# 1. 项目中的事务控制

## 1.1 ThreadLocal相关知识

在讲实际中的事务控制之前，先讲一下ThreadLocal类，这在后面的事务处理中将会用到。

Java文档中说“ThreadLocal提供了线程局部变量”，听起来不知其然，但我们通过下面的对ThreadLocal代码的模拟，就能看出ThreadLocal的含义和作用。

ThreadLocal可以理解为是这样设计的：

|  |
| --- |
| **public class** ThreadLocal {   **private** Map<Runnable,Object> **map** = **new** HashMap<Runnable,Object>();   **public void** set(Object obj){  **map**.put(Thread.*currentThread*(), obj);  }   **public void** remove(){  **map**.remove(Thread.*currentThread*());  }   **public** Object get(){  **return map**.get(Thread.*currentThread*());  } } |

上面的模拟代码很简单，其实意思就是为每个线程（Runnable）提供一个存储对象的地方，通过set和get方法设置和获取对象，通过remove方法移除对象。其特性就是每个线程只能获得本线程设置过的对象，各个线程之间互不干扰，不会获得其他线程的对象。

真实的ThreadLocal代码和上面模拟的有所不同，但其功能可以这样理解，且方法的使用和上面所讲一致。其实这样看来，ThreadLocal类还是比较简单的，但ThreadLocal类非常重要，能实现线程和数据的绑定，以后会应用到。之所以说“ThreadLocal提供了线程局部变量”，是因为ThreadLocal相当于一个容器，各线程只能拿到自己的局部变量。

下面演示ThreadLocal的特性。

首先创建一个线程类，该类接收LocalThread对象，并在run()方法中使用ThreadLocal设置值：

|  |
| --- |
| **package** com.company;  **public class** MyThread **extends** Thread {   **private** ThreadLocal **threadLocal**;   *// 接收传递过来的ThreadLocal对象* **public** MyThread(ThreadLocal threadLocal) {  **this**.**threadLocal** = threadLocal;  }   @Override  **public void** run() {  *// 使用ThreadLocal对象设置一个局部变量值* **threadLocal**.set(**"B"**);  } } |

下面是主类：

|  |
| --- |
| **public class** ThreadLocalDemo {  **public static void** main(String[] args) {  *// 创建ThreadLocal对象* ThreadLocal threadLocal = **new** ThreadLocal();  *// 在当前主线程中设置一个“A”值。* threadLocal.set(**"A"**);   *// 启动新线程，为了使用同一个ThreadLocal，将ThreadLocal作为参数传进去。* Thread myThread = **new** MyThread(threadLocal);  myThread.start(); *// 开启新线程  // 在本线程中获得ThreadLocal的局部变量。* System.***out***.println(threadLocal.get());  } } |

我们发现，输出的值只会是A，而不是myThread线程设置的B。即主线程是无法获得myThread线程的局部变量的。

## 1.2 项目中实现事务控制

现在我们在实际的基于三层架构的项目中考虑进行事务的处理。我们会一步步达到理想的实现事务控制的目标。

### 1.2.1 第一版的事务控制

在前面简单的实例中都没有考虑如何进行事务控制，如果现在要对转账业务使用事务控制，利用DbUtils应该在三层架构中实现呢？

我们以转账业务为例实现事务的控制，在此“三层架构”项目中，DAO层我们使用DbUtils。

首先搭建三层架构。按照我们之前的说法，先开发domain中的实体类，再开发Service接口，这样在开发时，自己具体做什么会很清晰。

（1）Account实体类（在com.company.domain中）：

|  |
| --- |
| **package** com.company.domain;  **import** java.io.Serializable;  **public class** Account **implements** Serializable {  **private** String **name**;  **private float money**;   **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  }   **public float** getMoney() {  **return money**;  }   **public void** setMoney(**float** money) {  **this**.**money** = money;  }   @Override  **public** String toString() {  **return "Account{"** +  **"name='"** + **name** + **'\''** +  **", money="** + **money** +  **'}'**;  } } |

