三、DbUtils类使用讲解

DbUtils ：提供如关闭连接、装载JDBC驱动程序等常规工作的工具类，里面的所有方法都是静态的。主要方法如下：
public static void close(…) throws java.sql.SQLException： DbUtils类提供了三个重载的关闭方法。这些方法检查所提供的参数是不是NULL，如果不是的话，它们就关闭Connection、Statement和ResultSet。
public static void closeQuietly(…): 这一类方法不仅能在Connection、Statement和ResultSet为NULL情况下避免关闭，还能隐藏一些在程序中抛出的SQLEeception。
public static void commitAndCloseQuietly(Connection conn)： 用来提交连接，然后关闭连接，并且在关闭连接时不抛出SQL异常。
public static boolean loadDriver(java.lang.String driverClassName)：这一方装载并注册JDBC驱动程序，如果成功就返回true。使用该方法，你不需要捕捉这个异常ClassNotFoundException。

四、JDBC开发中的事务处理

在开发中，对数据库的多个表或者对一个表中的多条数据执行更新操作时要保证对多个更新操作要么同时成功，要么都不成功，这就涉及到对多个更新操作的事务管理问题了。比如银行业务中的转账问题，A用户向B用户转账100元，假设A用户和B用户的钱都存储在Account表，那么A用户向B用户转账时就涉及到同时更新Account表中的A用户的钱和B用户的钱，用SQL来表示就是：

```
1 update account set money=money-100 where name='A'
2 update account set money=money+100 where name='B'
```

4.1、在数据访问层(Dao)中处理事务

对于这样的同时更新一个表中的多条数据的操作，那么必须保证要么同时成功，要么都不成功，所以需要保证这两个update操作在同一个事务中进行。在开发中，我们可能会在AccountDao写一个转账处理方法，如下：

```
1 /**
2     * @Method: transfer
3     * @Description:这个方法是用来处理两个用户之间的转账业务
4     * 在开发中，DAO层的职责应该只涉及到CRUD，
5     * 而这个transfer方法是处理两个用户之间的转账业务的，已经涉及到具体的业务操作，应该在业务层中做，不应该出现在DAO层的
6     * 所以在开发中DAO层出现这样的业务处理方法是完全错误的
7     * @Anthor:孤傲苍狼
8     *
9     * @param sourceName
10    * @param targetName
11    * @param money
12    * @throws SQLException
13    */
14    public void transfer(String sourceName,String targetName,float money) throws SQLException{
15        Connection conn = null;
16        try{
17            conn = JdbcUtils.getConnection();
18            //开启事务
19            conn.setAutoCommit(false);
20            /**
21             * 在创建QueryRunner对象时，不传递数据源给它，是为了保证这两条SQL在同一个事务中进行，
22             * 我们手动获取数据库连接，然后让这两条SQL使用同一个数据库连接执行
23             */
24            QueryRunner runner = new QueryRunner();
25            String sql1 = "update account set money=money-100 where name=?";
26            String sql2 = "update account set money=money+100 where name=?";
27            Object[] paramArr1 = {sourceName};
28            Object[] paramArr2 = {targetName};
29            runner.update(conn,sql1,paramArr1);
30            //模拟程序出现异常让事务回滚
31            int x = 1/0;
32            runner.update(conn,sql2,paramArr2);
33            //sql正常执行之后就提交事务
34            conn.commit();
35        }catch (Exception e) {
36            e.printStackTrace();
37            if(conn!=null){
```

```
                //出现异常之后就回滚事务
                conn.rollback();
            }
        }finally{
            //关闭数据库连接
            conn.close();
        }
    }
```

然后我们在AccountService中再写一个同名方法，在方法内部调用AccountDao的transfer方法处理转账业务，如下：

```
public void transfer(String sourceName,String targetName,float money) throws SQLException{
        AccountDao dao = new AccountDao();
        dao.transfer(sourceName, targetName, money);
}
```

