# Spring4第四天

内容回顾：

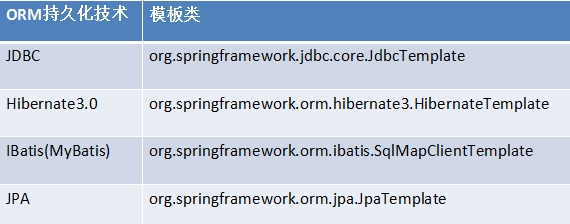
1. AOP：增强，原理：jdk+cglib
2. AOP代码的编写流程
   1. 编写目标类
   2. 编写切面类、通知方法
   3. 配置目标类、切面类
   4. 配置AOP
      1. 配置切入点
      2. 配置通知
3. 切入点表达式：execution(\* cn.itcast.service.impl.\*.\*(..))
4. 通知类型
   1. 前置通知
   2. 后置通知
   3. 环绕通知
   4. 异常通知
   5. 最终通知

5、JdbcTemplate

# Spring中的Jdbc模板

## Jdbc模板概述

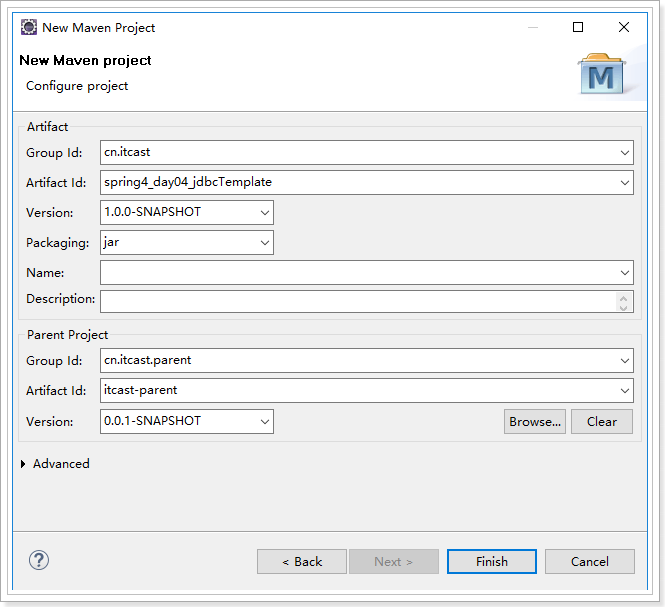
它是spring框架中提供的一个对象，**是对原始Jdbc API对象的简单封装**。spring框架为我们提供了很多的操作模板类，入下图所示：



我们今天的主角在**spring-jdbc-4.24.RELEASE.jar**中，我们在导包的时候，除了要导入这个jar包外，还需要导入一个**spring-tx-4.2.4.RELEASE.jar**（它是和事务相关的）。

## Spring中的Jdbc模板入门

### 创建工程，引入依赖



需要引入spring-jdbc,,druid,mysql驱动的坐标，pom.xml内容如下：

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>cn.itcast.parent</groupId>

<artifactId>itcast-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<groupId>cn.itcast</groupId>

<artifactId>spring4\_day04\_jdbcTemplate</artifactId>

<version>1.0.0-SNAPSHOT</version>

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-test</artifactId>

</dependency>

<!-- jdbc模板 -->

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-jdbc</artifactId>**

**</dependency>**

**<dependency>**

**<groupId>com.alibaba</groupId>**

**<artifactId>druid</artifactId>**

**</dependency>**

**<dependency>**

**<groupId>mysql</groupId>**

**<artifactId>mysql-connector-java</artifactId>**

**</dependency>**

</dependencies>

</project>

### 创建测试表

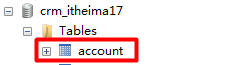
CREATE TABLE account(

id BIGINT PRIMARY KEY AUTO\_INCREMENT,

NAME VARCHAR(40),

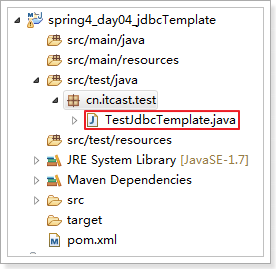
money DOUBLE

)CHARACTER SET utf8 COLLATE utf8\_general\_ci;



### 创建测试类

* 创建单元测试类TestJdbcTemplate，在其中创建test1方法



**public** **class** TestJdbcTemplate {

@Test

**public** **void** test1(){

// 创建jdbc模板对象

JdbcTemplate jdbcTemplate = **new** JdbcTemplate();

// 创建druid数据源,其它数据源也行，例如c3p0

DruidDataSource dataSource = **new** DruidDataSource();

dataSource.setDriverClassName("com.mysql.jdbc.Driver");

dataSource.setUrl("jdbc:mysql://localhost:3306/mybatis");

dataSource.setUsername("root");

dataSource.setPassword("123456");

// 设置数据源

jdbcTemplate.setDataSource(dataSource);

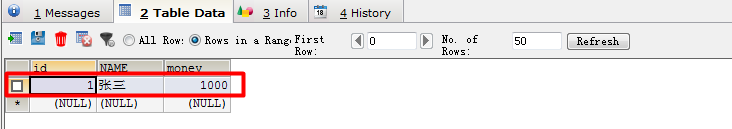
// 插入操作

jdbcTemplate.update("insert into account(name,money) values(?,?)", "张三", 1000.0);

}

}

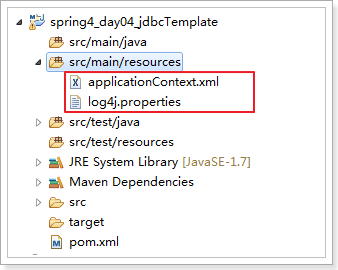
运行test1方法发现：向account表中插入了一条数据



### 将JdbcTemplate交给Spring管理

上面的程序中，我们是手动创建JdbcTemplate对象的。现在，可以把JdbcTemplate交给Spring来管理。

创建applicaitonContext.xml，并引入log4j.properteis



在applicationContext.xml中配置JdbcTemplate,需要注入数据源：

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd"*>

<!-- 配置JdbcTemplate -->

<bean id=*"jdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>

<!-- 注入数据源 -->

**<property name=*"dataSource"* ref=*"dataSource"*></property>**

</bean>

<!-- 配置数据源 -->

<bean id=*"dataSource"* class=*"com.alibaba.druid.pool.DruidDataSource"*>

<property name=*"driverClassName"* value=*"com.mysql.jdbc.Driver"*></property>

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/mybatis"*></property>

<property name=*"username"* value=*"root"*></property>

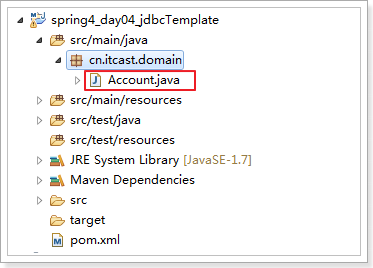
<property name=*"password"* value=*"123456"*></property>

</bean>

</beans>

### 在DAO中使用JdbcTemplate

* 创建Account实体类



**public** **class** Account **implements** Serializable{

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** Long id;

