# 20.事务

## 20.1.事务的概念

事务指逻辑上的一组操作，组成这组操作的各个单元，要么全部成功，要么全部不成功。

例如：A——B转帐，对应于如下两条sql语句

update account set money=money-100 where name=‘a’;

update account set money=money+100 where name=‘b’;

在这个例子中，我们要保证这两条sql要么一起成功，要么一起失败，不允许一部分成功一部分失败，这就要靠数据库的事务来实现了。

## 20.2.管理事务

**数据库默认的事务**

数据库默认支持事务的，但是数据库默认的事务是一条sql语句独占一个事务，这种模式意义不大。

**手动控制事务**

**sql控制事务**

start transaction;

开启事务，在这条语句之后的所有的sql将处在同一事务中，要么同时完成要么同时不完成。

事务中的sql在执行时，并没有真正修改数据库中的数据。

commit;

提交事务，将整个事务对数据库的影响一起发生。

rollback;

回滚事务，将这个事务对数据库的影响取消掉。

Sql代码示例：

|  |
| --- |
| 在添加事务之前：  create database tdb;  use tdb;  create table account (  id int primary key auto\_increment,  name varchar(255),  money double  );  insert into account values (null,'a',1000.0);  insert into account values (null,'b',1000.0);  update account set money = money-100 where name='a';  update account set money = money+100 where name='b';  select \* from account; |
| 在添加事务之后：  start transaction;  update account set money = money-100 where name='a';  update account set money = money+100 where name='b';  commit;    start transaction;  update account set money = money-100 where name='a';  update account set money = money+100 where name='b';  rollback; |

**JDBC中控制事务**

当Jdbc程序向数据库获得一个Connection对象时，默认情况下这个Connection对象会自动向数据库提交在它上面发送的SQL语句。若想关闭这种默认提交方式，让多条SQL在一个事务中执行，可使用下列语句：

conn.setAutoCommit(false);

关闭自动连接后,conn将不会帮我们提交事务,在这个连接上执行的所有sql语句将处在同一事务中,需要我们是手动的进行提交或回滚

conn.commit();

提交事务

conn.rollback();

回滚事务

也可以设置回滚点回滚部分事务。

avePoint sp = conn.setSavePoint();

conn.rollback(sp);

注意：回到回滚点后，回滚点之前的代码虽然没被回滚但是也没提交呢，如果想起作用还要做commit操作。

代码示例：

|  |
| --- |
| 创建java项目：day18，并创建lib文件夹，将mysql驱动包拷贝到lib目录下，并build path，在src下创建包：cn.tedu.jdbc.trans，创建类Demo1，添加如下代码：  **public** **static** **void** main(String[] args) {  Connection conn = **null**;  PreparedStatement ps = **null**;  ResultSet rs = **null**;  Savepoint sp = **null**;  **try**{  Class.*forName*("com.mysql.jdbc.Driver");  conn = DriverManager.*getConnection*("jdbc:mysql:///tdb", "root", "root");    //开启事务  conn.setAutoCommit(**false**);    ps = conn.prepareStatement("update account set money = money - ? where name = ?");  ps.setDouble(1, 100.0);  ps.setString(2, "a");  ps.executeUpdate();    ps = conn.prepareStatement("update account set money = money + ? where name = ?");  ps.setDouble(1, 100.0);  ps.setString(2, "b");  ps.executeUpdate();  //设置回滚点  sp = conn.setSavepoint();    ps = conn.prepareStatement("update account set money = money - ? where name = ?");  ps.setDouble(1, 100.0);  ps.setString(2, "a");  ps.executeUpdate();    **int** i = 1/0;    ps = conn.prepareStatement("update account set money = money + ? where name = ?");  ps.setDouble(1, 100.0);  ps.setString(2, "b");  ps.executeUpdate();    //提交事务  conn.commit();  }**catch** (Exception e) {  //回滚事务  **if**(conn != **null**){  **try** {  **if**(sp!=**null**){//走过了回滚点  conn.rollback(sp);  conn.commit();  }**else**{//没有到回滚点  conn.rollback();  }  } **catch** (SQLException e1) {  e1.printStackTrace();  }  }  e.printStackTrace();  }**finally**{  **if**(rs != **null**){  **try** {  rs.close();  } **catch** (SQLException e) {  e.printStackTrace();  } **finally** {  rs = **null**;  }  }  **if**(ps != **null**){  **try** {  ps.close();  } **catch** (SQLException e) {  e.printStackTrace();  } **finally** {  ps = **null**;  }  }  **if**(conn != **null**){  **try** {  conn.close();  } **catch** (SQLException e) {  e.printStackTrace();  } **finally** {  conn = **null**;  }  }  }  } |

