# springAOP

AOP来控制事务提交！！！！！！！！

本课我们使用的是JDBCTemplete对象进行增删改查？

JDBCTemplete不是框架？为什么不用框架？ 因为还没学

connection的进化版 ：JDBCTemplete这个对象提供了增删改查方法

等同于给了我们一个工具类，进行数据库表的增删改查

------------------------学习JDBCTemplete的增删改查------------------------

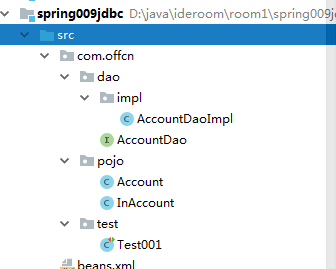
1：beans.xml配置如下：

<**bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"**>  
 <**property name="driverClassName" value="com.mysql.jdbc.Driver"**></**property**>  
 <**property name="url" value="jdbc:mysql://localhost:3306/aop001?useUnicode=true&amp;characterEncoding=utf8"**></**property**>  
 <**property name="username" value="root"**></**property**>  
 <**property name="password" value="root"**></**property**>  
  
 *<!-- 连接池启动时的初始值 -->* <**property name="initialSize" value="1"**/>  
 *<!-- 连接池的最大值 -->* <**property name="maxActive" value="500"**/>  
 *<!-- 最大空闲值.当经过一个高峰时间后，连接池可以慢慢将已经用不到的连接慢慢释放一部分，一直减少到maxIdle为止 -->* <**property name="maxIdle" value="2"**/>  
 *<!-- 最小空闲值.当空闲的连接数少于该值时，连接池就会预申请一些连接，以避免洪峰来时再申请而造成的性能开销 -->* <**property name="minIdle" value="1"**/>  
</**bean**>

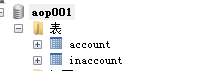
<**bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate"**>  
 *<!-- dataSource:数据源，数据连接池 -->* <**property name="dataSource" ref="dataSource"**></**property**>  
</**bean**>

<**bean id="accountDao" class="com.offcn.dao.impl.AccountDaoImpl"**>  
 <**property name="jdbcTemplate" ref="jdbcTemplate"**></**property**>  
</**bean**>

文件结构



数据库：



dao实现类中



---------------------------------------------准备完毕之后。。。进行增删改查。。。。。。

增

@Override  
**public boolean** saveUser(InAccount inAccount) {  
  
 String sql = **"insert into inaccount values (?,?)"**;  
  
 Object[] objects = {inAccount.getAccountId(),inAccount.getInBalance()};  
  
 **int** update = **jdbcTemplate**.update(sql, objects);  
  
 **return** update>0;  
}

删

@Override  
**public void** deleteUserById(String id) {  
  
  
 String sql = **"delete from account where accountid = ?"**;  
  
 **jdbcTemplate**.update(sql, id);  
  
 System.***out***.println(**"执行删除完毕。。。"**);  
  
}

改

@Override  
**public void** updateUser(Account account) {  
  
  
 String sql = **"update account set balance = ? where accountid = ?"**;  
  
 Object [] objects = {account.getBalance(),account.getAccountid()};  
  
 **jdbcTemplate**.update(sql,objects );  
  
}

查

查单个

@Override  
**public** Account finUserById(String id) {  
  
  
 String sql = **"select** *\** **from account where accountid = ?"**;  
 Object [] objects = {id};  
  
 */\*alt + 回车\*/* RowMapper<Account> rowMapper = **new** RowMapper<Account>() {  
 @Override  
 **public** Account mapRow(ResultSet resultSet, **int** i) **throws** SQLException {  
  
 System.***out***.println(**"i=="**+i);  
  
 Account account = **new** Account();  
 account.setAccountid(resultSet.getString(**"accountid"**));  
 account.setBalance(resultSet.getDouble(**"balance"**));  
 **return** account;  
 }  
 };  
  
 Account account = **jdbcTemplate**.queryForObject(sql, rowMapper, objects);  
 **return** account;  
}