**private** String name;

**private** Double money;

**public** Long getId() {

**return** id;

}

**public** **void** setId(Long id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** Double getMoney() {

**return** money;

}

**public** **void** setMoney(Double money) {

**this**.money = money;

}

@Override

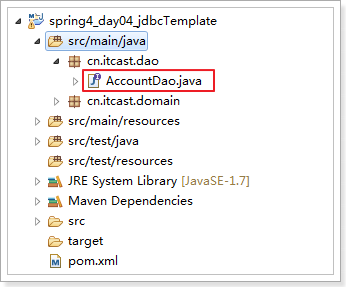
**public** String toString() {

**return** "Account [id=" + id + ", name=" + name + ", money=" + money + "]";

}

}

* 创建AccountDao接口



**public** **interface** AccountDao {

/\*\*

\* 账户保存

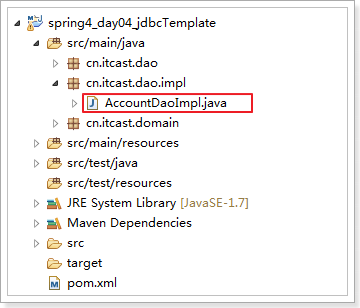
\* **@param** account

\*/

**public** **void** save(Account account);

}

* 创建AccountDaoImpl实现类，在其中声明JdbcTemplate属性，并提供set方法



**public** **class** AccountDaoImpl **implements** AccountDao {

**private JdbcTemplate jdbcTemplate;**

**public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {**

**this.jdbcTemplate = jdbcTemplate;**

**}**

@Override

**public** **void** save(Account account) {

**this**.jdbcTemplate.update("insert into account(name,money) values(?,?)",account.getName(),account.getMoney());

}

}

### 把JdbcTemplate注入给DAO

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd"*>

<bean id=*"accountDao"* class=*"cn.itcast.dao.impl.AccountDaoImpl"*>

**<property name=*"jdbcTemplate"* ref=*"jdbcTemplate"*></property>**

</bean>

<bean id=*"jdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>

<property name=*"dataSource"* ref=*"dataSource"*></property>

</bean>

<bean id=*"dataSource"* class=*"com.alibaba.druid.pool.DruidDataSource"*>

<property name=*"driverClassName"* value=*"com.mysql.jdbc.Driver"*></property>

<property name=*"url"* value=*"jdbc:mysql://localhost:3306/mybatis"*></property>

<property name=*"username"* value=*"root"*></property>

<property name=*"password"* value=*"123456"*></property>

</bean>

</beans>

### 编写测试类

修改TestJdbcTemplate，在该类上添加Spring整合Junit单元测试注解,创建单元测试方法test2:

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration("classpath:applicationContext.xml")

**public** **class** TestJdbcTemplate {

@Autowired

**private** AccountDao accountDao;

@Test

**public** **void** test1(){

// 创建jdbc模板对象

JdbcTemplate jdbcTemplate = **new** JdbcTemplate();

// 创建druid数据源,其它数据源也行，例如c3p0

DruidDataSource dataSource = **new** DruidDataSource();

dataSource.setDriverClassName("com.mysql.jdbc.Driver");

dataSource.setUrl("jdbc:mysql://localhost:3306/mybatis");

dataSource.setUsername("root");

dataSource.setPassword("123456");

// 设置数据源

jdbcTemplate.setDataSource(dataSource);

// 插入操作

jdbcTemplate.update("insert into account(name,money) values(?,?)", "张三", 1000.0);

}

/\*\*

\* 测试保存

\*/

@Test

**public** **void** test2(){

Account account = **new** Account();

account.setName("JAY");

account.setMoney(1000.0);

accountDao.save(account);

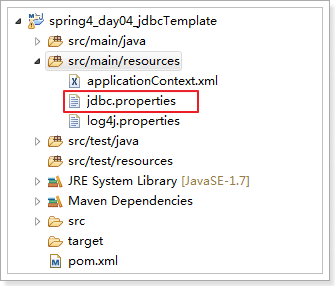
}

}

## 将数据库连接信息保存到属性文件中

必要性：之前，咱们都是把与数据库连接相关的信息直接写在applicationContext.xml中，当想要修改数据库连接信息时，就得改applicationContext.xml，很麻烦。可以把与数据库连接相关的信息写在jdbc.properties文件中。

### 新建jdbc.properties属性文件



jdbc.properties内容如下：

jdbc.driverClass=com.mysql.jdbc.Driver

jdbc.url=jdbc:mysql://localhost:3306/mybatis

jdbc.username=root

jdbc.password=123456

### 在applicationContext.xml中引入jdbc.properties文件

在applicationContext.xml中引入jdbc.properties文件的方式有如下两种：

* 方式一：配置bean引入，配置占位符解析器来从jdbc.properties中取值

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:context=*"http://www.springframework.org/schema/context"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context.xsd"*>

<!-- 配置bean引入jdbc.properties -->

**<bean**

**class=*"org.springframework.context.support.PropertySourcesPlaceholderConfigurer"*>**

**<property name=*"location"* value=*"classpath:jdbc.properties"*></property>**

**</bean>**

<bean id=*"accountDao"* class=*"cn.itcast.dao.impl.AccountDaoImpl"*>

<property name=*"jdbcTemplate"* ref=*"jdbcTemplate"*></property>

</bean>

<bean id=*"jdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>

<property name=*"dataSource"* ref=*"dataSource"*></property>

</bean>

<bean id=*"dataSource"* class=*"com.alibaba.druid.pool.DruidDataSource"*>

<property name=*"driverClassName"* value=*"${jdbc.driverClass}"*></property>

<property name=*"url"* value=*"${jdbc.url}"*></property>

<property name=*"username"* value=*"${jdbc.username}"*></property>

<property name=*"password"* value=*"${jdbc.password}"*></property>

</bean>

</beans>

* 方式二：通过context标签引入，还得引入context约束

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:context=*"http://www.springframework.org/schema/context"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context.xsd"*>

<!-- 通过context标签引入jdbc.properties -->

**<context:property-placeholder location=*"classpath:jdbc.properties"* />**

<bean id=*"accountDao"* class=*"cn.itcast.dao.impl.AccountDaoImpl"*>

<property name=*"jdbcTemplate"* ref=*"jdbcTemplate"*></property>

</bean>

<bean id=*"jdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>

<property name=*"dataSource"* ref=*"dataSource"*></property>

</bean>

<bean id=*"dataSource"* class=*"com.alibaba.druid.pool.DruidDataSource"*>

<property name=*"driverClassName"* value=*"${jdbc.driverClass}"*></property>

<property name=*"url"* value=*"${jdbc.url}"*></property>

<property name=*"username"* value=*"${jdbc.username}"*></property>

<property name=*"password"* value=*"${jdbc.password}"*></property>

</bean>

</beans>

## 使用Jdbc模板完成CRUD

### 在AccountDao接口中编写删除、修改、查询方法

修改AccountDao接口，如下：

**public** **interface** AccountDao {

/\*\*

\* 保存账户

\* **@param** account

\*/

**public** **void** save(Account account);

/\*\*

\* 更新账户

\*/

**public** **void** update(Account account);

/\*\*

\* 根据id删除账户

\* **@param** id

\*/

**public** **void** delete(Long id);

/\*\*

\* 查询某个账户的余额

\*/

**public** Double findMoney(Long id);

/\*\*

\* 根据id查询账户对象

\*/

**public** Account findById(Long id);

/\*\*

\* 查询所有账户

\*/

**public** List<Account> findAll();

}

### 在AccountDaoImpl中实现update方法

**public** **class** AccountDaoImpl **implements** AccountDao {

**private** JdbcTemplate jdbcTemplate;

**public** **void** setJdbcTemplate(JdbcTemplate jdbcTemplate) {

**this**.jdbcTemplate = jdbcTemplate;

}

@Override

**public** **void** save(Account account) {

jdbcTemplate.update("insert into account(name,money) values(?,?)",account.getName(),account.getMoney());

}

**@Override**

**public void update(Account account) {**

**jdbcTemplate.update("update account set name = ?,money = ? where id = ?",account.getName(),account.getMoney(),account.getId());**

**}**

@Override

**public** **void** delete(Long id) {

}

@Override

**public** Double findMoney(Long id) {

**return** null;

}

@Override

**public** Account findById(Long id) {

**return** null;

}

@Override

**public** List<Account> findAll() {

**return** null;

}

}

### 在AccountDaoImpl中实现delete方法

**public** **class** AccountDaoImpl **implements** AccountDao {

**private** JdbcTemplate jdbcTemplate;

**public** **void** setJdbcTemplate(JdbcTemplate jdbcTemplate) {

**this**.jdbcTemplate = jdbcTemplate;

}

@Override

**public** **void** save(Account account) {

jdbcTemplate.update("insert into account(name,money) values(?,?)",account.getName(),account.getMoney());

}

@Override

**public** **void** update(Account account) {

jdbcTemplate.update("update account set name = ?,money = ? where id = ?",account.getName(),account.getMoney(),account.getId());

}

**@Override**

**public void delete(Long id) {**

**jdbcTemplate.update("delete from account where id = ?",id);**

**}**

@Override

**public** Account findById(Long id) {

**return** null;

}

@Override

**public** List<Account> findAll() {

**return** null;

}

}

### 在AccountDaoImpl中实现查询方法

#### 在AccountDaoImpl中实现findMoney方法

**public** **class** AccountDaoImpl **implements** AccountDao {

**private** JdbcTemplate jdbcTemplate;

**public** **void** setJdbcTemplate(JdbcTemplate jdbcTemplate) {

**this**.jdbcTemplate = jdbcTemplate;

}

@Override

**public** **void** save(Account account) {

jdbcTemplate.update("insert into account(name,money) values(?,?)",account.getName(),account.getMoney());

