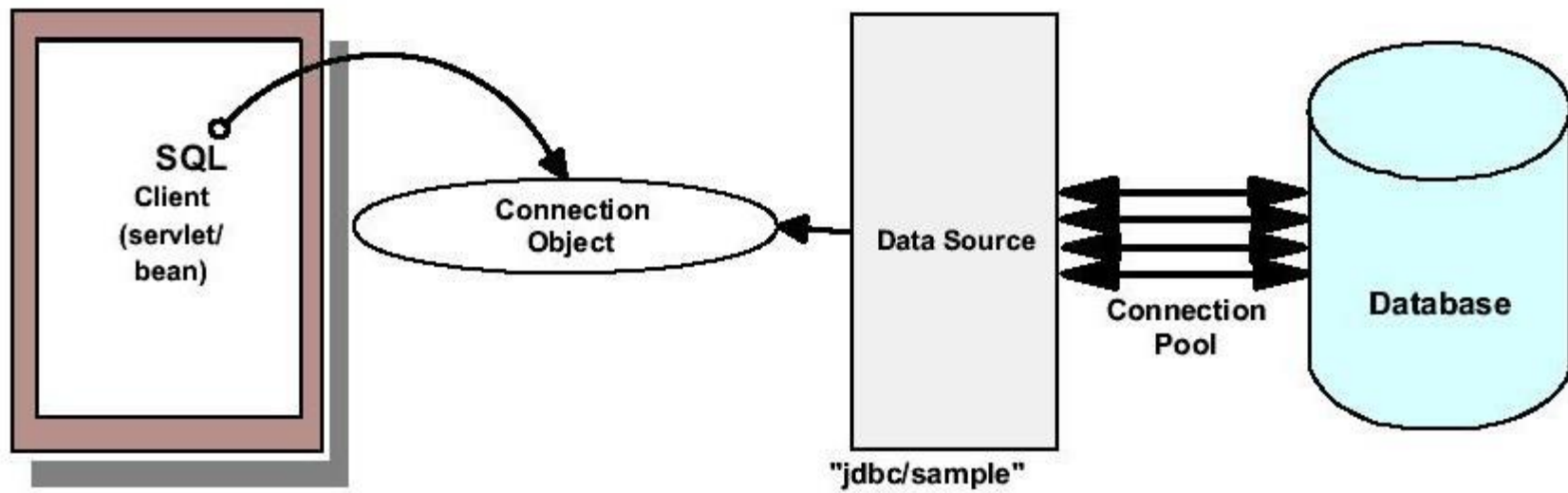
# JNDI

## ◆ 分布式应用程序：

- 通过RMI或CORBA向JNDI注册对象，其他任何客户机上的应用程序只需知道数据源对象在服务器JNDI中的逻辑名称，就可以通过RMI向服务器查询数据源，然后与数据库建立连接



# DataSource机制——连接池



# 数据库连接池

- ◆ 连接到数据库：需要通信、内存、授权等来创建连接，代价昂贵
- ◆ 对于访问站点的每一个客户机都建立了一个新的连接，费用太高
- ◆ 连接池：重用连接，而非建立新连接
  - 一组加载到内存中的数据库连接，以便重复使用
- ◆ 允许共享数据库连接，不是为每个客户分别提供单独的连接
- ◆ 借助连接池，对连接数量进行必要的定量限制，数据库才是最有效的。

# DataSource机制

- ◆ DataSource是JDBC Connection对象的一个工厂
- ◆ 允许使用已经在JNDI命名服务中注册的DataSource对象建立连接，由驱动程序供应商实现

# MySQL

◆ Lab-Tomcat-2. ppt

■ mysql-connector-java-5.\*.\*-bin.jar

# 配置Tomcat数据源

## ◆ Lab-Tomcat-2. ppt

- <http://localhost:8080/admin>
  - ◆ Data Source JNDI Name: **jdbc/abcdefds**
- 或context.xml文件（简单）

# JNDI API

- ◆ javax.naming 程序包
- ◆ JNDI API：定义了 InitialContext 类，有助于找到整个过程的起点

# DataSource机制

## ◆ 1. 查询数据源对象

- `Context ctx=new InitialContext();`
- `DataSource ods=(DataSource) ctx.lookup(DataSourceJNDIName);`

## ◆ 2. 获取数据库连接 (javax.sql.DataSource类的工厂方法)

- `ods.getConnection();`

## ◆ 3. 进行数据库操作 (与DriverManager机制相同)

- .....