## 20.3.事务的四大特性

事务的四大特性是事务本身具有的特点。简称ACID。

**原子性（Atomicity）**

原子性是指事务是一个不可分割的工作单位，事务中的操作要么都发生，要么都不发生。

**一致性（Consistency）**

事务前后数据的完整性必须保持一致。**隔离性（Isolation）**

事务的隔离性是指多个用户并发访问数据库时，一个用户的事务不能被其它用户的事务所干扰，多个并发事务之间数据要相互隔离。

**持久性（Durability）**

持久性是指一个事务一旦被提交，它对数据库中数据的改变就是永久性的，接下来即使数据库发生故障也不应该对其有任何影响。

## 20.4.隔离性

### 20.4.1.数据库隔离性分析

数据库的其他三大特性数据库可以帮我们保证，而隔离性我们需要再讨论。

我们知道数据库的隔离性问题本质上就是多线程并发安全性问题。

可以用锁来解决多线成并发安全问题，但是如果用了锁，必然会造成程序的性能大大的下降.对于数据库这种高并发要求的程序来说这是不可接受的.

我们可以具体分析下隔离性产生的细节：

如果两个线程并发修改，必然产生多线程并发安全问题，必须隔离开

如果两个线程并发查询，必然没有问题，不需要隔离

如果一个线程修改，一个线程查询，在不同的应用场景下有可能有问题，有可能没问题。

### 20.4.2.隔离性可能造成的问题

**脏读：**

打开两个mysql客户端，都执行以下语句。

set session transaction isolation level read uncommitted;

一个事务读取到另一个事务未提交的数据：a买鞋，b卖鞋

----------------------------

a 1000

b 1000

----------------------------

客户端a:

start transaction;

update account set money = money-100 where name='a';

update account set money = money+100 where name='b';

-----------------------------

客户端b:

start transaction;

select \* from account;

a 900

b 1100

commit;

-----------------------------

客户端a:

rollback;

-----------------------------

客户端b:

start transaction;

select \* from account;

a 1000

b 1000

commit;

-----------------------------

**不可重复读：**

一个事务多次读取数据库中的同一条记录，多次查询的结果不同(一个事务读取到另一个事务已经提交的数据) a:银行账户，W:银行工作人员（领导让W统计a的账户情况）

------------------------------

活期 定期 固定资产

a 1000 1000 1000

------------------------------

W:

start transaction;

select 活期 from account where name='a'; -- 活期存款:1000W元

select 定期 from account where name = 'a'; -- 定期存款:1000W元

select 固定 from account where name = 'a'; -- 固定资产:1000W元

---------------------------

a:

start transaction;

update account set 活期=活期-1000 where name= 'a';

commit;

---------------------------

select 活期+定期+固定 from account where name='a'; ---总资产:2000W元

**虚读(幻读)**

有可能出现,有可能不出现，概率非常低：一个事务多次查询整表数据，多次查询时，由于有其他事务增删数据， 造成的查询结果不同(一个事务读取到另一个事务已经提交的数据)

------------------------------

a 1000

b 2000

------------------------------

工作人员d:

start transaction;

select sum(money) from account; --- 总存款3000元

select count(\*) from account; --- 总账户数2个

-----------------

c:

start transaction;

insert into account values (null,'c',3000);

commit;

-----------------

select avg(mone) from account; --- 平均每个账户:2000元

## 20.5.数据库的隔离级别

数据库设计者在设计数据库时到底该防止哪些问题呢？防止的问题越多性能越低，防止的问题越少，则安全性越差。

到底该防止哪些问题应该由数据库使用者根据具体的业务场景来决定，所以数据库的设计者并没有把放置哪类问题写死，而是提供了如下选项：

**数据库的四大隔离级别:**

**read uncommitted;**

不做任何隔离，可能造成脏读、不可重复度、虚读(幻读)问题

**read committed;**

可以防止脏读，但是不能防止不可重复度、虚读(幻读)问题

**repeatable Read;**

可以防止脏读、不可重复度，但是不能防止虚读(幻读)问题

**serializable;**

可以防止所有隔离性的问题，但是数据库就被设计为了串行化的数据库，性能很低

从安全性上考虑:

Serializable > Repeatable Read > Read Committed > Read uncommitted

从性能上考虑:

Read uncommitted > Read committed > Repeatable Read > Serializable

我们作为数据库的使用者，综合考虑安全性和性能，从四大隔离级别中选择一个在可以防止想要防止的问题的隔离级别中性能最高的一个。

其中Serializable性能太低用的不多，Read uncommitted安全性太低用的也不多，我们通常从Repeatable Read和Read committed中选择一个。