}

@Override

**public** **void** update(Account account) {

jdbcTemplate.update("update account set name = ?,money = ? where id = ?",account.getName(),account.getMoney(),account.getId());

}

@Override

**public** **void** delete(Long id) {

jdbcTemplate.update("delete from account where id = ?",id);

}

/\*\*

\* queryForObject的参数

\* sql:查询语句

\* requiredType:查询的结果的类型

\* 占位符的值

\*/

**@Override**

**public Double findMoney(Long id) {**

**Double money = jdbcTemplate.queryForObject("select money from account where id = ?", Double.class,id);**

**return money;**

**}**

@Override

**public** Account findById(Long id) {

**return** null;

}

@Override

**public** List<Account> findAll() {

**return** null;

}

}

queryForObject方法说明：

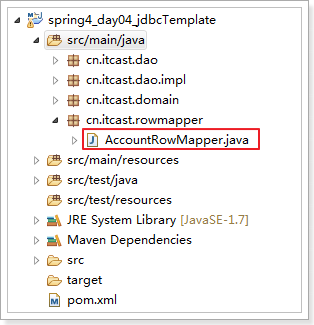
参数1：sql语句

参数2：查询的结果类型

参数3：占位符的值

#### 在AccountDaoImpl中实现findById方法

* 创建RowMapper，RowMapper就是把通过jdbc查询出来的一行数据转变成Java对象的“映射器”



**public** **class** AccountRowMapper **implements** RowMapper<Account>{

/\*\*

\* 如何把账户表里的一行数据形成账户对象

\*/

@Override

**public** Account mapRow(ResultSet rs, **int** rownum) **throws** SQLException {

Account account = **new** Account();

account.setId(rs.getLong("id"));

account.setName(rs.getString("name"));

account.setMoney(rs.getDouble("money"));

**return** account;

}

}

* 在AccountDaoImpl中实现findById方法如下：

**public** **class** AccountDaoImpl **implements** AccountDao {

**private** JdbcTemplate jdbcTemplate;

**public** **void** setJdbcTemplate(JdbcTemplate jdbcTemplate) {

**this**.jdbcTemplate = jdbcTemplate;

}

@Override

**public** **void** save(Account account) {

jdbcTemplate.update("insert into account(name,money) values(?,?)",account.getName(),account.getMoney());

}

@Override

**public** **void** update(Account account) {

jdbcTemplate.update("update account set name = ?,money = ? where id = ?",account.getName(),account.getMoney(),account.getId());

}

@Override

**public** **void** delete(Long id) {

jdbcTemplate.update("delete from account where id = ?",id);

}

/\*\*

\* queryForObject的参数

\* sql:查询语句

\* requiredType:查询的结果的类型

\* 占位符的值

\*/

@Override

**public** Double findMoney(Long id) {

Double money = jdbcTemplate.queryForObject("select money from account where id = ?", Double.**class**,id);

**return** money;

}

**@Override**

**public Account findById(Long id) {**

**Account account = jdbcTemplate.queryForObject("select \* from account where id = ?", new AccountRowMapper(), id);**

**return account;**

**}**

@Override

**public** List<Account> findAll() {

**return** null;

}

}

#### 在AccountDaoImpl中实现findAll方法

**public** **class** AccountDaoImpl **implements** AccountDao {

**private** JdbcTemplate jdbcTemplate;

**public** **void** setJdbcTemplate(JdbcTemplate jdbcTemplate) {

**this**.jdbcTemplate = jdbcTemplate;

}

@Override

**public** **void** save(Account account) {

jdbcTemplate.update("insert into account(name,money) values(?,?)",account.getName(),account.getMoney());

}

@Override

**public** **void** update(Account account) {

jdbcTemplate.update("update account set name = ?,money = ? where id = ?",account.getName(),account.getMoney(),account.getId());

}

@Override

**public** **void** delete(Long id) {

jdbcTemplate.update("delete from account where id = ?",id);

}

/\*\*

\* queryForObject的参数

\* sql:查询语句

\* requiredType:查询的结果的类型

\* 占位符的值

\*/

@Override

**public** Double findMoney(Long id) {

Double money = jdbcTemplate.queryForObject("select money from account where id = ?", Double.**class**,id);

**return** money;

}

@Override

**public** Account findById(Long id) {

Account account = jdbcTemplate.queryForObject("select \* from account where id = ?", **new** AccountRowMapper(), id);

**return** account;

}

**@Override**

**public List<Account> findAll() {**

**List<Account> list = jdbcTemplate.query("select \* from account", new AccountRowMapper());**

**return list;**

**}**

}

query方法说明:

参数1：sql语句

参数2：RowMapper对象

## 在DAO中使用JdbcTemplate的两种方式

在DAO中使用JdbcTemplate有有两种方式：

1. 直接在DAO中声明JdbcTemplate。以上程序都是采用的这种方式；
2. 让DAO继承JdbcDaoSupport；

### 方式一：在DAO中直接注入JdbcTemplate

#### 编写DAO,注入JdbcTemplate

示例代码：

**public** **class** AccountDaoImpl **implements** AccountDao {

**private JdbcTemplate jdbcTemplate;**

**public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {**

**this.jdbcTemplate = jdbcTemplate;**

**}**

@Override

**public** **void** save(Account account) {

jdbcTemplate.update("insert into account(name,money) values(?,?)", account.getName(),account.getMoney());

}

}

#### 把DAO配置到Spring中

配置AccountDaoImpl，注入JdbcTemplate

示例代码：

<bean id=*"accountDao"* class=*"cn.itcast.dao.impl.AccountDaoImpl"*>

**<property name=*"jdbcTemplate"* ref=*"jdbcTemplate"*></property>**

</bean>

<!-- 配置JdbcTemplate -->

<bean id=*"jdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>

<property name=*"dataSource"* ref=*"dataSource"*></property>

</bean>

### 方式二：在DAO中使用JdbcDaoSupport

#### 让DAO继承JdbcDaoSupport

* 修改AccountDaoImpl，让该类继承JdbcDaoSupport,不用在声明jdbcTemplate属性了，因为在其父类中已经声明了

**public** **class** AccountDaoImpl **extends JdbcDaoSupport** **implements** AccountDao {

@Override

**public** **void** save(Account account) {

**this**.getJdbcTemplate().update("insert into account(name,money) values(?,?)",account.getName(),account.getMoney());

}

@Override

**public** **void** update(Account account) {

**this**.getJdbcTemplate().update("update account set name=?,money=? where id = ?",account.getName(),account.getMoney(),account.getId());

}

@Override

**public** **void** delete(Long id) {

**this**.getJdbcTemplate().update("delete from account where id = ?",id);

}

@Override

**public** Double findMoney(Long id) {

Double money = **this**.getJdbcTemplate().queryForObject("select money from account where id = ?", Double.**class**, id);

**return** money;

}

@Override

**public** Account findById(Long id) {

AccountRowMapper mapper = **new** AccountRowMapper();

Account account = **this**.getJdbcTemplate().queryForObject("select \* from account where id = ?", mapper, id);

**return** account;

}

@Override

**public** List<Account> findAll() {

AccountRowMapper mapper = **new** AccountRowMapper();

List<Account> list = **this**.getJdbcTemplate().query("select \* from account", mapper);

**return** list;

}

}

#### 给DAO注入DataSource

配置AccountDaoImpl，并注入DataSource，不需要在配置JdbcTemplate了。

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:context=*"http://www.springframework.org/schema/context"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context.xsd"*>

<!-- 通过context标签引入jdbc.properties -->

<context:property-placeholder location=*"classpath:jdbc.properties"* />

<bean id=*"accountDao"* class=*"cn.itcast.dao.impl.AccountDaoImpl"*>

**<property name=*"dataSource"* ref=*"dataSource"*></property>**

</bean>

<bean id=*"dataSource"* class=*"com.alibaba.druid.pool.DruidDataSource"*>

<property name=*"driverClassName"* value=*"${jdbc.driverClass}"*></property>

<property name=*"url"* value=*"${jdbc.url}"*></property>

<property name=*"username"* value=*"${jdbc.username}"*></property>

<property name=*"password"* value=*"${jdbc.password}"*></property>

</bean>

</beans>

#### JdbcTemplate源码分析



**思考：**

**两版Dao有什么区别呢？**

**答案：**

**第一种在Dao类中定义JdbcTemplate的方式，适用于所有配置方式（xml和注解都可以）。**

**第二种让Dao继承JdbcDaoSupport的方式，只能用于基于XML的方式，注解用不了。**

# Spring中的事务控制（重点）

## 事务的回顾

* 事务的概念
  + 事务是逻辑上一组操作，组成这组操作各个逻辑单元，要么一起成功，要么一起失败。
* 事务的特性
  + 原子性
  + 一致性
  + 隔离性
  + 持久性
* 事物并发引起一些读的问题
  + - 脏读
    - 不可重复读
    - 虚读（幻读）
* 解决读问题
  + 设置事务隔离级别
    - read uncommitted
    - **read committed**
    - **repeatable read**
    - Serializable