# 与数据库交互(Statement)

- ◆ 与数据库的交互是由  
java.sql.Statement类完成的
- ◆ 利用Connection对象，创建一个  
Statement对象：
  - Statement stmt=con.createStatement();



# 与数据库的交互(ResultSet)

## ◆ 执行SQL语句

- `stmt.execute(“任何有效的SQL查询语句”);`

## ◆ 利用 `java.sql.ResultSet` 类（结果集），返回查询结果

- `ResultSet rs=stmt.getResultSet();`

## ◆ 快捷方法

- `ResultSet rs=stmt.executeQuery(“SQL查询语句”);`

# ResultSet

- ◆ 把数据库中的数据映射为Java对象的实例；
- ◆ 参考数据库驱动程序文档

# ResultSet

- ◆ **ResultSet对象拥有一个指向当前数据行的指针，开始，这个光标定位到第一个数据行之前的位置**
- ◆ **读取查询结果**
  - **next() 方法**
    - ◆ 把光标移动到下一行，当ResultSet对象没有更多的数据行时， next() 方法返回false

# ResultSet

- `getString(String ColumnName)` 方法
  - ◆ 读取ResultSet对象当前行中指定列名的字符串值
- `absolute(int row)` 方法
  - ◆ 把光标移到ResultSet对象的指定行上
- `beforeFirst()` 方法
  - ◆ 把光标移到ResultSet对象的起始位置
- `isAfterLast()` 方法
  - ◆ 光标是否位于ResultSet对象的最后一行之后

# ResultSet

- **isBeforeFirst() 方法**
  - ◆ 光标是否位于ResultSet对象的第一行之前
- **isFirst() 方法**
  - ◆ 光标是否位于ResultSet对象的第一行
- **isLast() 方法**
  - ◆ 光标是否位于ResultSet对象的最后一行
- **refreshRow() 方法**
  - ◆ 使用数据库中的最新值刷新当前行

# close()

- ◆ 完成查询之后，依次释放ResultSet对象，Statement对象，Connection对象
  - 调用close()方法实现

# XXXServlet.java

- ◆ 查询数据库，返回结果的Servlet
- ◆ 把只需执行一次的代码放在初始化阶段

```
Context ctx; DataSource ds;  
Statement stmt; ResultSet rs;  
public void init() {  
    ctx=new InitialContext();  
    ds= (DataSource)  
    ctx.lookup("java:comp/env/jdbc/abcdefds")  
    ;  
}
```

Tomcat数据  
源固定格式

## ◆ 查询数据库，结果返回

```
public void Service(HttpServletRequest req,  
HttpServletResponse res) throws  
IOException{
```

```
    Connection con= ds.getConnection();
```


```
    stmt=con.createStatement();
```

```
    rs=stmt.executeQuery(“SELECT * FROM EMPLOYEE”);
```

```
    PrintWriter out = resp.getWriter();
```

```
    .....
```





```
while(rs.next()) {  
    out.println (rs.getString("ID")+"- "+  
        rs.getString("NAME")+"- "  
        rs.getString("LOCATION")+"<P>");  
}  
rs.close();  
stmt.close();  
con.close();  
}
```

# 异常处理

```
public void Init() {  
    try{  
        .....  
    } catch (Exception E) {  
        System.out.println("Initial Error: "+E);  
    }  
}
```

# 异常处理

```
public void Service(.....)throws
IOException {
    try{
        .....
        rs.close();
        stmt.close();
    } catch (Exception E) {
        System.out.println("Initial Error: "+E);
    }
}
```

# 异常处理

```
    } finally{  
        if(rs!=null){  
            try{  
                rs.close();  
            } catch (Exception ignore) {};  
        }  
        if(stmt!=null){  
            try{  
                stmt.close();  
            } catch (Exception ignore) {};  
        }  
        if(con!=null){  
            try{  
                con.close();  
            } catch (Exception ignore) {};  
        }  
    }  
}
```