如果需要防止不可重复读选择Repeatable Read，如果不需要防止选择Read committed

mysql数据库默认的隔离级别就是Repeatable Read

Oracle数据库默认的隔离级别是Read committed

## 20.6.操作数据库的隔离级别

查询数据库的隔离级别：

select @@tx\_isolation;

修改数据库的隔离级别：

set [session/global] transaction isolation level xxxxxx;

不写默认就是session，修改的是当前客户端和服务器交互时是使用的隔离级别，并不会影响其他客户端的隔离级别

如果写成global，修改的是数据库默认的隔离级别(即新开客户端时，默认的隔离级别)，并不会修改当前客户端和已经开启的客户端的隔离级别

set global transaction isolation level serializable;

## 20.7.数据库中的锁

**共享锁**

共享锁和共享锁可以共存，共享锁和排他锁不能共存.在非Serializable隔离级别下做查询不加任何锁，在Serializable隔离级别下做查询加共享锁。

案例演示：打开两个mysql客户端，将隔离级别都设置为Serializable级别，

set session transaction isolation level Serializable;--设置后查询加了共享锁

分别在两个客户端中查询：

start transaction;

select \* from account;--都能查询出数据，说明共享锁可以共存。

**排他锁**

排他锁和共享锁不能共存，排他锁和排他锁也不能共存，在任何隔离级别下做增删改都加排他锁。

在7.1的基础上，在其中一个客户端执行修改操作，将一个客户端的共享锁升级为排他锁：

两个客户端都执行：

start transaction;

select \* from account;

-----------------

一个客户端执行：

update account set money = 900;-- #发现执行在等待，当另外一个客户端提交commit或者回滚rollback之后，修改才能成功。

另外一个客户端执行：

rollback/commit;

**死锁**

mysql可以自动检测到死锁，错误退出一方并执行另一方。

在7.1基础上：

两个客户端都执行：

start transaction;

select \* from account;

-----------------

一个客户端执行：

update account set money = 900;

另外一个客户端执行：

update account set money = 800;

发现彼此等待，直到一方报错结束，死锁才结束。

## 20.8.EasyMall添加商品 - 事务控制

事务应该加在业务层还是dao层？

因为dao层的一个方法就是一个C/R/U/D其中的一个操作，所以需要将事务的代码写在业务层的实现类中。

但是业务层获取的conn，和dao层的conn是各自的数据库连接，他们并没有共享在业务层获取的数据库连接，所以在调用方法时，应该将数据库连接对象作为参数传递过去。

EasyMall添加商品事务控制代码示例：（版本一）

|  |
| --- |
| 在ProdServiceImpl中添加事务控制代码，并将此connection通过参数传递到dao层，添加或修改如下代码：  **private** ProdDao dao = BasicFactory.*getFactory*().getInstance(ProdDao.**class**);    **public** **void** addProd(Prod prod) **throws** MsgException {  Connection conn = **null**;  **try** {  //开启事务  conn = JDBCUtils.*getConnection*();  conn.setAutoCommit(**false**);  //1.根据商品种类名称查询商品种类表  String cname = prod.getCname();  ProdCategory findPc = dao.findProdCategoryByCname(conn,cname);    //2.处理商品种类  **int** cid = 0;  **if**(findPc == **null**){  //如果找不到，则向商品种类表中加入新的商品种类  ProdCategory pc = **new** ProdCategory();  pc.setCname(cname);  dao.addProdCategory(pc);  //并获取新增加的商品的id 作为外键保存在商品信息里  ProdCategory findPc2 = dao.findProdCategoryByCname(conn,cname);  cid = findPc2.getId();  }**else**{  //如果查找到，则使用该商品种类的id作为外键保存在商品信息里  cid = findPc.getId();  }    //3.将商品信息加入商品表  prod.setCid(cid);  dao.addProd(conn,prod);  //提交事务  conn.commit();  } **catch** (SQLException e) {  //回滚事务  **try** {  **if**(conn != **null**){  conn.rollback();  }  } **catch** (SQLException e1) {  e1.printStackTrace();  }  e.printStackTrace();  **throw** **new** RuntimeException();  }  } |
| Dao层代码也需要为之改变（不做演示）。 |

此代码虽然可以解决事务控制问题，但是代码耦合性大

Service要控制事务，事务需要Connection对象，Connection对象是Dao层特有的对象，Service不应该持有Connection。

Service不能用Connection，又要控制事务，矛盾了，怎么办？

有些耦合性无法避免 --> 想办法管理起来 --> 将所有控制事务的代码整合到一个工具类中来进行管理

开发TransactionManager，在其中管理Connection，并对外提供getConn、startTran、commitTran、rollbackTran、release方法。