## Spring事务管理的API介绍(了解)

### PlatformTransactionManager

此接口是spring的平台事务管理器，它里面提供了我们常用的操作事务的方法，如下图：



我们在开发中都是使用它的实现类，如下图：



**真正管理事务的对象**

**org.springframework.jdbc.datasource.DataSourceTransactionManager 使用Spring JDBC或iBatis 进行持久化数据时使用**

**org.springframework.orm.hibernate3.HibernateTransactionManager 使用Hibernate版本进行持久化数据时使用**

### TransactionDefinition

它是事务的定义信息对象，里面有如下方法：



#### 事务的隔离级别



#### 事务的传播行为（重点）

REQUIRED:如果当前没有事务，就新建一个事务，如果已经存在一个事务中，加入到这个事务中。一般的选择（默认值）

REQUERS\_NEW:新建事务，如果当前在事务中，把当前事务挂起。

SUPPORTS:支持当前事务，如果当前没有事务，就以非事务方式执行（没有事务）

MANDATORY：使用当前的事务，如果当前没有事务，就抛出异常

NOT\_SUPPORTED:以非事务方式执行操作，如果当前存在事务，就把当前事务挂起

NEVER:以非事务方式运行，如果当前存在事务，抛出异常

NESTED:如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行REQUIRED类似的操作。

#### 超时时间

默认值是-1，没有超时限制。如果有，以秒为单位进行设置。

#### 事务是否只读(重点)

建议查询时设置为只读。

### TransactionStatus

此接口提供的是事务具体的运行状态，方法介绍如下图：



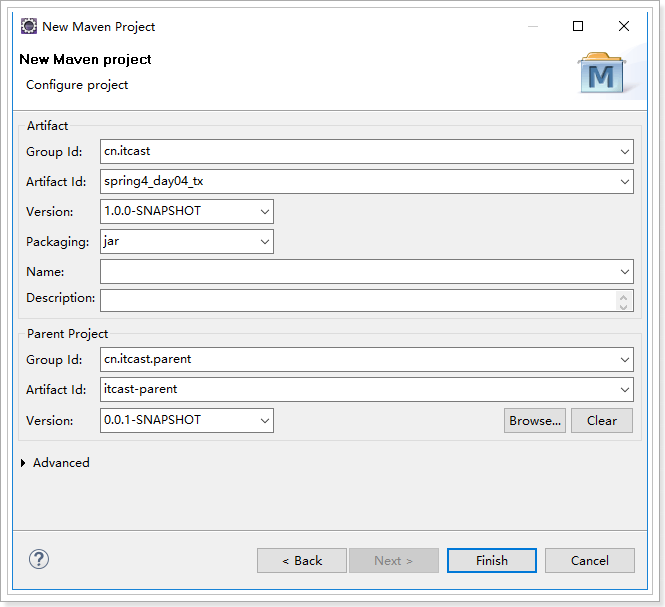
**小结：Spring框架进行事务的管理，首先使用TransactionDefinition对事务进行定义。通过PlatformTransactionManager根据TransactionDefinition的定义信息进行事务的管理。在事务管理过程中产生一系列的状态：保存到TransactionStatus中。**

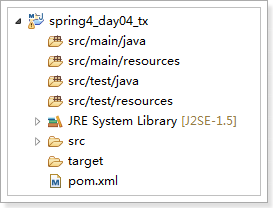
## 基于xml的声明式事务控制（重点掌握）

原理：AOP面向切面编程

### 环境搭建

#### 创建工程，引入依赖





Spring声明式事务控制的采用的是AOP思想，所以需要引入与AOP相关的依赖。

总共需要引入的依赖有：

1. ioc的依赖；
2. aop的依赖
3. jdbc模板+数据库连接池的依赖
4. spring整合junit单元测试的依赖

完整的pom.xml如下：

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>cn.itcast.parent</groupId>

<artifactId>itcast-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<groupId>cn.itcast</groupId>

<artifactId>spring4\_day04\_tx</artifactId>

<version>1.0.0-SNAPSHOT</version>

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-test</artifactId>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aspects</artifactId>

</dependency>

<!-- jdbc模板 -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

</dependency>

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>druid</artifactId>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

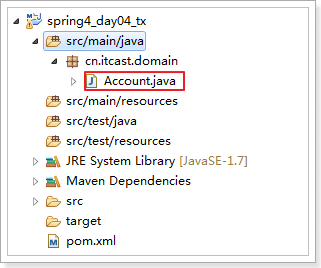
</dependency>

</dependencies>

</project>

#### 编写相关类

* 编写账户Account实体类



**public** **class** Account **implements** Serializable{

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** Long id;

**private** String name;

**private** Double money;

**public** Long getId() {

**return** id;

}

**public** **void** setId(Long id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** Double getMoney() {

**return** money;

}

**public** **void** setMoney(Double money) {

**this**.money = money;

}

@Override

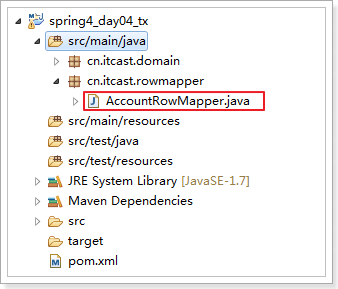
**public** String toString() {

**return** "Account [id=" + id + ", name=" + name + ", money=" + money + "]";

}

}

* 编写AccountRowMapper类，把表中的一行数据形成账户对象



**public** **class** AccountRowMapper **implements** RowMapper<Account>{

/\*\*

\* 如何把账户表里的一行数据形成账户对象

\*/

@Override

**public** Account mapRow(ResultSet rs, **int** row) **throws** SQLException {

Account account = **new** Account();

account.setId(rs.getLong("id"));

account.setName(rs.getString("name"));

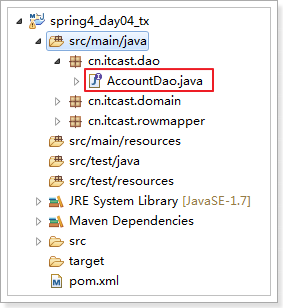
account.setMoney(rs.getDouble("money"));

**return** account;

}

}

* 编写AccountDao接口



**public** **interface** AccountDao {

/\*\*

\* 根据id查询账户对象

\* **@param** id

\* **@return**

\*/

**public** Account findById(Long id);

/\*\*

\* 更新账户对象

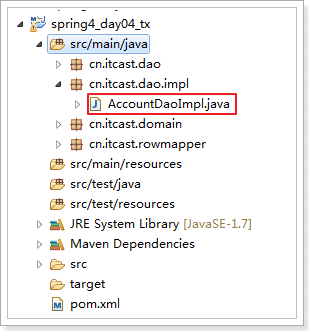
\* **@param** account

\*/

**public** **void** update(Account account);

}

* 编写AccountDaoImpl实现类，继承JdbcDaoSupport



**public** **class** AccountDaoImpl **extends** JdbcDaoSupport **implements** AccountDao {

@Override

**public** **void** update(Account account) {

**this**.getJdbcTemplate().update("update account set name = ?,money = ? where id = ?",account.getName(),account.getMoney(),account.getId());

}

@Override

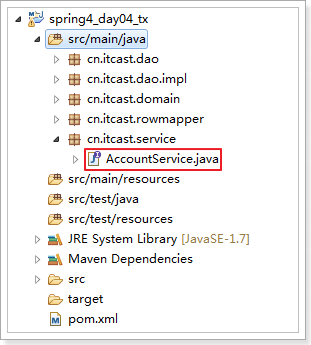
**public** Account findById(Long id) {

**return** **this**.getJdbcTemplate().queryForObject("select \* from account where id = ?", **new** AccountRowMapper(), id);