（2）编写业务接口：

|  |
| --- |
| **package** com.company.service;  **public interface** BusinessService {  */\*\*  \* 转账  \** ***@param srcAccountName*** *转出账户  \** ***@param targetAccountName*** *转入账户  \** ***@param money*** *金额  \*/* **void** transfer(String srcAccountName, String targetAccountName, **float** money); } |

（3）DAO接口

由于要实现事务，DAO层不能把这个转账业务分开，否则不能使用同一个连接对象，将无法实现事务。因此DAO层目前也只能这样写：

|  |
| --- |
| **package** com.company.dao;  **public interface** AccountDao {  *// 转账* **void** transfer(String srcAccountName, String targetAccountName, **float** money); } |

（4）DAO实现（com.company.dao.impl包下）。

如果不考虑事务，则写为：

|  |
| --- |
| **package** com.company.dao.impl;  **import** com.company.dao.AccountDao; **import** com.company.util.DBCPUtil; **import** org.apache.commons.dbutils.QueryRunner;  **import** java.sql.SQLException;  **public class** AccountDaoImpl **implements** AccountDao {  **private** QueryRunner **queryRunner** = **new** QueryRunner(DBCPUtil.*getDataSource*());  @Override  **public void** transfer(String srcAccountName, String targetAccountName, **float** money) {  **try** {  **queryRunner**.update(**"update account set money=money-? where name=?"**, money, srcAccountName); // 付款方金额减少  **queryRunner**.update(**"update account set money=money+? where name=?"**, money, targetAccountName); // 收款方金额增加  } **catch** (SQLException e) {  **throw new** RuntimeException(e);  }  } } |

（5）Service实现（com.company.service.impl包下）。

|  |
| --- |
| **package** com.company.service.impl;  **import** com.company.dao.AccountDao; **import** com.company.dao.impl.AccountDaoImpl; **import** com.company.service.BusinessService;  **public class** BusinessServiceImpl **implements** BusinessService {   **private** AccountDao **accountDao** = **new** AccountDaoImpl();   @Override  **public void** transfer(String srcAccountName, String targetAccountName, **float** money) {  **accountDao**.transfer(srcAccountName, targetAccountName, money);  } } |

（6）测试：

|  |
| --- |
| **package** com.company;  **import** com.company.service.BusinessService; **import** com.company.service.impl.BusinessServiceImpl; **import** org.junit.Test;  **public class** AppTest {  BusinessService **businessService** = **new** BusinessServiceImpl();  *// 测试转账* @Test  **public void** testTransfer() {  **businessService**.transfer(**"张三"**, **"李四"**, 100);  } } |

运行程序即可转账。但上面的DAO层代码并没有处理事务，这是不允许的。如果两次update()方法之间出现异常（可使用代码int i = 1/0;进行模拟），则导致账户金额错误。现在我们要将DAO层改为带有事务控制的。

如何改呢？由于事务的控制需要在同一个连接中，因此我们需要使用query等方法的带有connection参数的重载，只有这样才能使用同一个连接。同时，我们创建QueryRunner对象时也无需传递数据源对象了。所以代码改写为这样：

|  |
| --- |
| **package** com.company.dao.impl;  **import** com.company.dao.AccountDao; **import** com.company.util.DBCPUtil; **import** org.apache.commons.dbutils.DbUtils; **import** org.apache.commons.dbutils.QueryRunner;  **import** java.sql.Connection; **import** java.sql.SQLException;  **public class** AccountDaoImpl **implements** AccountDao {  *// 无数据源的QueryRunner。* **private** QueryRunner **queryRunner** = **new** QueryRunner();   @Override  **public void** transfer(String srcAccountName, String targetAccountName, **float** money) {  Connection connection = **null**;  **try** {  connection = DBCPUtil.*getConnection*(); *// 获得连接* connection.setAutoCommit(**false**); *// 开启事务  // 下面两个方法使用同一个连接对象* **queryRunner**.update(connection, **"update account set money=money-? where name=?"**, money, srcAccountName);  **queryRunner**.update(connection, **"update account set money=money+? where name=?"**, money, targetAccountName);  connection.commit(); *// 提交事务* } **catch** (SQLException e) {  **try** {  DbUtils.*rollback*(connection); *// 回滚事务。这里使用了DbUtils提供的回滚方法，其中实现和我们写的是一样的，也是先判断connection不为null再回滚，效果一样的，这里直接使用DbUtils的方法。* } **catch** (SQLException e1) {  e1.printStackTrace();  }  **throw new** RuntimeException(e);  } **finally** {  **try** {  DbUtils.*close*(connection); *// 这里也直接使用DbUtils提供的close()方法关闭连接。* } **catch** (SQLException e) {  **throw new** RuntimeException(e);  }  }  } } |