之后所有和事务 相关的操作都不要直接使用Conn 而是通过TransactionManager来实现管理

解决耦合性的问题 - 本质上是将耦合转移到了TransactionManager中同一管理。

虽然没有彻底的解决耦合，但是统一管理起来，方便未来开发和维护。

EasyMall添加商品事务控制代码示例：（版本二）

|  |
| --- |
| 在utils包中创建类：TransactionManager，并添加如下代码：  **private** TransactionManager(){  }  **private** **static** Connection *conn* = **null**;  **static**{  *conn* = JDBCUtils.*getConnection*();  }  /\*\*获取数据库连接的方法  \* **@return** 数据库连接对象  \*/  **public** **static** Connection getConn(){  **return** *conn*;  }  /\*\*开启事务  \*/  **public** **static** **void** startTran(){  **try** {  *conn*.setAutoCommit(**false**);  } **catch** (SQLException e) {  e.printStackTrace();  **throw** **new** RuntimeException();  }  }  /\*\*提交事务  \*/  **public** **static** **void** commitTran(){  **try** {  *conn*.commit();  } **catch** (SQLException e) {  e.printStackTrace();  **throw** **new** RuntimeException();  }  }  /\*\*回滚事务  \*/  **public** **static** **void** rollbackTran(){  **try** {  *conn*.rollback();  } **catch** (SQLException e) {  e.printStackTrace();  **throw** **new** RuntimeException();  }  }  /\*\*释放资源  \*/  **public** **static** **void** release(){  JDBCUtils.*close*(*conn*, **null**, **null**);  } |
| 在ProdServiceImpl中修改代码如下：  **private** ProdDao dao = BasicFactory.*getFactory*().getInstance(ProdDao.**class**);    **public** **void** addProd(Prod prod) **throws** MsgException {  **try** {  //开启事务  TransactionManager.*startTran*();  //1.根据商品种类名称查询商品种类表  String cname = prod.getCname();  ProdCategory findPc = dao.findProdCategoryByCname(cname);  //2.处理商品种类  **int** cid = 0;  **if**(findPc == **null**){  //如果找不到，则向商品种类表中加入新的商品种类  ProdCategory pc = **new** ProdCategory();  pc.setCname(cname);  dao.addProdCategory(pc);  //并获取新增加的商品的id 作为外键保存在商品信息里  ProdCategory findPc2 = dao.findProdCategoryByCname(cname);  cid = findPc2.getId();  }**else**{  //如果查找到，则使用该商品种类的id作为外键保存在商品信息里  cid = findPc.getId();  }  //3.将商品信息加入商品表  prod.setCid(cid);  dao.addProd(prod);  //提交事务  TransactionManager.*commitTran*();  } **catch** (Exception e) {  //回滚事务  TransactionManager.*rollbackTran*();  e.printStackTrace();  **throw** **new** RuntimeException();  }**finally**{  //关闭数据库链接  TransactionManager.*release*();  }  } |
| 在ProdDaoImpl中，将获取conn的代码从JDBCUtils.getConnection()改为从TransactionManager中获取，例如：  conn = TransactionManager.*getConn*();  将所有关闭数据库链接的操作中，不要将链接conn关闭，代码改为：  } **finally** {  JDBCUtils.*close*(**null**, ps, rs);  } |

测试发现，当添加完商品再次添加后，就会抛出异常：

SQLException: You can't operate on a closed Connection!!!

由于第三方管理类中，连接定义成了一个静态的，整个项目中共用一个数据库连接对象，添加完订单之后，关闭了该连接对象，再次查询全部商品时，使用的连接对象是一个已经关闭的数据库连接对象，所以抛出以上异常。

经过分析要想解决问题，需要让每一个服务的线程中保存一个独立数据库连接对象。

## 20.9.ThreadLocal-本地线程变量

在线程内部保存数据，利用线程对象在线程执行的过程中传递数据，另外由于每个线程各自保存各自的数据conn，所以可以避免线程并发安全问题。

Thread对象内置了一个Map来存取消息，但是这个map外界无法直接操作，需要通过ThreadLocal来实现对Thread中的Map进行数据的存取。

**ThreadLocal方法**

ThreadLocal tl = new ThreadLocal();