查多个

@Override  
**public** List<Account> fincUserAll() {  
  
 String sql = **"select** *\** **from account"**;  
 */\*  
 resultSet是查询到的结果集  
 resultSet可以将结果集给调出来，resultSet.getString() resultSet.getDouble()  
 \*/* RowMapper<Account> rowMapper = **new** RowMapper<Account>() {  
 @Override  
 **public** Account mapRow(ResultSet resultSet, **int** i) **throws** SQLException {  
 System.***out***.println(**"i=="**+i);  
 Account account = **new** Account();  
 account.setAccountid(resultSet.getString(**"accountid"**));  
 account.setBalance(resultSet.getDouble(**"balance"**));  
 **return** account;  
 }  
 };  
 List<Account> query = **jdbcTemplate**.query(sql, rowMapper);  
  
 **return** query;  
}

本课学习重点内容如下案例所示（AOP）

-- 案例 来看一下，没有事务提交会发生什么情况

按理逻辑如下：

两个用户： Account（银行账户） InAccount（流水账户）

accountid 银行id acocuntid 银行id

balance 余额 inBalance 存入了多少钱

需求： 存钱

逻辑： saveAccount（InAccount inAccount）

配置文件：

<**bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"**>  
 <**property name="driverClassName" value="com.mysql.jdbc.Driver"**></**property**>  
 <**property name="url" value="jdbc:mysql://localhost:3306/aop001?useUnicode=true&amp;characterEncoding=utf8"**></**property**>  
 <**property name="username" value="root"**></**property**>  
 <**property name="password" value="root"**></**property**>  
  
 *<!-- 连接池启动时的初始值 -->* <**property name="initialSize" value="1"**/>  
 *<!-- 连接池的最大值 -->* <**property name="maxActive" value="500"**/>  
 *<!-- 最大空闲值.当经过一个高峰时间后，连接池可以慢慢将已经用不到的连接慢慢释放一部分，一直减少到maxIdle为止 -->* <**property name="maxIdle" value="2"**/>  
 *<!-- 最小空闲值.当空闲的连接数少于该值时，连接池就会预申请一些连接，以避免洪峰来时再申请而造成的性能开销 -->* <**property name="minIdle" value="1"**/>  
</**bean**>  
  
  
  
 <**bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate"**>  
 <**property name="dataSource" ref="dataSource"**></**property**>  
 </**bean**>  
  
  
 <**bean id="accountDao" class="com.com.offcn.dao.impl.AccountDaoimpl"**>  
 <**property name="jdbcTemplate" ref="jdbcTemplate"**></**property**>  
 </**bean**>  
  
<**bean id="inAccountDao" class="com.com.offcn.dao.impl.InAccountDaoImpl"**>  
 <**property name="jdbcTemplate" ref="jdbcTemplate"**></**property**>  
</**bean**>  
  
 <**bean id="accountService" class="com.com.offcn.service.impl.AccountServiceImpl"**>  
 <**property name="accountDao" ref="accountDao"**></**property**>  
 <**property name="inAccountDao" ref="inAccountDao"**></**property**>  
 </**bean**>

service类中的代码：

@Override  
 **public void** saveAccount(InAccount inAccount) {  
*//√ 存钱 》》》 流水账户：直接存* **inAccountDao**.saveInAccount(inAccount);  
*// √ 银行账户》》》查询（根据id来查询是哪一个银行1001,1002）* Account account = **accountDao**.findAccountById(inAccount.getAccountId());  
*// 更改： 将新的account账户更新到数据库* **double** v = account.getBalance() + inAccount.getInBalance();  
*// 将以前的账户余额加上现在存的钱* account.setBalance(v);  
*//进行更新操作 √* **accountDao**.updateAccountByNewAccount(account);  
 }