上述代码可实现事务控制，即使DAO层有异常也不会导致转账错误。

那么有人会疑惑，难道每执行一次update等方法不是会自动调用close()方法吗？为什么还能使用同一个连接？为什么还要手动调用DbUtils.close()方法关闭连接？

这里需要详细说明一下解疑上述问题。当使用带有数据源的QueryRunner时（即构造函数传递数据源对象），一般调用QueryRunner的不带connection的update等方法，此时在这些方法执行结束时，会每次自动地关闭从数据源获得的连接对象（即放回池中），这样的话，每次调用update等方法，使用的连接可能是不同的，所以没法进行事务控制。当使用不带数据源的QueryRunner对象时（即使用无参构造），一般要调用带有connection参数的update等方法，此时这些方法中并不会自动关闭连接，所以我们能够自己控制使用同一个连接并进行事务等的管理，并且最后我们也要手动调用连接的close方法，而DbUtils提供的close方法只是一个内置的关闭连接的静态方法，可供我们直接使用，其实DbUtils.close()方法内部就是普通的关闭连接的代码。这样就解决了上面了疑问，其实这些都是看DbUtils代码知道的，也比较简单。

总而言之，如果需要进行连接的事务等的控制，就要使用update等带connection的重载，这时自己控制连接；如果没有上面的需求，就使用带有数据源的QueryRunner对象，调用无connection参数的方法，方法自动执行，无需你关心连接资源，也不能进行事务等的控制。

### 1.2.2 第二版的事务控制

但是，第一版的代码是完全不合格的！为什么呢？这时就要考虑分层的真正目的了。分层就是为了让各层职责明确，各司其职。对于DAO来说，其中最好只用来实现基本的CRUD。第一版的例子中，为了实现事务，几乎就是把转账这个业务写到了DAO中。

实际上对于事务的要求，都是实际业务的要求，例如转账这个业务要求必须进行事务控制。因此，在实际开发中，事务控制是应该在Service层中实现的。

现在我们就按照上述的思想，再次修改代码。即在DAO中进行基本的CRUD，Service中实现事务控制。

（1）修改DAO接口。对于转账，DAO只要有查询和更新功能即可，而不再提供转账功能，因此修改为：

|  |
| --- |
| **package** com.company.dao;  **import** com.company.domain.Account;  **public interface** AccountDao {  */\*\*  \* 根据姓名查询账户  \** ***@param accountName*** *姓名  \** ***@return*** *账户数据  \*/* Account findAccountByName(String accountName);   */\*\*  \* 更新账户数据  \** ***@param account*** *账户数据  \*/* **void** updateAccount(Account account); } |

（2）修改DAO实现。如果不考虑上层Service的事务需求，则代码为：

|  |
| --- |
| **package** com.company.dao.impl;  **import** com.company.dao.AccountDao; **import** com.company.domain.Account; **import** com.company.util.DBCPUtil; **import** org.apache.commons.dbutils.QueryRunner; **import** org.apache.commons.dbutils.handlers.BeanHandler;  **import** java.sql.SQLException;  **public class** AccountDaoImpl **implements** AccountDao {   **private** QueryRunner **queryRunner** = **new** QueryRunner(DBCPUtil.*getDataSource*()); // 使用数据源   @Override  **public** Account findAccountByName(String accountName) {  **try** {  **return queryRunner**.query(**"select name, money from account where name=?"**, **new** BeanHandler<Account>(Account.**class**), accountName);  } **catch** (SQLException e) {  **throw new** RuntimeException(e);  }  }   @Override  **public void** updateAccount(Account account) {  **try** {  **queryRunner**.update(**"update account set money=? where name=?"**, account.getMoney(), account.getName());  } **catch** (SQLException e) {  **throw new** RuntimeException(e);  }  } } |