# web.xml

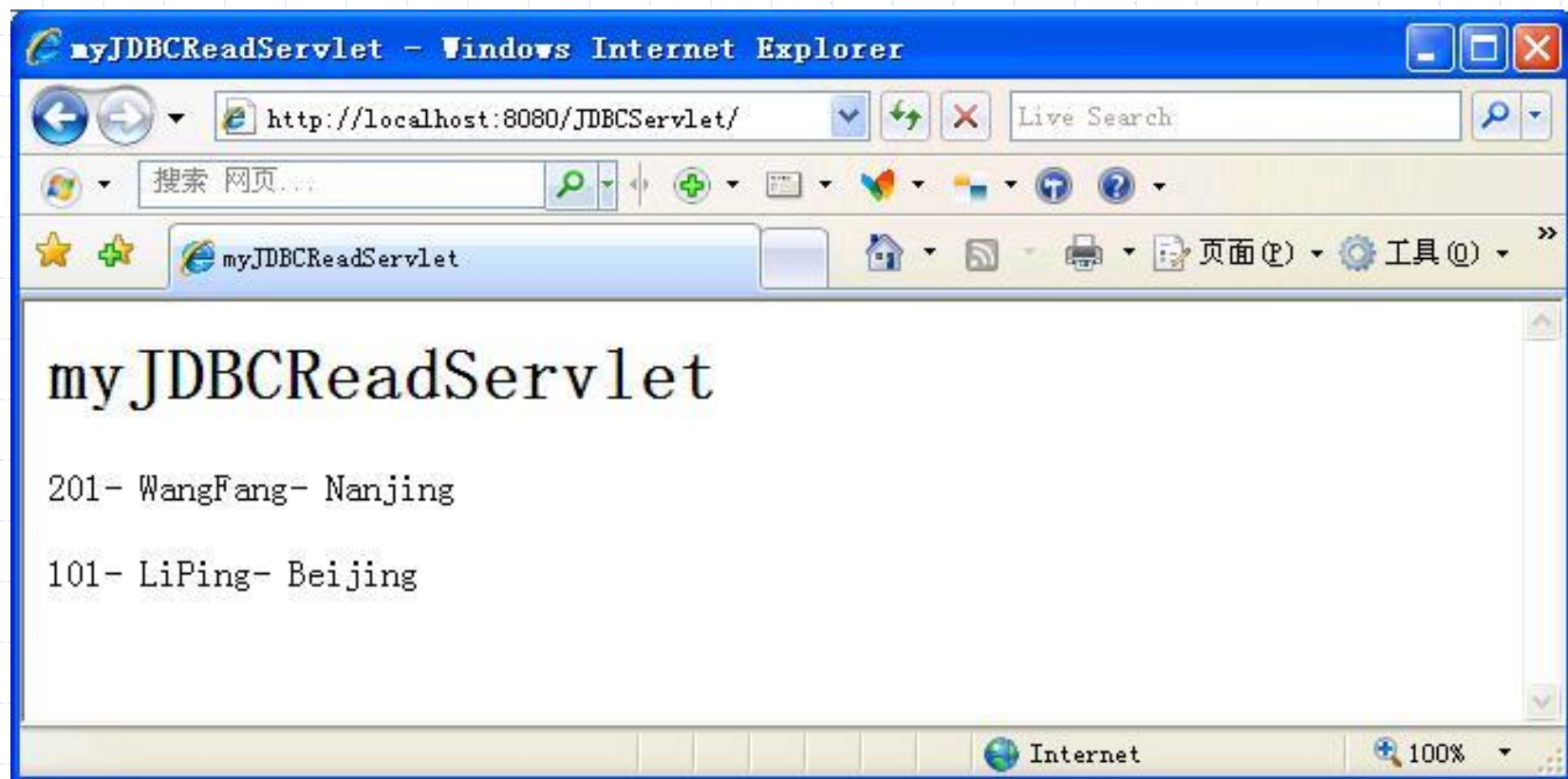
```
<web-app>
```

```
  <welcome-file-list>
```

```
    <welcome-file>/myJDBCReadServlet</welcome-file>
```

```
  </welcome-file-list>
```

```
</web-app>
```



# 使用JDBC更新数据库

- ◆使用JDBC数据源，建立数据库连接
- ◆执行SQL语句：INSERT, UPDATE, DELETE
  - Statement stmt=con.createStatement();
  - stmt.**executeUpdate**(“SQL语句”);
- ◆释放连接

# 增加数据库表

## ◆ Statement

```
stmt1=con.createStatement();
```

```
◆ stmt1.execute("create table  
myTable(id integer, name char(25))");
```

```
◆ stmt1.close();
```