}

}

* 编写AccountService接口



**public** **interface** AccountService {

/\*\*

\* 业务层：转账方法

\* **@param** fromId 转出账户id

\* **@param** toId 转入账户id

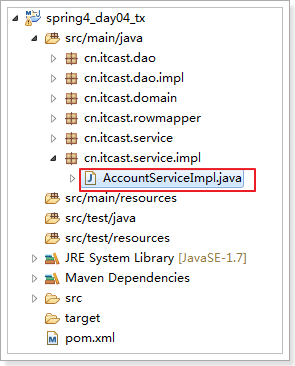
\* **@param** money 转账金额

\*/

**public** **void** transfer(Long fromId,Long toId,Double money);

}

* 编写AccountServiceImpl实现类



**public** **class** AccountServiceImpl **implements** AccountService{

**private** AccountDao accountDao;

**public** **void** setAccountDao(AccountDao accountDao) {

**this**.accountDao = accountDao;

}

@Override

**public** **void** transfer(Long fromId, Long toId, Double money) {

//查询转出账户

Account fromAccount = accountDao.findById(fromId);

//查询转入账户

Account toAccount = accountDao.findById(toId);

//转出账户减钱

fromAccount.setMoney(fromAccount.getMoney() - money);

//转入账户加钱

toAccount.setMoney(toAccount.getMoney() + money);

//更新转出账户

accountDao.update(fromAccount);

//更新转入账户

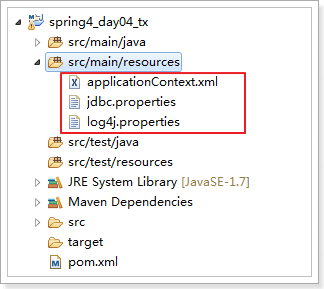
accountDao.update(toAccount);

}

}

#### 把类交给Spring来管理

* 创建applicationContext.xml，并引入jdbc.properties,log4j.properties



注意：需要新建jdbc.properties文件

jdbc.driverClass=com.mysql.jdbc.Driver

jdbc.url=jdbc:mysql://localhost:3306/mybatis

jdbc.username=root

jdbc.password=123456

applicationContext.xml内容如下：

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context.xsd"*>

<context:property-placeholder location=*"classpath:jdbc.properties"*/>

<bean id=*"accountService"* class=*"cn.itcast.service.impl.AccountServiceImpl"*>

<property name=*"accountDao"* ref=*"accountDao"*></property>

</bean>

<bean id=*"accountDao"* class=*"cn.itcast.dao.impl.AccountDaoImpl"*>

<property name=*"dataSource"* ref=*"dataSource"*></property>

</bean>

<bean id=*"dataSource"* class=*"com.alibaba.druid.pool.DruidDataSource"*>

<property name=*"driverClassName"* value=*"${jdbc.driverClass}"*></property>

<property name=*"url"* value=*"${jdbc.url}"*></property>

<property name=*"username"* value=*"${jdbc.username}"*></property>

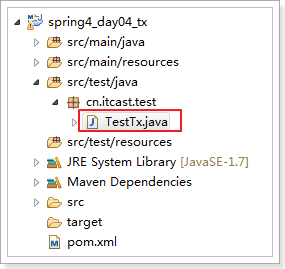
<property name=*"password"* value=*"${jdbc.password}"*></property>

</bean>

</bean>

#### 编写测试类测试转账方法

* 创建单元测试类TestTx,在其中创建test1单元测试方法测试转账方法



@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration("classpath:applicationContext.xml")

**public** **class** TestTx {

@Autowired

**private** AccountService accountService;

@Test

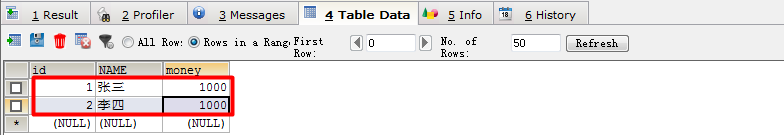
**public** **void** test1(){

accountService.transfer(1L, 2L, 100.0);

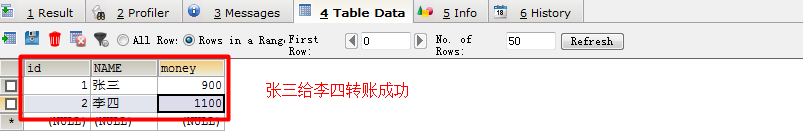
}

}

运行test1方法之前数据库中的数据：



运行test1方法之后数据库中的数据：



但此时，并不代表我们service中的方法设计是合理的。下面，我们在CustomerServiceImpl中的transfer方法中造一个异常出来，然后观察转账的结果是否正确。在CustomerServiceImpl中的transfer方法中造一个异常的代码如下：

**public** **class** AccountServiceImpl **implements** AccountService{

**private** AccountDao accountDao;

**public** **void** setAccountDao(AccountDao accountDao) {

**this**.accountDao = accountDao;

}

@Override

**public** **void** transfer(Long fromId, Long toId, Double money) {

//查询转出账户

Account fromAccount = accountDao.findById(fromId);

//查询转入账户

Account toAccount = accountDao.findById(toId);

//转出账户减钱

fromAccount.setMoney(fromAccount.getMoney() - money);

//转入账户加钱

toAccount.setMoney(toAccount.getMoney() + money);

//更新转出账户

accountDao.update(fromAccount);

**int i = 10 / 0;**

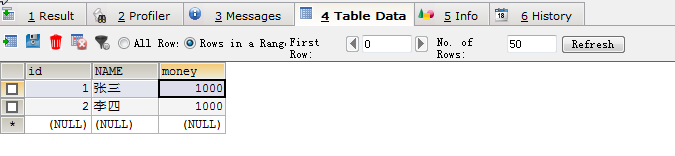
//更新转入账户

accountDao.update(toAccount);

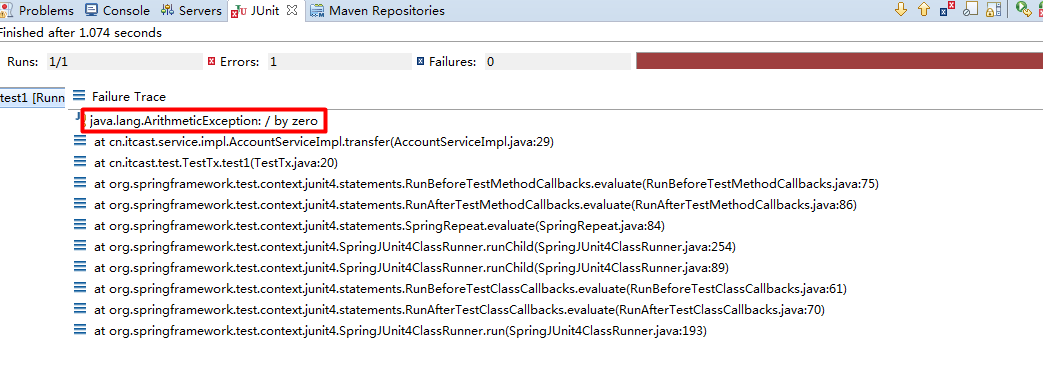
}

}

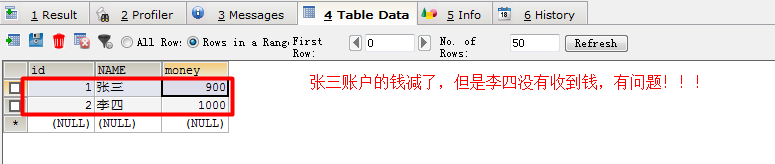
测试之前，先把数据库中account表的数据还原：



再次运行TestTx中的test1方法，控制台结果如下：



数据库account表中的数据如下：



所以，需要把transfer方法套在一个事务里，transfer中的所有操作要么全部成功，要么全部取消。下面，采用spring中AOP的思想来管理事务，我们叫spring的声明式事务控制。