上面AccountDao的这个transfer方法可以处理转账业务，并且保证了在同一个事务中进行，但是AccountDao的这个transfer方法是处理两个用户之间的转账业务的，已经涉及到具体的业务操作，应该在业务层中做，不应该出现在DAO层的，在开发中，DAO层的职责应该只涉及到基本的CRUD，不涉及具体的业务操作，所以在开发中DAO层出现这样的业务处理方法是一种不好的设计。

4.2、在业务层(BusinessService)处理事务

由于上述AccountDao存在具体的业务处理方法，导致AccountDao的职责不够单一，下面我们对AccountDao进行改造，让AccountDao的职责只是做CRUD操作，将事务的处理挪到业务层(BusinessService)，改造后的AccountDao如下：

```
package me.gacl.dao;

import java.sql.Connection;
import java.sql.SQLException;
import org.apache.commons.dbutils.QueryRunner;
import org.apache.commons.dbutils.handlers.BeanHandler;
import me.gacl.domain.Account;
import me.gacl.util.JdbcUtils;

/*account测试表
create table account(
    id int primary key auto_increment,
    name varchar(40),
    money float
)character set utf8 collate utf8_general_ci;

insert into account(name,money) values('A',1000);
insert into account(name,money) values('B',1000);
insert into account(name,money) values('C',1000);

*/

/**
 * @ClassName: AccountDao
 * @Description: 针对Account对象的CRUD
 * @author: 孤傲苍狼
 * @date: 2014-10-6 下午4:00:42
 *
 */
public class AccountDao {

    //接收service层传递过来的Connection对象
    private Connection conn = null;

```

```
35      public AccountDao(Connection conn){
36          this.conn = conn;
37      }
38
39      public AccountDao(){
40
41      }
42
43      /**
44      * @Method: update
45      * @Description:更新
46      * @Anthor:孤傲苍狼
47      *
48      * @param account
49      * @throws SQLException
50      */
51      public void update(Account account) throws SQLException{
52
53          QueryRunner qr = new QueryRunner();
54          String sql = "update account set name=?,money=? where id=?";
55          Object params[] = {account.getName(),account.getMoney(),account.getId()};
56          //使用service层传递过来的Connection对象操作数据库
57          qr.update(conn,sql, params);
58
59      }
60
61      /**
62      * @Method: find
63      * @Description:查找
64      * @Anthor:孤傲苍狼
65      *
66      * @param id
67      * @return
68      * @throws SQLException
69      */
70      public Account find(int id) throws SQLException{
71          QueryRunner qr = new QueryRunner();
72          String sql = "select * from account where id=?";
73          //使用service层传递过来的Connection对象操作数据库
74          return (Account) qr.query(conn,sql, id, new BeanHandler(Account.class));
75      }
76  }
```

接着对AccountService(业务层)中的transfer方法的改造，在业务层(BusinessService)中处理事务

```
 1  package me.gacl.service;
 2
 3  import java.sql.Connection;
 4  import java.sql.SQLException;
 5  import me.gacl.dao.AccountDao;
 6  import me.gacl.domain.Account;
 7  import me.gacl.util.JdbcUtils;
 8
 9  /**
10  * @ClassName: AccountService
11  * @Description: 业务逻辑处理层
```

```
 * @author: 孤傲苍狼
 * @date: 2014-10-6 下午5:30:15
 *
 */
public class AccountService {

    /**
     * @Method: transfer
     * @Description:这个方法是用来处理两个用户之间的转账业务
     * @Anthor:孤傲苍狼
     *
     * @param sourceid
     * @param tartgetid
     * @param money
     * @throws SQLException
     */
    public void transfer(int sourceid,int tartgetid,float money) throws SQLException{
        Connection conn = null;
        try{
            //获取数据库连接
            conn = JdbcUtils.getConnection();
            //开启事务
            conn.setAutoCommit(false);
            //将获取到的Connection传递给AccountDao，保证dao层使用的是同一个Connection对象操作数据库
            AccountDao dao = new AccountDao(conn);
            Account source = dao.find(sourceid);
            Account target = dao.find(tartgetid);

            source.setMoney(source.getMoney()-money);
            target.setMoney(target.getMoney()+money);

            dao.update(source);
            //模拟程序出现异常让事务回滚
            int x = 1/0;
            dao.update(target);
            //提交事务
            conn.commit();
        }catch (Exception e) {
            e.printStackTrace();
            //出现异常之后就回滚事务
            conn.rollback();
        }finally{
            conn.close();
        }
    }
}
```