# 删除数据库表

◆ Statement

```
stmt2=con.createStatement();
```

◆ stmt2.**execute**("drop table  
myTable");

◆ stmt2.close();

# 宏语句/准备语句 (PreparedStatement)

## ◆ 示例:

- Stmt.executeUpdate("INSERT INTO employee VALUES ('Benjamin', 'France', 55)");
- Stmt.executeUpdate("INSERT INTO employee VALUES ('Rob', 'Illinois', 56)");

## ◆ 对于每个语句，数据库和JDBC驱动程序必须把它们映射到底层数据库能理解的操作；

- 如果能够压缩或合并这个步骤，性能可以得到很大的提高——宏语句

## ◆ 防止SQL注入

- 如果采用“预编译”，执行阶段只是把输入串作为数据处理，而不需要对SQL语句进行解析——准备语句

# 宏语句

- ◆ 使用从Statement接口扩展而来的PreparedStatement接口；
- ◆ 创建PreparedStatement对象，以合并更新操作为例：
  - `PreparedStatement pstmt=myConn.  
prepareStatement(“INSERT INTO employee  
VALUES (?, ?, ?)”);`
  - `//设置通配符`

# 宏语句

## ◆ 执行更新1：

- `pstmt.setString(1, "Benjamin");`
- `//set`方法第一个参数：要代替的通配符的索引（从1开始）；第二个参数：要插入的值；
- `pstmt.setString(2, "France");`
- `pstmt.setInt(3, 55);`
- `int opNum= pstmt.executeUpdate();`
- `//返回row count`

# 宏语句

## ◆ 执行更新2 :

- `pstmt.setString(1, "Rob");`
- `pstmt.setString(2, "Illinois");`
- `pstmt.setInt(3, 56);`
- `opNum= pstmt.executeUpdate();`

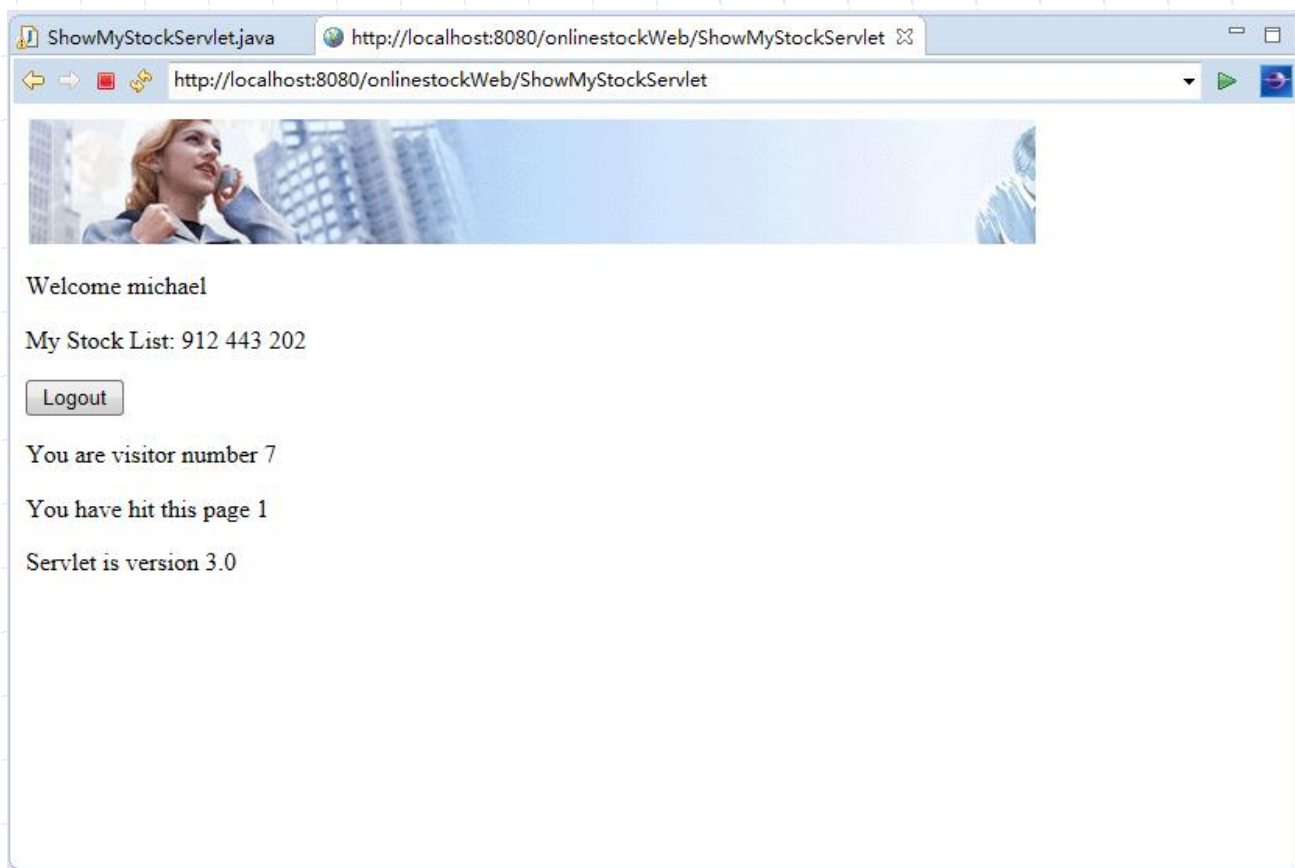
## ◆ 尽量使用宏语句，使数据库能够把SQL编译成只需提供不同的参数即可重复执行的语句，提高执行的速度，因为数据库不需要重新编译SQL