（3）最后修改BusinessServiceImpl代码为：

|  |
| --- |
| **package** com.company.service.impl;  **import** com.company.dao.AccountDao; **import** com.company.dao.impl.AccountDaoImpl; **import** com.company.domain.Account; **import** com.company.service.BusinessService;  **public class** BusinessServiceImpl **implements** BusinessService {   **private** AccountDao **accountDao** = **new** AccountDaoImpl();   @Override  **public void** transfer(String srcAccountName, String targetAccountName, **float** money) {  *// 查询到转账双方信息* Account srcAccount = **accountDao**.findAccountByName(srcAccountName);  Account targetAccount = **accountDao**.findAccountByName(targetAccountName);  *// 更新金额* srcAccount.setMoney(srcAccount.getMoney() - money);  targetAccount.setMoney(targetAccount.getMoney() + money);  *// 更新账户* **accountDao**.updateAccount(srcAccount);  **accountDao**.updateAccount(targetAccount);  } } |

这样编写代码就显得各层职能明确了，更加符合OOP的思想。当然这样还不够，还要考虑为Service实现加上事务控制，否则如果更新一个Account对象后出现异常，那么转账金额也会出错。

要在Service中控制事务，则要让Service方法中执行的所有DAO操作使用同一个connection对象，这样才能控制事务。如何做呢？那么现在想到的办法就是在Service中创建连接对象控制事务，然后DAO层需要使用该连接对象，因此只能在DAO实现类中注入一个Connection对象，通过构造方法传进去。

因此先修改DAO实现：

|  |
| --- |
| **package** com.company.dao.impl;  **public class** AccountDaoImpl **implements** AccountDao {  **private** Connection **connection**;  **private** QueryRunner **queryRunner** = **new** QueryRunner(); *// 使用无参构造    // 注入connection* **public** AccountDaoImpl(Connection connection) {  **this**.**connection** = connection;  }   *// 下面的两个方法都使用带connection参数的重载。* @Override  **public** Account findAccountByName(String accountName) {  **try** {  **return queryRunner**.query(**connection**, **"select name, money from account where name=?"**, **new** BeanHandler<Account>(Account.**class**), accountName);  } **catch** (SQLException e) {  **throw new** RuntimeException(e);  }  }   @Override  **public void** updateAccount(Account account) {  **try** {  **queryRunner**.update(**connection**, **"update account set money=? where name=?"**, account.getMoney(), account.getName());  } **catch** (SQLException e) {  **throw new** RuntimeException(e);  }  } } |

然后把获得连接和事务控制写在Service实现中：

|  |
| --- |
| **package** com.company.service.impl;  **import** com.company.dao.AccountDao; **import** com.company.dao.impl.AccountDaoImpl; **import** com.company.domain.Account; **import** com.company.service.BusinessService; **import** com.company.util.DBCPUtil; **import** org.apache.commons.dbutils.DbUtils;  **import** java.sql.Connection; **import** java.sql.SQLException;  **public class** BusinessServiceImpl **implements** BusinessService {   @Override  **public void** transfer(String srcAccountName, String targetAccountName, **float** money) {  Connection connection = **null**;  **try** {  connection = DBCPUtil.*getConnection*();  connection.setAutoCommit(**false**); *// 开启事务* AccountDao accountDao = **new** AccountDaoImpl(connection); *// 这时AccountDao就在方法中维护了。  // 下面是转账代码* Account srcAccount = accountDao.findAccountByName(srcAccountName);  Account targetAccount = accountDao.findAccountByName(targetAccountName);  srcAccount.setMoney(srcAccount.getMoney() - money);  targetAccount.setMoney(targetAccount.getMoney() + money);  accountDao.updateAccount(srcAccount);  accountDao.updateAccount(targetAccount);  *// 最后提交事务* connection.commit();  } **catch** (Exception e) {  **try** {  DbUtils.*rollback*(connection);  } **catch** (SQLException e1) {  e1.printStackTrace();  }  **throw new** RuntimeException(e);  } **finally** {  **try** {  DbUtils.*close*(connection);  } **catch** (SQLException e) {  **throw new** RuntimeException(e);  }  }  } } |