通过我们的测试我们发现，在查询方法发生异常的时候，会导致我们的数据库出现数据错误，

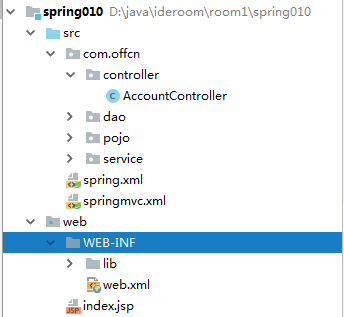
所以我们需要配置AOP事务

二：加上事务提交 配置事务提交，如下所示

*<!--事务管理器-->*<**bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager"**>  
 <**property name="dataSource" ref="dataSource"**></**property**>  
</**bean**>  
  
  
*<!--事务通知-->* <**tx:advice id="advice" transaction-manager="transactionManager"**>  
 <**tx:attributes**>  
 <**tx:method name="save\*" propagation="REQUIRED" isolation="DEFAULT" read-only="false"**/>  
 <**tx:method name="delete\*" propagation="REQUIRED" isolation="DEFAULT" read-only="false"**/>  
 <**tx:method name="find\*" propagation="REQUIRED" isolation="DEFAULT" read-only="true"**/>  
 <**tx:method name="update\*" propagation="REQUIRED" isolation="DEFAULT"**/>  
 <**tx:method name="\*" propagation="REQUIRED" isolation="DEFAULT"**/>  
 </**tx:attributes**>  
 </**tx:advice**>

## springmvc和spring整合延伸

项目结构：



### web.xml

<**servlet**>  
 <**servlet-name**>springmvc</**servlet-name**>  
 <**servlet-class**>org.springframework.web.servlet.DispatcherServlet</**servlet-class**>  
 <**init-param**>  
 <**param-name**>contextConfigLocation</**param-name**>  
 <**param-value**>classpath:springmvc.xml</**param-value**>  
 </**init-param**>  
 <**multipart-config**>  
 </**multipart-config**>  
</**servlet**>  
<**servlet-mapping**>  
 <**servlet-name**>springmvc</**servlet-name**>  
 <**url-pattern**>/</**url-pattern**>  
</**servlet-mapping**>  
  
  
*<!-- spring配置文件加载 -->*<**context-param**>  
 <**param-name**>contextConfigLocation</**param-name**>  
 <**param-value**>classpath:spring.xml</**param-value**>  
</**context-param**>

*<!-- spring配置文件加载 必须要有监听器，如果没有监听器是无法初始化的，无法加载spring-->*  
<**listener**>  
 <**listener-class**>org.springframework.web.context.ContextLoaderListener</**listener-class**>  
</**listener**>

### spring.xml中的配置

<**context:component-scan base-package="com.offcn"**></**context:component-scan**>  
  
<**aop:aspectj-autoproxy** ></**aop:aspectj-autoproxy**>  
  
*<!--事务管理-->*<**bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager"**>  
 <**property name="dataSource" ref="dataSource"**></**property**>  
</**bean**>  
  
<**tx:annotation-driven transaction-manager="transactionManager"** ></**tx:annotation-driven**>  
  
  
<**bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"**>  
 <**property name="driverClassName" value="com.mysql.jdbc.Driver"**></**property**>  
 <**property name="url" value="jdbc:mysql://localhost:3306/aop001?useUnicode=true&amp;characterEncoding=utf8"**></**property**>  
 <**property name="username" value="root"**></**property**>  
 <**property name="password" value="root"**></**property**>  
  
 *<!-- 连接池启动时的初始值 -->* <**property name="initialSize" value="1"**/>  
 *<!-- 连接池的最大值 -->* <**property name="maxActive" value="500"**/>  
 *<!-- 最大空闲值.当经过一个高峰时间后，连接池可以慢慢将已经用不到的连接慢慢释放一部分，一直减少到maxIdle为止 -->* <**property name="maxIdle" value="2"**/>  
 *<!-- 最小空闲值.当空闲的连接数少于该值时，连接池就会预申请一些连接，以避免洪峰来时再申请而造成的性能开销 -->* <**property name="minIdle" value="1"**/>  
</**bean**>  
  
  
 <**bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate"**>  
 <**property name="dataSource" ref="dataSource"**></**property**>  
 </**bean**>