# 批量更新

## ◆ 所有的更新通过一个数据库调用完成

- `pstmt.setString(1, "Benjamin");`
- `pstmt.setString(2, "France");`
- `pstmt.setInt(3, 55);`
- `pstmt.addBatch();`
- `pstmt.setString(1, "Rob");`
- `pstmt.setString(2, "Illinois");`
- `pstmt.setInt(3, 56);`
- `pstmt.addBatch();`
- `int[] updateCounts= pstmt.executeBatch();`

# 示例: ShowMyStockServlet



# ShowMyStockServlet.java

```
public void init() {  
    InitialContext jndiContext = null;  
    Properties properties = new Properties();  
    properties.put(javax.naming.Context.PROVIDER_URL, "jnp:///");  
    properties.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,  
        "org.apache.naming.java.javaURLContextFactory");  
    try {  
        jndiContext = new InitialContext(properties);  
        datasource = (DataSource) jndiContext  
            .lookup("java:comp/env/jdbc/onlinestock");  
    } catch (NamingException e) {  
        e.printStackTrace();  
    }  
}
```



# Service()

.....

```
Connection connection = null;
PreparedStatement stmt = null;
ResultSet result = null;
ArrayList list = new ArrayList();
try {
    connection = datasource.getConnection();
} catch (SQLException e) {
    e.printStackTrace();
}
```

```
try {  
    stmt = connection.prepareStatement("select stockid  
        from mystock where userid=?");  
    stmt.setString(1, (String)  
        req.getAttribute("login"));  
    result = stmt.executeQuery();  
    while (result.next()) {  
        Stock stock = new Stock();  
        stock.setId(result.getInt("stockid"));  
        .....  
        list.add(stock);  
    }  
} catch (SQLException e) {  
    e.printStackTrace();  
}
```

.....

```
out.println("My Stock List: ");  
for (int i = 0; i < list.size(); i++) {  
    Stock stock = (Stock) list.get(i);  
    out.println(stock.getId());  
    .....
```

```
}
```

.....

# SQLException

## ◆ 错误处理

- try {
  - ◆ //do JDBC work here
- } catch (SQLException sqle) {
  - ◆ System.out.println("JDBC exception encountered:"+sqle);
  - ◆ System.out.println("SQL state string:"+sqle.getSQLState());
  - ◆ System.out.println("Database specific error code:"+sqle.getErrorCode());
- }

# SQLWarning

## ◆ 处理警告

- try{
  - ◆ //do JDBC work here
  - ◆ SQLWarning sqlw=stmt.getWarnings();
  - ◆ if(sqlw!=null) {
    - System.out.println("SQL warning encountered:"+sqlw);
    - System.out.println("SQL state string:"+sqlw.getSQLState());
    - System.out.println("Database specific error code:"+sqlw.getErrorCode());
  - ◆ }
- } catch (SQLException sqle) {
  - ◆ ...
- }finally{ stmt.close();}