通过上述方式，实现了在Service中实现事务。测试是有效的。

### 1.2.3 第三版的事务控制

上一版的事务控制还是不好，因为在Service层用到了JDBC的API，而JDBC的API应该只被DAO使用。这样导致Service层代码比较混乱，不利于解耦。如何解决既要在Service中实现事务，又不使用JDBC的API这个矛盾的问题呢？我们需要使用一个中间类TranscactionManager来管理事务，其中用到ThreadLocal对象。

中间类TransactionManager，我们称之为事务管理器类。在类中，我们利用ThreadLocal存储当前线程的连接。由于事务用的是同一个连接对象，那么也需要在同一个线程中使用该对象完成事务控制，因此我们将连接对象与当前的线程进行绑定。这样在事务管理器中提供获得连接、开启事务、提交和回滚、关闭连接等的方法，即使有多个线程同时访问数据库，也不会发生连接对象的混用情况。

因此，事务管理器的代码如下：

|  |
| --- |
| **package** com.company.util;  **import** java.sql.Connection; **import** java.sql.SQLException;  *// 事务管理器。* **public class** TransactionManager {  *// 一个只能存储Connection的ThreadLocal* **private static** ThreadLocal<Connection> *threadLocal* = **new** ThreadLocal<>();   *// 得到与当前线程绑定的连接对象* **public static** Connection getConnection() {  Connection connection = (Connection) *threadLocal*.get();  **if** (connection == **null**) {  *// 如果拿不到连接，就从池中获得一个连接，并绑定到当前线程。* connection = DBCPUtil.*getConnection*();  *threadLocal*.set(connection);  }  **return** connection;  }   *// 开启事务的方法* **public static void** startTransaction() {  Connection connection = *getConnection*(); *// 调用本身的getConnection方法，肯定可以拿到和当前线程绑定的连接* **try** {  connection.setAutoCommit(**false**); *// 开启事务* } **catch** (SQLException e) {  **throw new** RuntimeException(e);  }  }   *// 提交事务的方法* **public static void** commitTransaction() {  Connection connection = *getConnection*();  **try** {  connection.commit();  } **catch** (SQLException e) {  **throw new** RuntimeException(e);  }  }   *// 回滚事务的方法* **public static void** rollbackTransaction() {  Connection connection = *getConnection*();  **try** {  connection.rollback();  } **catch** (SQLException e) {  **throw new** RuntimeException(e);  }  }   *// 关闭连接，释放资源的方法* **public static void** release() {  Connection connection = *getConnection*();  **try** {  connection.close();  *// 关闭后，需要把连接与线程解绑，调用remove方法移除对象即可。  threadLocal*.remove();  } **catch** (SQLException e) {  **throw new** RuntimeException(e);  }  } } |

那么DAO实现类这时需要从TransactionManager获得连接：

|  |
| --- |
| **package** com.company.dao.impl;  **import** com.company.dao.AccountDao; **import** com.company.domain.Account; **import** com.company.util.TransactionManager; **import** org.apache.commons.dbutils.QueryRunner; **import** org.apache.commons.dbutils.handlers.BeanHandler;  **import** java.sql.SQLException;  **public class** AccountDaoImpl **implements** AccountDao {   **private** QueryRunner **queryRunner** = **new** QueryRunner(); *// 使用无参构造  // 这时也无需注入Connection   // 下面使用TransactionManager.getConnection()获得连接。* @Override  **public** Account findAccountByName(String accountName) {  **try** {  **return queryRunner**.query(TransactionManager.*getConnection*(), **"select name, money from account where name=?"**, **new** BeanHandler<Account>(Account.**class**), accountName);  } **catch** (SQLException e) {  **throw new** RuntimeException(e);  }  }   @Override  **public void** updateAccount(Account account) {  **try** {  **queryRunner**.update(TransactionManager.*getConnection*(), **"update account set money=? where name=?"**, account.getMoney(), account.getName());  } **catch** (SQLException e) {  **throw new** RuntimeException(e);  }  } } |