程序经过这样改造之后就比刚才好多了，AccountDao只负责CRUD，里面没有具体的业务处理方法了，职责就单一了，而AccountService则负责具体的业务逻辑和事务的处理，需要操作数据库时，就调用AccountDao层提供的CRUD方法操作数据库。

4.3、使用ThreadLocal进行更加优雅的事务处理

上面的在businessService层这种处理事务的方式依然不够优雅，为了能够让事务处理更加优雅，我们使用ThreadLocal类进行改造，**ThreadLocal一个容器，向这个容器存储的对象，在当前线程范围内都可以取得出来，向ThreadLocal里面存东西就是向它里面的Map存东西的，然后ThreadLocal把这个Map挂到当前的线程底下，这样Map就只属于这个线程了**

ThreadLocal类的使用范例如下：

```
package me.gacl.test;

public class ThreadLocalTest {

    public static void main(String[] args) {
        //得到程序运行时的当前线程
        Thread currentThread = Thread.currentThread();
        System.out.println(currentThread);
        //ThreadLocal一个容器，向这个容器存储的对象，在当前线程范围内都可以取得出来
        ThreadLocal<String> t = new ThreadLocal<String>();
        //把某个对象绑定到当前线程上 对象以键值对的形式存储到一个Map集合中，对象的的key是当前的线程，如： map(currentThread,"aaa")
        t.set("aaa");
        //获取绑定到当前线程中的对象
        String value = t.get();
        //输出value的值是aaa
        System.out.println(value);
    }
}
```

使用使用ThreadLocal类进行改造数据库连接工具类JdbcUtils，改造后的代码如下：

```
package me.gacl.util;

import java.sql.Connection;
import java.sql.SQLException;
import javax.sql.DataSource;
import com.mchange.v2.c3p0.ComboPooledDataSource;

/**
 * @ClassName: JdbcUtils2
 * @Description: 数据库连接工具类
 * @author: 孤傲苍狼
 * @date: 2014-10-4 下午6:04:36
 *
 */
public class JdbcUtils2 {

    private static ComboPooledDataSource ds = null;
    //使用ThreadLocal存储当前线程中的Connection对象
    private static ThreadLocal<Connection> threadLocal = new ThreadLocal<Connection>();

    //在静态代码块中创建数据库连接池
    static{
        try{
            //通过代码创建C3P0数据库连接池
            /*ds = new ComboPooledDataSource();
            ds.setDriverClass("com.mysql.jdbc.Driver");
            ds.setJdbcUrl("jdbc:mysql://localhost:3306/jdbcstudy");
            ds.setUser("root");
            ds.setPassword("XDP");
            ds.setInitialPoolSize(10);
            ds.setMinPoolSize(5);
            ds.setMaxPoolSize(20);*/

            //通过读取C3P0的xml配置文件创建数据源，C3P0的xml配置文件c3p0-config.xml必须放在src目录下
            //ds = new ComboPooledDataSource();//使用C3P0的默认配置来创建数据源
```

```
            ds = new ComboPooledDataSource("MySQL");//使用C3P0的命名配置来创建数据源

        }catch (Exception e) {
            throw new ExceptionInInitializerError(e);
        }
    }

    /**
    * @Method: getConnection
    * @Description: 从数据源中获取数据库连接
    * @Anthor:孤傲苍狼
    * @return Connection
    * @throws SQLException
    */
    public static Connection getConnection() throws SQLException{
        //从当前线程中获取Connection
        Connection conn = threadLocal.get();
        if(conn==null){
            //从数据源中获取数据库连接
            conn = getDataSource().getConnection();
            //将conn绑定到当前线程
            threadLocal.set(conn);
        }
        return conn;
    }

    /**
    * @Method: startTransaction
    * @Description: 开启事务
    * @Anthor:孤傲苍狼
    *
    */
    public static void startTransaction(){
        try{
            Connection conn =  threadLocal.get();
            if(conn==null){
                conn = getConnection();
                 //把 conn绑定到当前线程上
                threadLocal.set(conn);
            }
            //开启事务
            conn.setAutoCommit(false);
        }catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    /**
    * @Method: rollback
    * @Description:回滚事务
    * @Anthor:孤傲苍狼
    *
    */
    public static void rollback(){
        try{
            //从当前线程中获取Connection
            Connection conn = threadLocal.get();
            if(conn!=null){
```

```
                //回滚事务
                conn.rollback();
            }
        }catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    /**
    * @Method: commit
    * @Description:提交事务
    * @Anthor:孤傲苍狼
    *
    */
    public static void commit(){
        try{
            //从当前线程中获取Connection
            Connection conn = threadLocal.get();
            if(conn!=null){
                //提交事务
                conn.commit();
            }
        }catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    /**
    * @Method: close
    * @Description:关闭数据库连接(注意，并不是真的关闭，而是把连接还给数据库连接池)
    * @Anthor:孤傲苍狼
    *
    */
    public static void close(){
        try{
            //从当前线程中获取Connection
            Connection conn = threadLocal.get();
            if(conn!=null){
                conn.close();
                 //解除当前线程上绑定conn
                threadLocal.remove();
            }
        }catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    /**
    * @Method: getDataSource
    * @Description: 获取数据源
    * @Anthor:孤傲苍狼
    * @return DataSource
    */
    public static DataSource getDataSource(){
        //从数据源中获取数据库连接
        return ds;
    }
}
```