tl.set(obj); --向当前线程中的map保存对象，key为当前线程变量，值为存入的数据。

tl.get(); --从当前线程中获取对象，如果获取不到对象，调用initialValue()创建一个新的对象。

tl.remove(); --从当前线程中删除对象。

initialValue() --创建对象。

EasyMall添加商品事务控制代码示例：（版本三：最终）

|  |
| --- |
| 在TransactionManager中，加入ThreadLocal代码，通过本地线程变量获取conn，代码如下：  **private** TransactionManager(){  }    **private** **static** ThreadLocal<Connection> *tl* = **new** ThreadLocal<Connection>(){  **protected** Connection initialValue() {  **return** JDBCUtils.*getConnection*();  };  };    /\*\*获取数据库连接的方法  \* **@return** 数据库连接对象  \*/  **public** **static** Connection getConn(){  **return** *tl*.get();  }  /\*\*开启事务  \*/  **public** **static** **void** startTran(){  **try** {  *tl*.get().setAutoCommit(**false**);  } **catch** (SQLException e) {  e.printStackTrace();  **throw** **new** RuntimeException();  }  }  /\*\*提交事务  \*/  **public** **static** **void** commitTran(){  **try** {  *tl*.get().commit();  } **catch** (SQLException e) {  e.printStackTrace();  **throw** **new** RuntimeException();  }  }  /\*\*回滚事务  \*/  **public** **static** **void** rollbackTran(){  **try** {  *tl*.get().rollback();  } **catch** (SQLException e) {  e.printStackTrace();  **throw** **new** RuntimeException();  }  }  /\*\*释放资源  \*/  **public** **static** **void** release(){  JDBCUtils.*close*(*tl*.get(), **null**, **null**);  //从本地线程变量中删除  *tl*.remove();  } |

## 20.10.Easymall商品模块开发

### 20.10.1.商品列表

index.jsp：提供“所有商品”链接

ProdListServlet：调用service查询所有商品信息，存入request域带到页面展示。

ProdService：提供查询所有商品的方法

ProdDao：提供查询所有商品的方法

prodList.jsp：展示所有商品

代码示例：

|  |
| --- |
| 在\_head.jsp中，全部商品对应的<a>标签中修改href属性：  <li><a href=*"*${ app }*/servlet/ProdListServlet"*>全部商品</a></li> |
| 在web包中创建servlet：*ProdListServlet，并添加如下代码：*  //1.调用Service查询寻所有上商品  ProdService service = BasicFactory.*getFactory*().getInstance(ProdService.**class**);  List<Prod> list = service.findProdList();  //2.将查询到的所有商品存入request域  request.setAttribute("list", list);  //3.转发到prodList.jsp展示数据  request.getRequestDispatcher("/prodList.jsp").forward(request, response); |
| 在ProdService接口中添加findProdList方法：  /\*\*  \* 查询所有商品  \* **@return** 所有商品组成的列表  \*/  List<Prod> findProdList(); |
| 在ProdServiceImpl中实现该方法，并调用dao层findProdList方法：  **public** List<Prod> findProdList() {  **return** dao.findProdList();  } |
| 在ProdDao接口中添加findProdList方法：  /\*\*  \* 查询所有商品  \* **@return** 所有商品信息组成的bean  \*/  List<Prod> findProdList(); |
| 在ProdDaoImpl中实现该方法，并添加如下代码：  **public** List<Prod> findProdList() {  Connection conn = **null**;  PreparedStatement ps = **null**;  ResultSet rs = **null**;  **try**{  conn = JDBCUtils.*getConnection*();  String sql = "select \* from prod,prod\_category where prod.cid = prod\_category.id";  ps = conn.prepareStatement(sql);  rs = ps.executeQuery();  List<Prod> list = **new** ArrayList<Prod>();  **while**(rs.next()){  Prod prod = **new** Prod();  prod.setId(rs.getInt("prod.id"));  prod.setName(rs.getString("prod.name"));  prod.setPrice(rs.getDouble("prod.price"));  prod.setCid(rs.getInt("prod.cid"));  prod.setCname(rs.getString("prod\_category.cname"));  prod.setPnum(rs.getInt("pnum"));  prod.setImgurl(rs.getString("imgurl"));  prod.setDescription(rs.getString("description"));  list.add(prod);  }  **return** list;  }**catch** (Exception e) {  **throw** **new** RuntimeException(e);  } **finally** {  JDBCUtils.*close*(conn, ps, rs);  }  } |
| 在WebRoot下创建prodList.jsp，加入page指令标签，将编码改为utf-8，并将课前资料静态页面中的prodlist文件夹中css和img拷贝到WebRoot中，将prodList.html中的内容全部复制到prodList.jsp中。  在相对路径改为绝对路径：即在路径前加“${ app }/”即可  在<body>中，在最上面将头部包含进来，在最后将尾部包含进来：  <body>  <!-- 将头部引入进来 -->  <%@ include file=*"/\_head.jsp"* %>  ......  <!-- 将尾部引入进来 -->  <%@ include file=*"/\_foot.jsp"* %>  </body>  需要通过循环，将list中所有商品展示出来，所以在jsp最上面需要引入jstl标签库：  <%@ taglib uri=*"http://java.sun.com/jsp/jstl/core"* prefix=*"c"*%>  在<div id="prod\_content">中，将内容修改为如下代码：  <c:forEach items="${ requestScope.list }" var=*"prod"*>  <div id=*"prod\_div"*>  <img src=*"*${ app }*/servlet/ProdImgServlet?imgurl=*${ prod.imgurl }*"*></img>  <div id=*"prod\_name\_div"*>  ${ prod.name }  </div>  <div id=*"prod\_price\_div"*>  ${ prod.price }元  </div>  <div>  <div id=*"gotocart\_div"*>  <a href=*"#"*>加入购物车</a>  </div>  <div id=*"say\_div"*>  133人评价  </div>  </div>  </div>  </c:forEach> |
| 图片中的src属性地址如果直接写图片地址，则无法访问，因为图片在WEB-INF下，外界无法访问，只能通过程序内部访问，所以图片访问一个servlet，并将图片地址作为参数传给servlet，servlet中获取图片路径，并将图片读取到，再响应给jsp。在web包下创建servlet：ProdImgServlet，并添加如下代码：  String imgurl = request.getParameter("imgurl");  InputStream in = **new** FileInputStream(**this**.getServletContext().getRealPath(imgurl));  OutputStream out = response.getOutputStream();  **byte** [] data = **new** **byte**[1024];  **int** len = -1;  **while**((len = in.read(data))!=-1){  out.write(data,0,len);  }  in.close(); |