### 在Spring中配置事务

#### 配置事务管理器

在applicationContext.xml中配置事务管理器**DataSourceTransactionManager**

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context.xsd"*>

<context:property-placeholder location=*"classpath:jdbc.properties"*/>

<bean id=*"accountService"* class=*"cn.itcast.service.impl.AccountServiceImpl"*>

<property name=*"accountDao"* ref=*"accountDao"*></property>

</bean>

<bean id=*"accountDao"* class=*"cn.itcast.dao.impl.AccountDaoImpl"*>

<property name=*"dataSource"* ref=*"dataSource"*></property>

</bean>

<bean id=*"dataSource"* class=*"com.alibaba.druid.pool.DruidDataSource"*>

<property name=*"driverClassName"* value=*"${jdbc.driverClass}"*></property>

<property name=*"url"* value=*"${jdbc.url}"*></property>

<property name=*"username"* value=*"${jdbc.username}"*></property>

<property name=*"password"* value=*"${jdbc.password}"*></property>

</bean>

<!-- 配置事务管理器 -->

**<bean id=*"transactionManager"* class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>**

<!-- 注入数据源 -->

**<property name=*"dataSource"* ref=*"dataSource"*></property>**

**</bean>**

</bean>

#### 配置事务属性

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:tx=*"http://www.springframework.org/schema/tx"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context.xsd*

*http://www.springframework.org/schema/tx*

*http://www.springframework.org/schema/tx/spring-tx.xsd"*>

<context:property-placeholder location=*"classpath:jdbc.properties"*/>

<bean id=*"accountService"* class=*"cn.itcast.service.impl.AccountServiceImpl"*>

<property name=*"accountDao"* ref=*"accountDao"*></property>

</bean>

<bean id=*"accountDao"* class=*"cn.itcast.dao.impl.AccountDaoImpl"*>

<property name=*"dataSource"* ref=*"dataSource"*></property>

</bean>

<bean id=*"dataSource"* class=*"com.alibaba.druid.pool.DruidDataSource"*>

<property name=*"driverClassName"* value=*"${jdbc.driverClass}"*></property>

<property name=*"url"* value=*"${jdbc.url}"*></property>

<property name=*"username"* value=*"${jdbc.username}"*></property>

<property name=*"password"* value=*"${jdbc.password}"*></property>

</bean>

<!-- 配置事务管理器 -->

<bean id=*"transactionManager"* class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>

<!-- 注入数据源 -->

<property name=*"dataSource"* ref=*"dataSource"*></property>

</bean>

<!-- 配置事务的属性

id:指定advice的id，后边要用到

transaction-manager:写的是事务管理器的id

-->

**<tx:advice id=*"txAdvice"* transaction-manager=*"transactionManager"*>**

**<tx:attributes>**

**<!-- find开头的方法加只读事务 ，\*表示通配符，匹配任意方法-->**

**<tx:method name=*"find\*"* read-only=*"true"*/>**

**<!-- 其余方法是加可读写的事务 -->**

**<tx:method name=*"\*"*/>**

**</tx:attributes>**

**</tx:advice>**

</bean>

#### 配置事务切面

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:aop=*"http://www.springframework.org/schema/aop"*

xmlns:tx=*"http://www.springframework.org/schema/tx"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context.xsd*

*http://www.springframework.org/schema/tx*

*http://www.springframework.org/schema/tx/spring-tx.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop.xsd"*>

<context:property-placeholder location=*"classpath:jdbc.properties"*/>

<bean id=*"accountService"* class=*"cn.itcast.service.impl.AccountServiceImpl"*>

<property name=*"accountDao"* ref=*"accountDao"*></property>

</bean>

<bean id=*"accountDao"* class=*"cn.itcast.dao.impl.AccountDaoImpl"*>

<property name=*"dataSource"* ref=*"dataSource"*></property>

</bean>

<bean id=*"dataSource"* class=*"com.mchange.v2.c3p0.ComboPooledDataSource"*>

<property name=*"driverClass"* value=*"${jdbc.driverClass}"*></property>

<property name=*"jdbcUrl"* value=*"${jdbc.url}"*></property>

<property name=*"user"* value=*"${jdbc.username}"*></property>

<property name=*"password"* value=*"${jdbc.password}"*></property>

</bean>

<!-- 配置事务管理器 -->

<bean id=*"transactionManager"* class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>

<property name=*"dataSource"* ref=*"dataSource"*></property>

</bean>

<!-- 配置事务的属性

id:指定advice的id，后边要用到

transaction-manager:写的是事务管理器的id

-->

<tx:advice id=*"txAdvice"* transaction-manager=*"transactionManager"*>

<tx:attributes>

<!-- find开头的方法加只读事务 ，\*表示通配符，匹配任意-->

<tx:method name=*"find\*"* read-only=*"true"*/>

<!-- 其余方法是加可读写的事务 -->

<tx:method name=*"\*"*/>

</tx:attributes>

</tx:advice>

<!-- 配置事务切面 -->

<aop:config>

<!-- 配置切入点表达式：告诉框架哪些方法要控制事务 -->

<aop:pointcut expression=*"execution(\* cn.itcast.service.impl.\*.\*(..))"* id=*"pt"*/>

<!--将定义好的事务属性应用到上述的切入点 -->

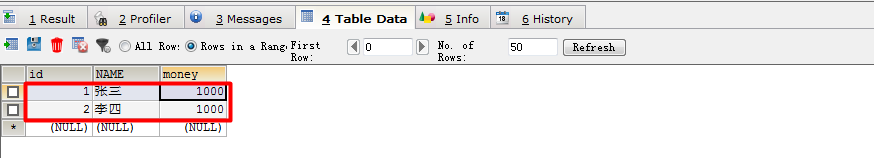
<aop:advisor advice-ref=*"txAdvice"* pointcut-ref=*"pt"*/>

</aop:config>

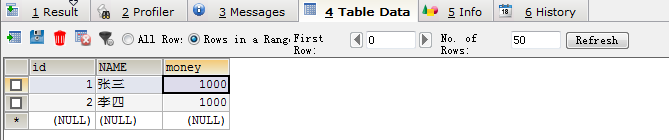
</beans>

#### 测试事务是否成功

* 首先，把数据库中account表的数据还原：



* 然后，再次运行TestTx中的test1方法，控制台依然会报“被零除异常”，但是表中的数据没有发生变化：张三的钱没有转，李四的钱也没有变化。



#### 测试只读事务

* 在AcountService接口中增加一个findById方法：根据id查询账户对象

**public** **interface** AccountService {

/\*\*

\* 业务层：转账方法

\* **@param** fromId 转出账户id

\* **@param** toId 转入账户id

\* **@param** money 转账金额

\*/

**public** **void** transfer(Long fromId,Long toId,Double money);

/\*\*

\* 根据id查询账户对象

\* **@param** id

\* **@return**

\*/

**public** Account findById(Long id);

}

* 在AccountServiceImpl中实现findById方法

**public** **class** AccountServiceImpl **implements** AccountService{

**private** AccountDao accountDao;

**public** **void** setAccountDao(AccountDao accountDao) {

**this**.accountDao = accountDao;

}

@Override

**public** **void** transfer(Long fromId, Long toId, Double money) {

//查询转出账户

Account fromAccount = accountDao.findById(fromId);

//查询转入账户

Account toAccount = accountDao.findById(toId);

//转出账户减钱

fromAccount.setMoney(fromAccount.getMoney() - money);

//转入账户加钱

toAccount.setMoney(toAccount.getMoney() + money);

//更新转出账户

accountDao.update(fromAccount);

**int i = 10 / 0;**

//更新转入账户

accountDao.update(toAccount);

}

@Override

**public** Account findById(Long id) {

Account account = accoutDao.findById(id);

/\*

\* 应该是只能做查询操作，但现在做了修改操作，是不被允许的.