# 元数据

◆ **ResultSetMetaData**: 结果集的结构信息, 包括列名、列的数量等

- **ResultSetMetaData**  
rsmd=rs. getMetaData ();
- rsmd. getColumnCount ();
- rsmd. getColumnName (index);

# BLOB/CLOB

## ◆ 二进制大对象/字符大对象

### ◆ 访问:

- `java.sql.Blob myBlob=rs.getBlob("blobcolumn");`
- `java.sql.Clob myClob=rs.getClob("clobcolumn");`

### ◆ 显示:

- `java.io.InputStream readis=  
myBlob.getBinaryStream();`
- `java.io.InputStream readClobis=  
myClob.getAsciiStream();`

# 事务

## ◆ 保证一系列数据库操作能够准确的完成

- 如：确保——当我们在库存清单中删除某个项时，同时也会从购买者的账号中扣除相应的货款
- 当用户对拍卖会上的给定物品进行投标时，事务防止——多个同时进行的投标用户在同一拍卖场中胜出，或对某项给定的物品投出完全相同的价格

## ◆ 除非事务中的所有操作都成功，否则事务就不会完成



# 事务的4个特性（ACID）

◆ 应用服务器和数据库协同工作，确保事务的四个特性：

- Atomicity（原子性）
- Consistency（一致性）
- Isolation（隔离性）
- Durability（持久性）

◆ 以银行事务为例：

# Atomicity/ Consistency

## ◆ 原子性

- 资金转账：保证不会从一个账号减掉了钱，而没有在另一个账号加钱

## ◆ 一致性

- 从一个账户减去部分钱数，不应当影响系统中的任何其他账户

# Isolation/ Durability

## ◆ 隔离性

- 资金转账需要把资金从一个账号划走，同时把资金加到另一个账号上，应该与依次进行这两个行为所得到的结果一致
- 在操作比较复杂的情况下，隔离性变得十分重要

## ◆ 持久性

- 转账记录应该存储在磁盘上

# Java事务类型

◆ JDBC事务、JTA (Java Transaction API)  
事务、容器事务

# JDBC 事务

- ◆ 用 `Connection` 对象控制
- ◆ 提供了两种事务模式
  - 自动提交
  - 手工提交
- ◆ `java.sql.Connection` 提供了以下控制事务的方法：
  - `public void setAutoCommit(boolean)`
  - `public boolean getAutoCommit()`
  - `public void commit()`
  - `public void rollback()`

◆使用 JDBC 事务界定时，可以将多个 SQL 语句结合到一个事务中

◆缺点：

- 事务的范围局限于一个数据库连接，一个 JDBC 事务不能跨越多个数据库

# JDBC Transaction实例


◆ To ensure that the order is processed in its entirety, the call to **buyBooks** is wrapped in a single transaction.

◆ 购买购物车中的书（多项）

# buyBooks


```
◆ public void buyBooks(ShoppingCart cart) throws  
  OrderException{  
    ■ Collection items = cart.getItems();  
    ■ Iterator i = items.iterator();  
    ■ try {  
      ◆ getConnection();  
      ◆ //关闭自动提交, 此时, 除非通过调用commit()显式地告诉它提交  
        语句, 否则无法提交SQL语句 (即, 数据库将不会被持久地更新)。  
      ◆ con.setAutoCommit(false);  
      ◆ while (i.hasNext()) {  
        ■ ShoppingCartItem sci = (ShoppingCartItem)i.next();  
        ■ BookDetails bd = (BookDetails)sci.getItem();  
        ■ String id = bd.getBookId();  
        ■ int quantity = sci.getQuantity();  
        ■ buyBook(id, quantity);  
      ◆ }  
    }  
  }
```