### 20.10.2.商品修改

在backend/manage.jsp 提供修改商品功能，访问ManageProdListServlet。

ManageProdListServlet 查询所有商品，存入request域并带到页面展示。

manageProdList.jsp 展示所有商品信息，并提供修改商品数量的输入框，当 修改数量时触发AJAX请求访问服务器，修改数量。

ManageUpdatePnumServlet 获取商品编号和最新数量，调用Service修改商品数量，发回成功或失败的消息。

ProdService 提供方法，可以通过调用dao，根据商品编号修改商品库存。

ProdDao 提供方法，通过商品编号修改商品库存。

代码示例：

|  |
| --- |
| 在backend下\_left.jsp中，修改商品修改对应<div>：  <div><a href=*"*${ app }*/servlet/ManageProdListServlet"* target=*"rightFrame"*>> 商品修改</a></div> |
| 在backend包下创建servlet：ManageProdListServlet，并添加如下代码：  //1.查询所有商品信息  ProdService service = BasicFactory.*getFactory*().getInstance(ProdService.**class**);  List<Prod> list = service.findProdList();  //2.存入request域中  request.setAttribute("list", list);  //3.带到manageProdList.jsp页面展示  request.getRequestDispatcher("/backend/manageProdList.jsp").forward(request, response); |
| 将课前资料中的backend文件夹中manageProdList.jsp拷贝到WebRoot下的backend文件夹中，并添加taglib指令，添加for循环展示商品，并在商品库存栏提供输入框，并添加onblur事件，当失去焦点时触发ajax进行数量的修改，代码如下：  Jsp最上面引入taglib指令：  <%@ taglib uri=*"http://java.sun.com/jsp/jstl/core"* prefix=*"c"* %>  在<table>中，删除模板数据，并通过循环遍历list中所有商品信息：  <c:forEach items="${requestScope.list }" var=*"prod"* >  <tr>  <td><img width=*"120px"* height=*"120px"* src=*"*${ app }*/servlet/ProdImgServlet?imgurl=*${prod.imgurl}*"*/></td>  <td>${prod.id }</td>  <td>${prod.name }</td>  <td>${prod.cname }</td>  <td>${prod.price }</td>  <td><input type=*"text"* value=*"*${prod.pnum }*"* style="width: *50px*" onblur=*"changeNum(this,'*${prod.id}*')"*/></td>  <td>${prod.description }</td>  <td><a class=*"del"* href=*"javascript:void(0)"*>删 除</a></td>  </tr>  </c:forEach>  在<head>中添加<script>标签，引入ajax：  <**script** **type**="text/javascript" **src**="${ app }/**js**/jquery-1*.4.2.js*"></**script**>  <**script** **type**="text/javascript">  **function** **changeNum**(inp**,id**){  $.get("${app}/**servlet**/AjaxChangePnumServlet"**,** { "**id**" *:* **id,** "**pnum**" *:* **inp***.value*}, **function**(data){  alert(data);  });  }  </**script**> |
| 在backend包中创建servlet：AjaxChangePnumServlet，并添加如下代码：  **try** {  //1.获取参数  **int** id = Integer.*parseInt*(request.getParameter("id"));  **int** pnum = Integer.*parseInt*(request.getParameter("pnum"));  //2.调用Service修改商品  ProdService service = BasicFactory.*getFactory*().getInstance(ProdService.**class**);  service.changePnum(id,pnum);  //3.输出处理结果  response.getWriter().write("商品数量修改成功！");  } **catch** (Exception e) {  response.getWriter().write("商品数量修改失败！");  e.printStackTrace();  } |
| 在ProdService接口中添加如下代码：  /\*\*  \* 修改指定id商品的 库存数量  \* **@param** id 商品编号  \* **@param** pnum 最新的数量  \*/  **void** changePnum(**int** id, **int** pnum); |
| 在ProdServiceImpl实现类中调用dao层方法，代码如下：  **public** **void** changePnum(**int** id, **int** pnum) {  dao.changePnum(id,pnum);  } |
| 在ProdDao接口中添加如下代码：  /\*\*  \* 修改商品数量  \* **@param** id 商品编号  \* **@param** pnum 新的库存数量  \*/  **void** changePnum(**int** id, **int** pnum); |
| 在ProdDaoImpl实现类中添加如下代码：  **public** **void** changePnum(**int** id, **int** pnum) {  Connection conn = **null**;  PreparedStatement ps = **null**;  ResultSet rs = **null**;  **try**{  conn = JDBCUtils.*getConnection*();  String sql = "update prod set pnum = ? where id = ?";  ps = conn.prepareStatement(sql);  ps.setInt(1, pnum);  ps.setInt(2, id);  ps.executeUpdate();  }**catch** (Exception e) {  **throw** **new** RuntimeException(e);  } **finally** {  JDBCUtils.*close*(conn, ps, rs);  }  } |