\* 怎么才能不允许在查询方法中，做增删改操作呢？给该方法加只读事务

\*/

account.setMoney(100000.0);

accoutDao.update(account);

**return** account;

}

}

* 在单元测试类TestTx中创建第二个单元测试方法test2,测试findById方法

@RunWith(SpringJUnit4ClassRunner.**class**)

@ContextConfiguration("classpath:applicationContext.xml")

**public** **class** TestTx {

@Autowired

**private** AccountService accountService;

@Test

**public** **void** test1(){

accountService.transfer(1L, 2L, 100.0);

}

@Test

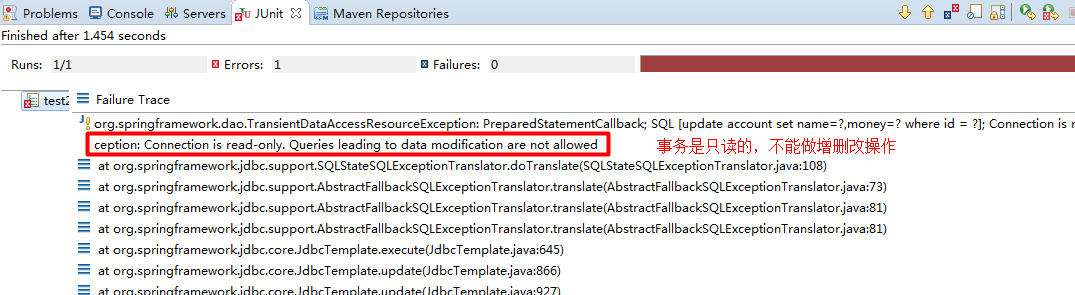
**public** **void** test2(){

accountService.findById(1L);

}

}

* 运行test2方法，发现控制台报错，异常信息如下：



## 基于注解的声明式事务控制（重点掌握）

### 环境搭建

#### 创建工程，引入jar包

同xml方式

#### 编写相关类

同xml方式

#### 开启Spring对注解的支持

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:aop=*"http://www.springframework.org/schema/aop"*

xmlns:tx=*"http://www.springframework.org/schema/tx"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context.xsd*

*http://www.springframework.org/schema/tx*

*http://www.springframework.org/schema/tx/spring-tx.xsd*

*http://www.springframework.org/schema/aop*

[*http://www.springframework.org/schema/aop/spring-aop.xsd"*>](http://www.springframework.org/schema/aop/spring-aop.xsd%22%3e)

<!-- 开启spring注解扫描 -->

**<context:component-scan base-package=*"cn.itcast"*></context:component-scan>**

<context:property-placeholder location=*"classpath:jdbc.properties"*/>

<bean id=*"dataSource"* class=*"com.alibaba.druid.pool.DruidDataSource"*>

<property name=*"driverClassName"* value=*"${jdbc.driverClass}"*></property>

<property name=*"url"* value=*"${jdbc.url}"*></property>

<property name=*"username"* value=*"${jdbc.username}"*></property>

<property name=*"password"* value=*"${jdbc.password}"*></property>

</bean>

</bean>

#### 在相关类上加注解

**注意：因为采用的是注解配置，此处不能继承JdbcDaoSupport，只能在DAO中声明JdbcTemplate属性**

@Repository("accountDao")

**public** **class** AccountDaoImpl **implements** AccountDao {

@Autowired

**private** JdbcTemplate jdbcTemplate;

@Override

**public** **void** update(Account account) {

**this**.jdbcTemplate.update("update account set name = ?,money = ? where id = ?",account.getName(),account.getMoney(),account.getId());

}

@Override

**public** Account findById(Long id) {

**return** **this**.jdbcTemplate.queryForObject("select \* from account where id = ?", **new** AccountRowMapper(), id);

}

}

@Service("accountService")

**public** **class** AccountServiceImpl **implements** AccountService{

@Autowired

**private** AccountDao accountDao;

@Override

**public** **void** transfer(Long fromId, Long toId, Double money) {

//查询转出账户

Account fromAccount = accountDao.findById(fromId);

//查询转入账户

Account toAccount = accountDao.findById(toId);

//转出账户减钱

fromAccount.setMoney(fromAccount.getMoney() - money);

//转入账户加钱

toAccount.setMoney(toAccount.getMoney() + money);

//更新转出账户

accountDao.update(fromAccount);

**int** i = 10 / 0;

//更新转入账户

accountDao.update(toAccount);

}

}

### 配置步骤

#### 第一步：配置事务管理器

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:aop=*"http://www.springframework.org/schema/aop"*

xmlns:tx=*"http://www.springframework.org/schema/tx"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context.xsd*

*http://www.springframework.org/schema/tx*

*http://www.springframework.org/schema/tx/spring-tx.xsd*

*http://www.springframework.org/schema/aop*

[*http://www.springframework.org/schema/aop/spring-aop.xsd"*>](http://www.springframework.org/schema/aop/spring-aop.xsd%22%3e)

<!-- 开启spring注解扫描 -->

<context:component-scan base-package=*"cn.itcast"*></context:component-scan>

<context:property-placeholder location=*"classpath:jdbc.properties"*/>

<bean id=*"dataSource"* class=*"com.alibaba.druid.pool.DruidDataSource"*>

<property name=*"driverClassName"* value=*"${jdbc.driverClass}"*></property>

<property name=*"url"* value=*"${jdbc.url}"*></property>

<property name=*"username"* value=*"${jdbc.username}"*></property>

<property name=*"password"* value=*"${jdbc.password}"*></property>

</bean>

<!-- 注意：采用注解来配置，一定要显示定义JdbcTemplate，因为在dao中如要注入JdbcTemplate -->

**<bean id=*"jdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>**

**<property name=*"dataSource"* ref=*"dataSource"*></property>**

**</bean>**

<!-- 配置事务管理器 -->

**<bean id=*"transactionManager"* class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>**

**<property name=*"dataSource"* ref=*"dataSource"*></property>**

**</bean>**

</beans>

#### 第二步：开启Spring对注解事务的支持

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:aop=*"http://www.springframework.org/schema/aop"* xmlns:tx=*"http://www.springframework.org/schema/tx"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop.xsd*

*http://www.springframework.org/schema/tx*

[*http://www.springframework.org/schema/tx/spring-tx.xsd"*>](http://www.springframework.org/schema/tx/spring-tx.xsd%22%3e)

<!-- 开启spring注解扫描 -->

<context:component-scan base-package=*"cn.itcast"*></context:component-scan>

<!-- 开启事务注解的支持

transaction-manager:写事务管理器的id

-->

**<tx:annotation-driven transaction-manager=*"transactionManager"*/>**

<context:property-placeholder location=*"classpath:jdbc.properties"*/>

<bean id=*"dataSource"* class=*"com.alibaba.druid.pool.DruidDataSource"*>

<property name=*"driverClassName"* value=*"${jdbc.driverClass}"*></property>

<property name=*"url"* value=*"${jdbc.url}"*></property>

<property name=*"username"* value=*"${jdbc.username}"*></property>

<property name=*"password"* value=*"${jdbc.password}"*></property>

</bean>

<bean id=*"jdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>

<property name=*"dataSource"* ref=*"dataSource"*></property>

</bean>

<!-- 配置事务管理器 -->

<bean id=*"transactionManager"* class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>

<property name=*"dataSource"* ref=*"dataSource"*></property>

</bean>

#### 第三步：在业务层加@Transactional注解

@Service("accountService")

**@Transactional**//该类中所有的方法都加可读写的事务

**public** **class** AccountServiceImpl **implements** AccountService{

@Autowired

**private** AccountDao accountDao;

@Override

**public** **void** transfer(Long fromId, Long toId, Double money) {

//查询转出账户

Account fromAccount = accountDao.findById(fromId);

//查询转入账户

Account toAccount = accountDao.findById(toId);

//转出账户减钱

fromAccount.setMoney(fromAccount.getMoney() - money);

//转入账户加钱

toAccount.setMoney(toAccount.getMoney() + money);

//更新转出账户

accountDao.update(fromAccount);

**int** i = 10 / 0;

//更新转入账户

accountDao.update(toAccount);

}

}

#### 第四步：在find开头的方法加只读事务

* 在AccountService接口中添加方法

**public** **interface** AccountService {

/\*\*

\* 业务层：转账方法

\* **@param** fromId 转出账户id

\* **@param** toId 转入账户id

\* **@param** money 转账金额

\*/

**public** **void** transfer(Long fromId,Long toId,Double money);

**public void findById(Long id);**

}

* 在AccountServiceImpl中实现方法

@Service("accountService")

@Transactional//该类中所有的方法都加可读写的事务

**public** **class** AccountServiceImpl **implements** AccountService{

@Autowired

**private** AccountDao accountDao;

@Override

**public** **void** transfer(Long fromId, Long toId, Double money) {

//查询转出账户

Account fromAccount = accountDao.findById(fromId);

//查询转入账户

Account toAccount = accountDao.findById(toId);

//转出账户减钱

fromAccount.setMoney(fromAccount.getMoney() - money);

//转入账户加钱

toAccount.setMoney(toAccount.getMoney() + money);

//更新转出账户

accountDao.update(fromAccount);

**int** i = 10 / 0;

//更新转入账户

accountDao.update(toAccount);

}

@Override

**@Transactional(readOnly=true)//只读事务**

**public** **void** findById(Long id) {

Account fromAccount = accountDao.findById(id);

fromAccount.setMoney(10000D);

accountDao.update(fromAccount);