对AccountDao进行改造，数据库连接对象不再需要service层传递过来，而是直接从JdbcUtils2提供的getConnection方法去获取，改造后的AccountDao如下：

```
package me.gacl.dao;

import java.sql.Connection;
import java.sql.SQLException;
import org.apache.commons.dbutils.QueryRunner;
import org.apache.commons.dbutils.handlers.BeanHandler;
import me.gacl.domain.Account;
import me.gacl.util.JdbcUtils;
import me.gacl.util.JdbcUtils2;

/*
create table account(
    id int primary key auto_increment,
    name varchar(40),
    money float
)character set utf8 collate utf8_general_ci;

insert into account(name,money) values('A',1000);
insert into account(name,money) values('B',1000);
insert into account(name,money) values('C',1000);

*/

/**
 * @ClassName: AccountDao
 * @Description: 针对Account对象的CRUD
 * @author: 孤傲苍狼
 * @date: 2014-10-6 下午4:00:42
 *
 */
public class AccountDao2 {

    public void update(Account account) throws SQLException{

        QueryRunner qr = new QueryRunner();
        String sql = "update account set name=?,money=? where id=?";
        Object params[] = {account.getName(),account.getMoney(),account.getId()};
        //JdbcUtils2.getConnection()获取当前线程中的Connection对象
        qr.update(JdbcUtils2.getConnection(),sql, params);

    }

    public Account find(int id) throws SQLException{
        QueryRunner qr = new QueryRunner();
        String sql = "select * from account where id=?";
        //JdbcUtils2.getConnection()获取当前线程中的Connection对象
        return (Account) qr.query(JdbcUtils2.getConnection(),sql, id, new BeanHandler(Account.class));
    }
}
```

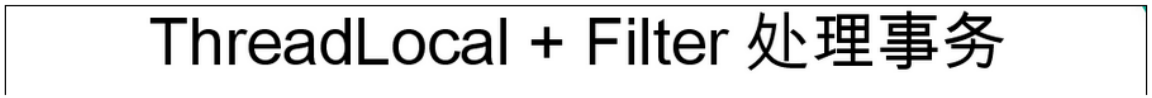
对AccountService进行改造，service层不再需要传递数据库连接Connection给Dao层，改造后的AccountService如下：