### 20.10.3.删除商品

在backend/manageProdList.jsp 商品中提供删除按钮，点击后带着当前商品的id访问后台。

ManageDelProdServlet 获取要删除的商品的id 调用Service层删除商品，删除成功后回到商品管理列表页面。

ProdService 提供根据id删除商品的方法：按照商品id查询商品信息，获取商品种类id，根据商品种类id查询商品种类，根据商品种类id查询商品表，判断该种类商品还剩多少。如果只有当前这件商品，则删除商品信息及商品种类信息，如果不是，则只删除当前商品信息。

ProdDao 提供相应方法。

代码示例：

|  |
| --- |
| 在manageProdList.jsp中，修改删除对应<a>标签中个href属性：  <td><a class=*"del"* href=*"*${ app }*/servlet/ManageDelProdServlet?id=*${prod.id}*"*>删 除</a></td> |
| 在backend包下创建servlet：ManageDelProdServlet，并添加如下代码：  //1.获取要删除的商品id  **int** id = Integer.*parseInt*(request.getParameter("id"));  //2.调用Service删除指定id的商品  ProdService service = BasicFactory.*getFactory*().getInstance(ProdService.**class**);  service.delProd(id);  //3.重定向回到商品管理页面  response.getWriter().write("删除成功！");  response.setHeader("Refresh", "1;url="+request.getContextPath()+"/servlet/ManageProdListServlet"); |
| 在ProdService接口中添加如下代码：  /\*\*  \* 删除指定id商品  \* **@param** id 要删除的商品的id  \*/  **void** delProd(**int** id); |
| 在ProdServiceImpl实现类中添加如下代码：  **public** **void** delProd(**int** id) {  **try** {  TransactionManager.*startTran*();  //1.根据商品id查找商品信息  Prod prod = dao.findProdById(id);  //2.获取种类  **int** cid = prod.getCid();  //3.根据商品种类查询商品表 获知 该种类的商品剩余几种  List<Prod> list = dao.findProdByCid(cid);  //4.根据不同情况处理  **if**(list.size() > 1){  //--如果大于1 则只删除当前商品即可  dao.delProdById(id);  }**else**{  //--如果小于等于1 则删除当前商品 再删除该商品种类  dao.delProdById(id);  dao.delProdCategoryByCid(cid);  }  TransactionManager.*commitTran*();  } **catch** (Exception e) {  TransactionManager.*rollbackTran*();  e.printStackTrace();  **throw** **new** RuntimeException(e);  } **finally** {  TransactionManager.*release*();  }  } |
| 在ProdDao接口中添加如下代码：  /\*\*  \* 根据商品id查询商品  \* **@param** id 商品id  \* **@return** 封装了商品信息的bean  \*/  Prod findProdById(**int** id);  /\*\*  \* 根据商品种类查询商品信息  \* **@param** cid 商品的种类id  \* **@return** 找到的所有商品组成的bean  \*/  List<Prod> findProdByCid(**int** cid);  /\*\*  \* 根据id删除商品  \* **@param** id 商品编号  \*/  **void** delProdById(**int** id);  /\*\*  \* 根据商品种类id 删除商品种类  \* **@param** cid 商品种类编号  \*/  **void** delProdCategoryByCid(**int** cid); |
| 在ProdDaoImpl实现类中添加如下代码：  **public** Prod findProdById(**int** id) {  Connection conn = **null**;  PreparedStatement ps = **null**;  ResultSet rs = **null**;  **try**{  conn = TransactionManager.*getConn*();  String sql = "select \* from prod where id = ?";  ps = conn.prepareStatement(sql);  ps.setInt(1, id);  rs = ps.executeQuery();  Prod prod = **null**;  **if**(rs.next()){  prod = **new** Prod();  prod.setId(rs.getInt("id"));  prod.setName(rs.getString("name"));  prod.setPrice(rs.getDouble("price"));  prod.setCid(rs.getInt("cid"));  prod.setPnum(rs.getInt("pnum"));  prod.setImgurl(rs.getString("imgurl"));  prod.setDescription(rs.getString("description"));  }  **return** prod;  }**catch** (Exception e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  } **finally** {  JDBCUtils.