然后Service的实现能够使用和以前一样的结构，只要加上事务的处理即可：

|  |
| --- |
| **package** com.company.service.impl;  **import** com.company.dao.AccountDao; **import** com.company.dao.impl.AccountDaoImpl; **import** com.company.domain.Account; **import** com.company.service.BusinessService; **import** com.company.util.TransactionManager;  **public class** BusinessServiceImpl **implements** BusinessService {  *// 还是在类中维护accountDao* AccountDao **accountDao** = **new** AccountDaoImpl();  @Override  **public void** transfer(String srcAccountName, String targetAccountName, **float** money) {  **try** {  TransactionManager.*startTransaction*(); *// 开始事务  // 下面是转账代码* Account srcAccount = **accountDao**.findAccountByName(srcAccountName);  Account targetAccount = **accountDao**.findAccountByName(targetAccountName);  srcAccount.setMoney(srcAccount.getMoney() - money);  targetAccount.setMoney(targetAccount.getMoney() + money);  **accountDao**.updateAccount(srcAccount);  **accountDao**.updateAccount(targetAccount);  TransactionManager.*commitTransaction*(); *// 提交事务* } **catch** (Exception e) {  TransactionManager.*rollbackTransaction*(); *// 回滚* e.printStackTrace();  } **finally** {  TransactionManager.*release*(); *// 释放资源* }  } } |

代码既简洁又有效，可以实现事务控制。

### 1.2.4 第四版的事务控制

上面第三版基本很好地实现了我们的要求，但是还不够，因为我们的目的是让开发者将主要的精力放在业务的开发中，而不是在每开发一个Service方法时，都要考虑编写事务控制的代码。

如果Service中有多个方法需要添加事务，则每个方法都要重复编写事务的代码。实际上不仅仅是事务的实现，例如要实现统计方法执行时间、在方法执行前后记录日志，都会写这样的重复代码。有没有办法解决这个问题呢？其实有印象的同学马上就能想到我们在Java SE中提到这个问题的解决办法，还是使用代理模式！

使用代理就可以在方法前后扩展功能。现在就使用动态代理，返回一个能在原本service实现对象的前后完成事务控制的代理对象。这里我们使用工厂设计模式，专门用一个BusinessServiceFactory工厂类创建符合要求的Service实现对象。该类提供了getBusinessService(boolean proxy)方法，proxy参数传递true就返回拥有事务处理的代理类，否则返回原实现对象。代码如下：

|  |
| --- |
| **package** com.company.util;  **import** com.company.service.BusinessService; **import** com.company.service.impl.BusinessServiceImpl;  **import** java.lang.reflect.InvocationHandler; **import** java.lang.reflect.Method; **import** java.lang.reflect.Proxy;  **public class** BusinessServiceFactory {  **public static** BusinessService getBusinessService(**boolean** proxy) {  BusinessService businessService = **new** BusinessServiceImpl();  **if** (proxy) {  *// 进行代理，返回代理后的对象* **return** (BusinessService) Proxy.*newProxyInstance*(businessService.getClass().getClassLoader(), businessService.getClass().getInterfaces(), **new** InvocationHandler() {  @Override  **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {  *// 给每个方法加上事务的代码，就是把原来写在Service实现类方法中的事务移到这里  // 在执行原方法的前后执行事务代码* Object rtValue = **null**; *// 调用原来方法的返回值* **try** {  TransactionManager.*startTransaction*(); *// 开始事务  // 下面调用原本Service中的业务方法* rtValue = method.invoke(businessService, args);  TransactionManager.*commitTransaction*(); *// 提交事务* } **catch** (Exception e) {  TransactionManager.*rollbackTransaction*(); *// 回滚* e.printStackTrace();  } **finally** {  TransactionManager.*release*(); *// 释放资源* }  **return** rtValue;  }  });  }  *// proxy为假，直接返回原对象* **return** businessService;  } } |