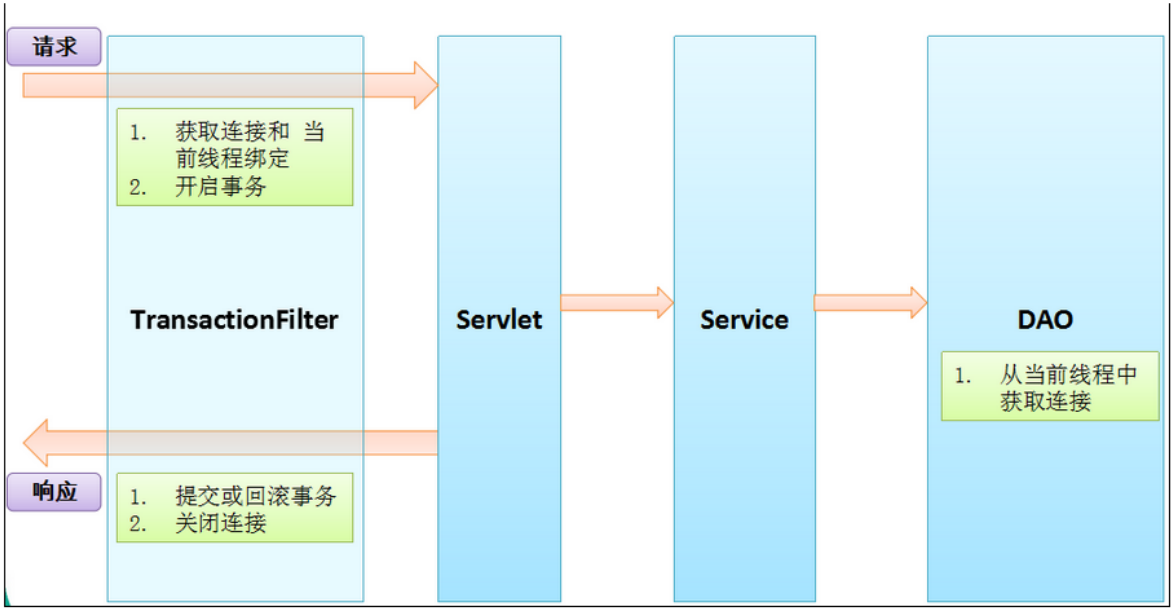
```java
package me.gacl.service;

import java.sql.SQLException;
import me.gacl.dao.AccountDao2;
import me.gacl.domain.Account;
import me.gacl.util.JdbcUtils2;

public class AccountService2 {

    /**
    * @Method: transfer
    * @Description:在业务层处理两个账户之间的转账问题
    * @Anthor:孤傲苍狼
    *
    * @param sourceid
    * @param tartgetid
    * @param money
    * @throws SQLException
    */
    public void transfer(int sourceid,int tartgetid,float money) throws SQLException{
        try{
            //开启事务，在业务层处理事务，保证dao层的多个操作在同一个事务中进行
            JdbcUtils2.startTransaction();
            AccountDao2 dao = new AccountDao2();

            Account source = dao.find(sourceid);
            Account target = dao.find(tartgetid);
            source.setMoney(source.getMoney()-money);
            target.setMoney(target.getMoney()+money);

            dao.update(source);
            //模拟程序出现异常让事务回滚
            int x = 1/0;
            dao.update(target);

            //SQL正常执行之后提交事务
            JdbcUtils2.commit();
        }catch (Exception e) {
            e.printStackTrace();
            //出现异常之后就回滚事务
            JdbcUtils2.rollback();
        }finally{
            //关闭数据库连接
            JdbcUtils2.close();
        }
    }
}
```

这样在service层对事务的处理看起来就更加优雅了。ThreadLocal类在开发中使用得是比较多的，**程序运行中产生的数据要想在一个线程范围内共享，只需要把数据使用ThreadLocal进行存储即可**。

4.4、ThreadLocal + Filter 处理事务

上面介绍了JDBC开发中事务处理的3种方式，下面再介绍的一种使用ThreadLocal + Filter进行统一的事务处理，这种方式主要是使用过滤器进行统一的事务处理，如下图所示：