### springmvc.xml中的配置

<**context:component-scan base-package="com.offcn.controller"**>

</**context:component-scan**>

### controller层

*//注入*@Controller  
**public class** AccountController {  
 @Autowired  
 AccountService **accountService**;  
 @RequestMapping(**"saveAccount"**)  
 **public** String saveAccount(InAccount inAccount , Model model){  
 System.***out***.println(**"接收的数据为："**+inAccount);  
 **accountService**.saveAccount(inAccount);  
 model.addAttribute(**"msg"**,**"执行完毕"**);  
 **return "index.jsp"**;  
}  
}

service实现层  
@Service  
**public class** AccountServiceImpl **implements** AccountService {  
  
 @Autowired  
 **private** AccountDao **accountDao**;  
  
 @Autowired  
 **private** InAccountDao **inAccountDao**;  
  
 @Override  
 @Transactional(propagation = Propagation.***REQUIRED***,isolation = Isolation.***DEFAULT***,readOnly = **false**)  
 **public void** saveAccount(InAccount inAccount) {  
  
 **inAccountDao**.saveInAccount(inAccount);  
  
  
 System.***out***.println(9/0);  
 Account account = **accountDao**.findAccountById(inAccount.getAccountId());  
  
 **double** v = account.getBalance() + inAccount.getInBalance();  
 account.setBalance(v);  
  
 **accountDao**.updateAccountByNewAccount(account);  
  
 }  
}

### dao实现：accountDaoImpl

@Repository  
**public class** AccountDaoimpl **implements** AccountDao {  
  
 @Resource(name = **"jdbcTemplate"**)  
 **private** JdbcTemplate **jdbcTemplate**;  
  
  
 @Override  
 **public** Account findAccountById(String accountId) {  
  
*// System.out.println("------异常开始！！！！");  
// System.out.println(9/0);* System.***out***.println(**"------异常开始1111！！！！"**);  
 String sql =**"select** *\** **from account where accountid = ?"**;  
 System.***out***.println(**"------异常开始2222！！！！"**);  
  
 RowMapper<Account> rowMapper = **new** RowMapper<Account>() {  
 @Override  
 **public** Account mapRow(ResultSet resultSet, **int** i) **throws** SQLException {  
  
 Account account = **new** Account();  
 account.setAccountid(resultSet.getString(**"accountid"**));  
 account.setBalance(resultSet.getDouble(**"balance"**));  
  
 **return** account;  
 }  
 };  
  
 Account account = **jdbcTemplate**.queryForObject(sql, rowMapper, accountId);  
 System.***out***.println(**"------异常开始33333！！！！"**);  
  
 **return** account;  
 }  
  
 @Override  
 **public void** updateAccountByNewAccount(Account account) {  
  
 String sql =**"update account set balance = ? where accountid = ?"**;  
 Object [] objects = {account.getBalance(),account.getAccountid()};  
 **jdbcTemplate**.update(sql,objects);  
  
 }  
}

dao实现：InAccountDaoImpl  
@Repository  
**public class** InAccountDaoImpl **implements** InAccountDao {  
  
 @Resource(name = **"jdbcTemplate"**)  
 **private** JdbcTemplate **jdbcTemplate**;  
  
 @Override  
 **public void** saveInAccount(InAccount inAccount) {  
  
 System.***out***.println(**"流水账户执行保存inaccount操作"**);  
 String sql = **"insert into inaccount values (?,?)"**;  
 Object [] objects = {inAccount.getAccountId(),inAccount.getInBalance()};  
 **jdbcTemplate**.update(sql,objects);  
  
 }  
}

发送请求：

<http://localhost:8080/saveAccount?accountId=1001&inBalance=2.88>

注意：

springmvc.xml中的扫描必须扫描controller层

<context:component-scan base-package="com.offcn.controller">

</context:component-scan>

spring 的扫描，可以扫描其余的层

<context:component-scan base-package="com.offcn">

</context:component-scan>

数据库数据隔离级别的例子

<https://baijiahao.baidu.com/s?id=1611918898724887602&wfr=spider&for=pc>