- ◆ //手动提交
- ◆ **con. commit();**
- ◆ **con. setAutoCommit(true);**
- ◆ **releaseConnection();**
- **} catch (Exception ex) {**
  - ◆ **try {**
    - //在提交之前, 调用rollback()回滚事务, 并恢复最近的提交值 (在尝试更新之前)。
    - **con.rollback();**
    - **releaseConnection();**
    - **throw new OrderException("Transaction failed: " + ex.getMessage());**
  - ◆ **} catch (SQLException sqx) {**
    - **releaseConnection();**
    - **throw new OrderException("Rollback failed: " + sqx.getMessage());**
  - ◆ **}**
- **}**
- ◆ **}**

```
◆ public void buyBook(String bookId, int quantity)
    throws OrderException {
    ■ try {
        ◆ String selectStatement = "select * " + "from books where id
          = ? ";
        ◆ PreparedStatement prepStmt =
          con.prepareStatement(selectStatement);
        ◆ prepStmt.setString(1, bookId);
        ◆ ResultSet rs = prepStmt.executeQuery();
        ◆ if (rs.next()) {
            ■ int inventory = rs.getInt(9);
            ■ prepStmt.close();
            ■ if ((inventory - quantity) >= 0) {
                ■ String updateStatement = "update books set inventory =
                  inventory - ? where id = ?";
                ■ prepStmt = con.prepareStatement(updateStatement);
                ■ prepStmt.setInt(1, quantity);
                ■ prepStmt.setString(2, bookId);
                ■ prepStmt.executeUpdate();
                ■ prepStmt.close();
            }
            ■ } else
```



- throw new OrderException("Not enough of " + bookId + " in stock to complete order.");
- }

- } catch (Exception ex) {
  - ◆ throw new OrderException("Couldn't purchase book: " + bookId + ex.getMessage());}

- ◆ }

# Java Transaction API

- ◆ The Java EE architecture provides a **default auto commit** to handle transaction commits and rollbacks.
- ◆ An **auto commit** means that any other applications that are viewing data will see the updated data after each database read or write operation.
- ◆ However, if your application performs **two separate database access operations** that depend on each other, you will want to use the **JTA API** to demarcate where the entire transaction, including both operations, begins, rolls back, and commits.

# JTA事务

- ◆ JTA允许应用程序执行分布式事务处理 - 在两个或多个网络计算机资源上访问并且更新数据，这些数据可以**分布在多个数据库上**
- ◆ 两阶段提交：事务管理器和资源管理器之间使用的协议是XA；
  - XA：资源和事务管理器之间的标准化接口

◆ 需要有一个实现

javax.sql.XADataSource 、  
javax.sql.XAConnection 和  
javax.sql.XAResource 接口的 JDBC 驱动程序

◆ XA 连接参与了 JTA 事务, XA 连接不支持 JDBC 的自动提交功能

# JTA API

- ◆ `javax.transaction.UserTransaction` 对象的 `begin()` 方法、`commit()` 方法和 `rollback()` 方法
- ◆ 应用程序应该使用 `UserTransaction.begin()`、`UserTransaction.commit()` 和 `UserTransaction.rollback()`

# 使用JTA

## ◆ 开发人员声明事务的开始和提交

- 1. 建立事务
- 2. 启动事务
- 3. 定位数据源
- 4. 建立数据库连接
- 5. 执行与资源有关的操作
- 6. 关闭连接
- 7. 完成事务



# 建立事务

## ◆ 找到一个新的初始上下文环境

- `Context ctx=new InitialContext();`

## ◆ 在位于JNDI中的Context对象中找出UserTransaction对象

- `UserTransaction tx= (UserTransaction) ctx.lookup("javax.transaction. UserTransaction");`

# 启动事务

- ◆ 调用UserTransaction对象的begin()方法
  - tx.begin();
  - //针对数据库的任何操作均处于这个事务的范围之内

# 定位数据源、建立数据库连接

## ◆ 定位支持事务的数据源

- `javax.sql.DataSource ds=  
    (javax.sql.DataSource)  
    ctx.lookup("MyDemoJDBCDataSource");`

## ◆ 建立数据库连接

- `javax.sql.Connection con=  
    ds.getConnection();`

# 执行与资源有关的操作

- ◆ 使用Connection对象，生成一个针对数据库的Statement对象，执行多个SQL语句
  - `Statement stmt=con.createStatement();`
- ◆ 作为一个事务单位，这些SQL或者都成功的执行和提交，或者都回滚到数据库的初始状态
  - `Stmt.executeUpdate("INSERT INTO employee VALUES ('Benjamin', 'France', 55)");`
  - `Stmt.executeUpdate("INSERT INTO employee VALUES ('Rob', 'Illinois', 56)");`

# 关闭连接

## ◆调用close()方法

- `stmt.close()`;
- `con.close()`;

◆此时，`Connection`对象暂时处于一种中间过渡状态，数据库连接不会返回到连接池，极大的影响了服务器的性能，应确保尽可能快的提交或回滚事务

# 完成事务

- ◆ 如果事务的全部操作都成功的执行，提交事务，否则，应回滚

```
try{  
    ...Step 1 through 5 here...  
    tx.commit(); //提交  
} catch (Exception txe) {  
    System.out.println("Servlet  
error:" + txe);  
}
```