1、编写一个事务过滤器TransactionFilter

代码如下：

```
 1 package me.gac.web.filter;
 2
 3 import java.io.IOException;
 4 import java.sql.Connection;
 5 import java.sql.SQLException;
 6 import javax.servlet.Filter;
 7 import javax.servlet.FilterChain;
 8 import javax.servlet.FilterConfig;
 9 import javax.servlet.ServletException;
10 import javax.servlet.ServletRequest;
11 import javax.servlet.ServletResponse;
12 import javax.servlet.http.HttpServletRequest;
13 import javax.servlet.http.HttpServletResponse;
14 import me.gacl.util.JdbcUtils;
15
16 /**
17 * @ClassName: TransactionFilter
18 * @Description:ThreadLocal + Filter 统一处理数据库事务
19 * @author: 孤傲苍狼
20 * @date: 2014-10-6 下午11:36:58
21 *
22 */
23 public class TransactionFilter implements Filter {
24
25     @Override
26     public void init(FilterConfig filterConfig) throws ServletException {
27
28     }
29
```

```
30      @Override
31      public void doFilter(ServletRequest request, ServletResponse response,
32              FilterChain chain) throws IOException, ServletException {
33
34          Connection connection = null;
35          try {
36              //1、获取数据库连接对象Connection
37              connection = JdbcUtils.getConnection();
38              //2、开启事务
39              connection.setAutoCommit(false);
40              //3、利用ThreadLocal把获取数据库连接对象Connection和当前线程绑定
41              ConnectionContext.getInstance().bind(connection);
42              //4、把请求转发给目标Servlet
43              chain.doFilter(request, response);
44              //5、提交事务
45              connection.commit();
46          } catch (Exception e) {
47              e.printStackTrace();
48              //6、回滚事务
49              try {
50                  connection.rollback();
51              } catch (SQLException e1) {
52                  e1.printStackTrace();
53              }
54              HttpServletRequest req = (HttpServletRequest) request;
55              HttpServletResponse res = (HttpServletResponse) response;
56              //req.setAttribute("errMsg", e.getMessage());
57              //req.getRequestDispatcher("/error.jsp").forward(req, res);
58              //出现异常之后跳转到错误页面
59              res.sendRedirect(req.getContextPath()+"/error.jsp");
60          }finally{
61              //7、解除绑定
62              ConnectionContext.getInstance().remove();
63              //8、关闭数据库连接
64              try {
65                  connection.close();
66              } catch (SQLException e) {
67                  e.printStackTrace();
68              }
69          }
70      }
71
72      @Override
73      public void destroy() {
74
75      }
76  }
```

我们在TransactionFilter中把获取到的数据库连接使用ThreadLocal绑定到当前线程之后，在DAO层还需要从ThreadLocal中取出数据库连接来操作数据库，因此需要编写一个ConnectionContext类来存储ThreadLocal，ConnectionContext类的代码如下：

```
1  package me.gac.web.filter;
2
3  import java.sql.Connection;
4
5  /**
```

```
 * @ClassName: ConnectionContext
 * @Description:数据库连接上下文
 * @author: 孤傲苍狼
 * @date: 2014-10-7 上午8:36:01
 *
 */
public class ConnectionContext {

    /**
     * 构造方法私有化，将ConnectionContext设计成单例
     */
    private ConnectionContext(){

    }
    //创建ConnectionContext实例对象
    private static ConnectionContext connectionContext = new ConnectionContext();

    /**
    * @Method: getInstance
    * @Description:获取ConnectionContext实例对象
    * @Anthor:孤傲苍狼
    *
    * @return
    */
    public static ConnectionContext getInstance(){
        return connectionContext;
    }

    /**
    * @Field: connectionThreadLocal
    *         使用ThreadLocal存储数据库连接对象
    */
    private ThreadLocal<Connection> connectionThreadLocal = new ThreadLocal<Connection>();

    /**
    * @Method: bind
    * @Description:利用ThreadLocal把获取数据库连接对象Connection和当前线程绑定
    * @Anthor:孤傲苍狼
    *
    * @param connection
    */
    public void bind(Connection connection){
        connectionThreadLocal.set(connection);
    }

    /**
    * @Method: getConnection
    * @Description:从当前线程中取出Connection对象
    * @Anthor:孤傲苍狼
    *
    * @return
    */
    public Connection getConnection(){
        return connectionThreadLocal.get();
    }

    /**
    * @Method: remove
    * @Description: 解除当前线程上绑定Connection
```

```
    * @Anthor:孤傲苍狼
    *
    */
    public void remove(){
        connectionThreadLocal.remove();
    }
}
```