当然，工厂类也可根据配置文件返回需要的对象，这里就不做更多工作了。此时，Service实现类只需要关注业务即可，那么Service实现类就可以重新“回归”到原来的干净代码：

|  |
| --- |
| **package** com.company.service.impl;  **import** com.company.dao.AccountDao; **import** com.company.dao.impl.AccountDaoImpl; **import** com.company.domain.Account; **import** com.company.service.BusinessService; **import** com.company.util.TransactionManager;  **public class** BusinessServiceImpl **implements** BusinessService {  AccountDao **accountDao** = **new** AccountDaoImpl();  @Override  **public void** transfer(String srcAccountName, String targetAccountName, **float** money) {  *// 这里直接写业务代码* Account srcAccount = **accountDao**.findAccountByName(srcAccountName);  Account targetAccount = **accountDao**.findAccountByName(targetAccountName);  srcAccount.setMoney(srcAccount.getMoney() - money);  targetAccount.setMoney(targetAccount.getMoney() + money);  **accountDao**.updateAccount(srcAccount);  **accountDao**.updateAccount(targetAccount);  } } |

无需再改动其他类，使用时只要用BusinessServiceFactory产生拥有事务处理的代理对象，然后使用该代理对象就“自动”有事务处理的功能：

|  |
| --- |
| **package** com.company;  **import** com.company.service.BusinessService; **import** com.company.util.BusinessServiceFactory; **import** org.junit.Test;  **public class** AppTest {  *// 返回经过事务代理的实现类对象* BusinessService **businessService** = BusinessServiceFactory.*getBusinessService*(**true**);  *// 测试转账* @Test  **public void** testTransfer() {  **businessService**.transfer(**"张三"**, **"李四"**, 100);  } } |

而且如果要扩展Service的功能也方便，比如为Service中方法添加统计执行时间的功能，只要在动态代理中添加功能即可：

|  |
| --- |
| **package** com.company.util; // 省略import **public class** BusinessServiceFactory {  **public static** BusinessService getBusinessService(**boolean** proxy) {  BusinessService businessService = **new** BusinessServiceImpl();  **if** (proxy) {  *// 进行代理，返回代理后的对象* **return** (BusinessService) Proxy.*newProxyInstance*(businessService.getClass().getClassLoader(), businessService.getClass().getInterfaces(), **new** InvocationHandler() {  @Override  **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {  *// 记录开始执行的时间* **long** startMillis = System.*currentTimeMillis*();  Object rtValue = **null**;  **try** {  TransactionManager.*startTransaction*(); *// 开始事务  // 下面调用原本Service中的业务方法* rtValue = method.invoke(businessService, args);  TransactionManager.*commitTransaction*(); *// 提交事务* } **catch** (Exception e) {  TransactionManager.*rollbackTransaction*(); *// 回滚* e.printStackTrace();  } **finally** {  TransactionManager.*release*(); *// 释放资源* }  *// 打印执行的时间* System.***out***.println(**"方法"** + method.getName() + **"执行花费时间："** + (System.*currentTimeMillis*() - startMillis) + **"毫秒"**);  **return** rtValue;  }  });  }  **return** businessService;  } } |

这样，我们的“事务控制”就完全讲完了，最终实现了：各层职责明确、低耦合、仅关注业务、可扩展性高。

# 2. AOP编程简述

AOP全称是“Aspect Oriented Programming”，意思是“面向切面编程”。什么意思呢？以我们上述完成的最终版事务控制来看，其实我们可以把事务、日志和性能统计这些功能看做是方法实现中的一些“横切面”，即以不同的方面来看待方法。方法的切面不是编程的主要关注点，而是编程中的扩展功能。我们希望能灵活地配置这些切面，实现功能的扩展，这就是“面向切面编程”，即AOP。

例如我们事务控制的最终版就是把事务当做方法的一个方面，把它单独用代理做出来了，实现了为Service中的每个方法添加事务控制的切面编程的功能。这样做不仅能针对众多方法统一添加事务控制，添加和删除该功能都很方便。更为重要的是，实现了AOP后，开发者只需关注业务实现点即可，而无需编写事务、日志这些重复性的、和业务关系不密切的这些功能。