# 完成事务

```
try{
    tx.rollback(); //回滚
}catch(javax.transaction.SystemException se) {}
}finally{
    try{
        if(rs!=null)
            rs.close();
        if(stmt!=null)
            stmt.close();
        if(con!=null)
            if(!con.isClosed())
                con.close();
    }catch(SQLException sqle) {}
}
```

# 容器事务

- ◆ 是J2EE应用服务器提供的，容器事务大多是**基于JTA**完成
- ◆ 与编码实现JTA事务管理相比，可以通过EJB容器提供的容器事务管理机制（CMT）完成同一个功能
  - 可以简单的指定将哪个方法加入事务，一旦指定，容器将负责事务管理任务



# 事务的隔离级

- ◆ 当两个或多个操作（或事务）同时作用于同一个数据库时，会引起访问的冲突或数据的不一致
  - 例如：一个线程（或事务）可能正在读取数据库中的某个数据，而另一个线程（或事务）却要对同一个数据进行更新
- ◆ 事务的隔离级：指定数据库如何处理各种冲突情况

# 事务的隔离级

- ◆ JDBC规范：5个级别
- ◆ 阅读JDBC驱动程序和数据库资料，了解数据库在每种隔离级别中具体都采用哪些处理措施

# 事务的隔离级

- ◆ **TRANSACTION\_NONE**: 对事务和数据不进行任何隔离限制
- ◆ **TRANSACTION\_READ\_UNCOMMITTED**: 允许事务读取另一个事务未提交的数据(脏读)
  - 事务1把某个账号的余额从1000元改为500元, 在事务1没有提交前, 事务2又读到这个帐号的余额, 它读到的数据为500元

# 事务的隔离级

◆ TRANSACTION\_READ\_COMMITTED: 保证所有读取的数据都是已经提交的数据，不能保证可重复读

- 事务1读到账号余额为1000元，在事务1提交之前，事务2把帐号的余额从1000元改为500元，并提交更新，当事务1再次读这个帐号的余额时，它读到的数据为500元——同一列在事务1中被读取两次，返回结果不同（不可重复读）
- Oracle: 默认的隔离级别
- 最常用

# 事务的隔离级

- ◆ **TRANSACTION\_REPEATABLE\_READ: 防止脏读和不可重复读**
  - 在表中增加或删除一行，仍有副作用——幻读
  - 代价昂贵，很少使用
- ◆ **TRANSACTION\_SERIALIZABLE: 防止脏读、不可重复读和幻读，提供最高级别的保护，性能最低**

# 事务的隔离级

- ◆ 当多个客户端共享数据时：设置事务隔离级别
- ◆ 修改事务隔离级：
  - 从连接池中获取数据库连接conn;
  - `conn.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);`

# 事务的隔离级

- ◆ 许多数据库并不支持所有的隔离级，每个数据库也可能使用不同的锁定算法——选择隔离级时，查阅数据库的资料
- ◆ 隔离级别越高，应用程序执行的速度也就越慢（由于用于锁定的资源耗费增加了，而用户间的并发操作减少了）

# JDBC最佳实践

- ◆ 尽可能使查询更灵活更准确：SQL语句
- ◆ 调整数据库的参数设置：适当的数据库缓冲策略等
- ◆ 把初始化代码放到init()方法中：只执行一次
- ◆ 使用批量更新：一次连接完成所有动作
- ◆ 尽可能做现场更新：使用UPDATE而不使用INSERT, DELETE, REMOVE



# JDBC最佳实践

- ◆ 使用适当的方法取得JDBC连接：  
DataSource
- ◆ 适当的释放JDBC资源：finally代码块中  
释放connection资源
- ◆ 不要闲置JDBC连接：尽快释放
- ◆ 尽快的提交或回滚事务
- ◆ 适当设置连接池的容量

# JDBC最佳实践

- ◆ **不要让事务涵盖用户的输入：事务耗费数据库和应用服务器的资源，应避免使用长时间的事务**
  - 例如：一个事务从加载浏览器页面开始，用户单击submit按钮才提交——耗费资源！
- ◆ **尽可能利用数据库的功能：数据库触发器等**