在DAO层想获取数据库连接时，就可以使用ConnectionContext.getInstance().getConnection()来获取，如下所示：

```
package me.gacl.dao;

import java.sql.SQLException;
import org.apache.commons.dbutils.QueryRunner;
import org.apache.commons.dbutils.handlers.BeanHandler;

import me.gac.web.filter.ConnectionContext;
import me.gacl.domain.Account;

/*
create table account(
    id int primary key auto_increment,
    name varchar(40),
    money float
)character set utf8 collate utf8_general_ci;

insert into account(name,money) values('A',1000);
insert into account(name,money) values('B',1000);
insert into account(name,money) values('C',1000);

*/

/**
* @ClassName: AccountDao
* @Description: 针对Account对象的CRUD
* @author: 孤傲苍狼
* @date: 2014-10-6 下午4:00:42
*
*/
public class AccountDao3 {

    public void update(Account account) throws SQLException{

        QueryRunner qr = new QueryRunner();
        String sql = "update account set name=?,money=? where id=?";
        Object params[] = {account.getName(),account.getMoney(),account.getId()};
        //ConnectionContext.getInstance().getConnection()获取当前线程中的Connection对象
        qr.update(ConnectionContext.getInstance().getConnection(),sql, params);

    }

    public Account find(int id) throws SQLException{
        QueryRunner qr = new QueryRunner();
        String sql = "select * from account where id=?";
        //ConnectionContext.getInstance().getConnection()获取当前线程中的Connection对象
        return (Account) qr.query(ConnectionContext.getInstance().getConnection(),sql, id, new BeanHandler(Account.class));
```

```
    }
}
```

businessService层也不用处理事务和数据库连接问题了，这些统一在TransactionFilter中统一管理了，businessService层只需要专注业务逻辑的处理即可，如下所示：

```
package me.gacl.service;

import java.sql.SQLException;
import me.gacl.dao.AccountDao3;
import me.gacl.domain.Account;

public class AccountService3 {

    /**
    * @Method: transfer
    * @Description:在业务层处理两个账户之间的转账问题
    * @Anthor:孤傲苍狼
    *
    * @param sourceid
    * @param tartgetid
    * @param money
    * @throws SQLException
    */
    public void transfer(int sourceid, int tartgetid, float money)
            throws SQLException {
        AccountDao3 dao = new AccountDao3();
        Account source = dao.find(sourceid);
        Account target = dao.find(tartgetid);
        source.setMoney(source.getMoney() - money);
        target.setMoney(target.getMoney() + money);
        dao.update(source);
        // 模拟程序出现异常让事务回滚
        int x = 1 / 0;
        dao.update(target);
    }
}
```

Web层的Servlet调用businessService层的业务方法处理用户请求，需要注意的是：调用businessService层的方法出异常之后，继续将异常抛出，这样在TransactionFilter就能捕获到抛出的异常，继而执行事务回滚操作，如下所示：

```
package me.gacl.web.controller;

import java.io.IOException;
import java.sql.SQLException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import me.gacl.service.AccountService3;

public class AccountServlet extends HttpServlet {

    public void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
```

```
        AccountService3 service = new AccountService3();
        try {
            service.transfer(1, 2, 100);
        } catch (SQLException e) {
            e.printStackTrace();
            //注意：调用service层的方法出异常之后，继续将异常抛出，这样在TransactionFilter就能捕获到抛出的异常，继而执行事务回滚操作
            throw new RuntimeException(e);
        }
    }

    public void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        doGet(request, response);
    }
}
```