如何实现AOP编程呢？其实就是利用代理实现的，代理能修改和扩展原有方法。代理和AOP的知识非常重要，AOP是基于代理实现的，在以后要学习的Spring框架使用中也会有更深刻的应用和体会。

# 附1：JDBC连接SQL Server数据库

步骤如下：

（1）下载适用于SQL Server的JDBC驱动，网址：<https://www.microsoft.com/zh-CN/download/details.aspx?id=11774> 。下载tar.gz压缩包即可。下载前可看看页面上的说明。在“系统要求”中可知道，若使用Java8，则可使用Sqljdbc42.jar包，支持的数据库版本是SQL Server2008——2016。

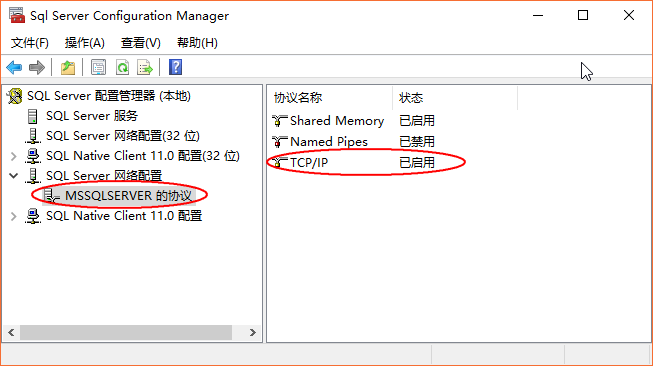
（2）解压压缩包，拿到jre8目录中的jar文件，引用到项目中。

（3）然后即可正常写Java代码。注意的是：数据库连接url格式为：jdbc:sqlserver://IP地址:端口;databaseName=数据库名；需要加载的驱动类为：com.microsoft.sqlserver.jdbc.SQLServerDriver。SQLServer默认端口是1433。

（4）Java代码中使用sa账户和密码验证。这时可成功连接数据库。示例代码：

|  |
| --- |
| Class.*forName*(**"com.microsoft.sqlserver.jdbc.SQLServerDriver"**); Connection conn = DriverManager.*getConnection*(**"jdbc:sqlserver://localhost:1433;databaseName=MyTest"**, **"sa"**, **"123456"**); |

有时不能正常连接数据库，则检查SQL Server服务是否打开了，并且还要保证SQL Server的网络配置正常：打开“Sql Server Configuration Manager”，设置“MSSQLSERVER的协议”中，TCP/IP为“启用”状态，如图：



若使用外界的计算机访问本地数据库，即远程访问，那么本机也需要在防火墙中配置1433端口为“允许入站”，方法和MySQL配置时方法一致。

另外，学习过.Net的人还会使用Windows连接验证方式，这样无需使用密码，但是不推荐这样做，并且这样做较麻烦。步骤如下：

（1）数据库连接url使用Windows验证，即加上“integratedSecurity=true”，比如：“jdbc:sqlserver://localhost:1433;databaseName=MyTest;integratedSecurity=true”。

（2）Java代码中无需写用户名和密码，即：“DriverManager.getConnection(url)”即可。

（3）仅仅上述步骤会出错。还需要向Windows系统注册一个“sqljdbc\_auth.dll”的文件。该文件在压缩包的auth文件夹中已经提供了。根据自己电脑位数选择合适的版本注册。有两种注册的方法：1.把该dll所在文件夹添加为Path环境变量；2.把该dll文件添加到“C:/Windows/System32”文件夹中。

另外，连接SQL Server数据库时，也能使用DBUtils组件和连接池技术（DBCP等）。只是需要注意下述问题：

使用DBUtils时，SQL Server不能正确识别处理预编译的SQL语句，需要在创建QueryRunner对象时，提供一个knowParaType参数，此参数类型是boolean，设置为true即可。比如：

QueryRunner qr = new QueryRunner(true)；

当使用数据库连接池提供的DataSource数据源时，也别忘了提供此参数：

QueryRunner qr = new QueryRunner(ds, true);