*close*(**null**, ps, rs);  }  }  **public** List<Prod> findProdByCid(**int** cid) {  Connection conn = **null**;  PreparedStatement ps = **null**;  ResultSet rs = **null**;  **try**{  conn = TransactionManager.*getConn*();  String sql = "select \* from prod where cid = ?";  ps = conn.prepareStatement(sql);  ps.setInt(1, cid);  rs = ps.executeQuery();  List<Prod> list = **new** ArrayList<Prod>();  **while**(rs.next()){  Prod prod = **new** Prod();  prod.setId(rs.getInt("id"));  prod.setName(rs.getString("name"));  prod.setPrice(rs.getDouble("price"));  prod.setCid(rs.getInt("cid"));  prod.setPnum(rs.getInt("pnum"));  prod.setImgurl(rs.getString("imgurl"));  prod.setDescription(rs.getString("description"));  list.add(prod);  }  **return** list;  }**catch** (Exception e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  } **finally** {  JDBCUtils.*close*(**null**, ps, rs);  }  }  **public** **void** delProdById(**int** id) {  Connection conn = **null**;  PreparedStatement ps = **null**;  ResultSet rs = **null**;  **try**{  conn = TransactionManager.*getConn*();  String sql = "delete from prod where id = ?";  ps = conn.prepareStatement(sql);  ps.setInt(1, id);  ps.executeUpdate();  }**catch** (Exception e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  } **finally** {  JDBCUtils.*close*(**null**, ps, rs);  }  }  **public** **void** delProdCategoryByCid(**int** cid) {  Connection conn = **null**;  PreparedStatement ps = **null**;  ResultSet rs = **null**;  **try**{  conn = TransactionManager.*getConn*();  String sql = "delete from prod\_category where id = ?";  ps = conn.prepareStatement(sql);  ps.setInt(1, cid);  ps.executeUpdate();  }**catch** (Exception e) {  e.printStackTrace();  **throw** **new** RuntimeException(e);  } **finally** {  JDBCUtils.*close*(**null**, ps, rs);  }  } |

## 20.11.作业练习

1. 数据库技术中的“脏数据”，是指（）的数据

A、错误

B、回返

C、未提交

D、未提交的随后又被撤销

2.事务的概念和特点？

答：事务指逻辑上的一组操作，组成这组操作的各个单元，要么全部成功，要么全部不 成功。

四大特点，分别是原子性、一致性、隔离性、持久性。

3.说说共享锁和排它锁？

答：共享锁和共享锁可以共存，共享锁和排他锁不能共存.在非Serializable隔离级别下 做查询不加任何锁，在Serializable隔离级别下做查询加共享锁。

排他锁和共享锁不能共存，排他锁和排他锁也不能共存，在任何隔离级别下做增删改都加排他锁。

1. 事务的隔离级别？

答：read uncommitted;

不做任何隔离，可能造成脏读、不可重复度、虚读(幻读)问题

read committed;

可以防止脏读，但是不能防止不可重复度、虚读(幻读)问题

repeatable Read;

可以防止脏读、不可重复度，但是不能防止虚读(幻读)问题

serializable;

可以防止所有隔离性的问题，但是数据库就被设计为了串行化的数据库，性能 很低

1. 说说ThreadLocal？

答：ThreadLocal是本地线程变量，用来解决多线程的并发问题，当使用ThreadLocal维护变量时，ThreadLocal为每个使用该变量的线程提供独立的变量副本，所以每个线程都可以独立地改变自己的副本，而不会影响其它线程所对应的副本。从线程的角度看，就好像每一个线程都完全拥有该变量。