}

}

**提示：@Transactional注解也可以加在方法上，如果类上和方法上都有@Transactional,则以方法上的为准。**

#### 测试注解事务是否成功

运行单元测试类TestTx中的test1方法，控制台报“被零除”异常，account表中的数据也没有发生变化，表示注解事务配置成功啦！

实际开发中，事务我们用xml配置，因为可以一劳永逸

# Spring中的核心监听器

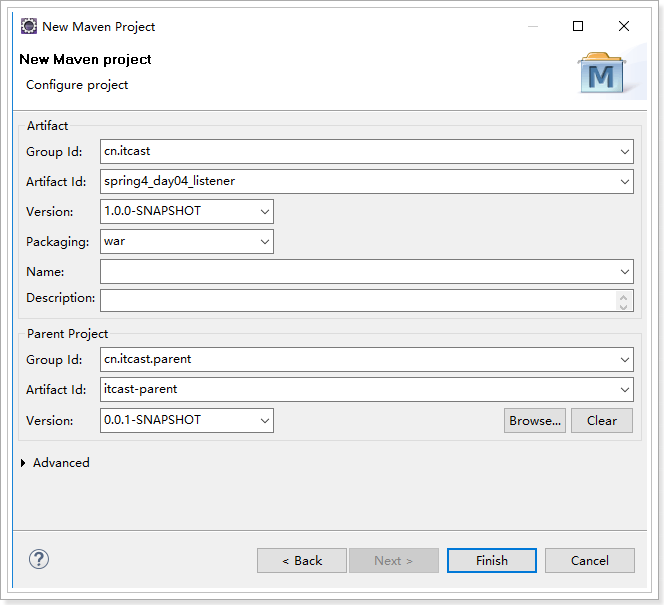
## 介绍

ContextLoaderListener是Spring中的核心监听器，当web工程启动时，该监听器负责创建Spring的IOC容器，存放在ServletContext中；当需要用IOC容器时，就从ServletContext中获取。

好处：

1. 不用我们手动创建IOC容器，由监听器负责创建；
2. 可以保证整个工程中只有一个IOC容器；

## 创建web工程



## 引入依赖

注意：引入spring-web的依赖

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<parent>

<groupId>cn.itcast.parent</groupId>

<artifactId>itcast-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<groupId>cn.itcast</groupId>

<artifactId>spring4\_day04\_listener</artifactId>

<version>1.0.0-SNAPSHOT</version>

<dependencies>

<!-- JSP相关 -->

<dependency>

<groupId>jstl</groupId>

<artifactId>jstl</artifactId>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>servlet-api</artifactId>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jsp-api</artifactId>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

</dependency>

**<dependency>**

**<groupId>org.springframework</groupId>**

**<artifactId>spring-web</artifactId>**

**<version>4.3.13.RELEASE</version>**

**</dependency>**

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

</dependency>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-test</artifactId>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aspects</artifactId>

</dependency>

<!-- jdbc模板 -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

</dependency>

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>druid</artifactId>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

</dependency>

</dependencies>

<build>

<plugins>

<!-- 配置Tomcat插件 -->

<plugin>

<groupId>org.apache.tomcat.maven</groupId>

<artifactId>tomcat7-maven-plugin</artifactId>

<configuration>

<port>8080</port>

<path>/</path>

</configuration>

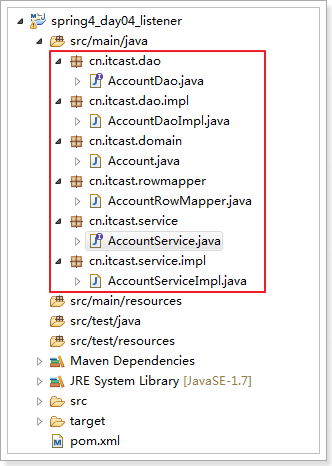
</plugin>

</plugins>

</build>

</project>

## 创建dao、service



* 创建Account实体类，可直接复制之前的

**public** **class** Account **implements** Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** Long id;

**private** String name;

**private** Double money;

**public** Long getId() {

**return** id;

}

**public** **void** setId(Long id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** Double getMoney() {

**return** money;

}

**public** **void** setMoney(Double money) {

**this**.money = money;

}

@Override

**public** String toString() {

**return** "Account [id=" + id + ", name=" + name + ", money=" + money + "]";

}

}

* 创建AccountRowMapper，可直接复制之前的

/\*\*

\* 行映射器

\* **@author** xiaokaibo

\*

\*/

**public** **class** AccountRowMapper **implements** RowMapper<Account> {

/\*\*

\* 如何把表中的一行数据形成java对象

\*/

@Override

**public** Account mapRow(ResultSet rs, **int** row) **throws** SQLException {

Account account = **new** Account();

Long id = rs.getLong("id");

account.setId(id);

String name = rs.getString("name");

account.setName(name);

Double money = rs.getDouble("money");

account.setMoney(money);

**return** account;

}

}

* 创建AccountDao接口

**public** **interface** AccountDao {

**public** List<Account> findAll();

}

* 创建AccountDao接口的实现类

**public** **class** AccountDaoImpl **extends** JdbcDaoSupport **implements** AccountDao {

@Override

**public** List<Account> findAll() {

List<Account> list = **this**.getJdbcTemplate().query("select \* from account",**new** AccountRowMapper());

**return** list;

}

}

* 创建AccountService接口

**public** **interface** AccountService {

/\*\*

\* 查询所有账户

\* **@return**

\*/

**public** List<Account> findAllAccount();

}

* 创建AccountService接口的实现类

**public** **class** AccountServiceImpl **implements** AccountService {

**private** AccountDao accountDao;

**public** **void** setAccountDao(AccountDao accountDao) {

**this**.accountDao = accountDao;

}

@Override

**public** List<Account> findAllAccount() {

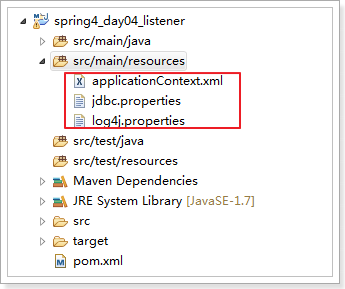
**return** accountDao.findAll();

}

}

## 把相关bean配置到spring中

引入spring配置文件：



<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:aop=*"http://www.springframework.org/schema/aop"*

xmlns:tx=*"http://www.springframework.org/schema/tx"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop.xsd*

*http://www.springframework.org/schema/tx*

*http://www.springframework.org/schema/tx/spring-tx.xsd"*>

<context:property-placeholder location=*"classpath:jdbc.properties"*/>

<bean id=*"accountService"* class=*"cn.itcast.service.impl.AccountServiceImpl"*>

<property name=*"accountDao"* ref=*"accountDao"*></property>

</bean>

<bean id=*"accountDao"* class=*"cn.itcast.dao.impl.AccountDaoImpl"*>

<property name=*"dataSource"* ref=*"dataSource"*></property>

</bean>

<!-- 配置数据源 -->

<bean id=*"dataSource"* class=*"com.alibaba.druid.pool.DruidDataSource"*>

<property name=*"driverClassName"* value=*"${jdbc.driverClass}"*></property>

<property name=*"url"* value=*"${jdbc.url}"*></property>

<property name=*"username"* value=*"${jdbc.username}"*></property>

<property name=*"password"* value=*"${jdbc.password}"*></property>

</bean>

<!-- 配置事务管理器 -->

<bean id=*"transactionManager"* class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"*>

<!-- 注入数据源 -->

<property name=*"dataSource"* ref=*"dataSource"*></property>

</bean>

<!-- 配置事务的属性 -->

<tx:advice transaction-manager=*"transactionManager"* id=*"txAdvice"*>

<tx:attributes>

<!-- find开头方法只读事务 -->

<tx:method name=*"find\*"* read-only=*"true"* />

<!-- 其它方法加可读写事务 -->

<tx:method name=*"\*"*/>

</tx:attributes>

</tx:advice>

<!-- aop配置 -->

<aop:config>

<!-- 告诉spring对哪些方法加事务 -->

<aop:pointcut expression=*"execution(\* cn.itcast.service.impl.\*.\*(..))"* id=*"pt"*/>

<!-- 把上面定义的事务属性应用到切入点 -->

<aop:advisor advice-ref=*"txAdvice"* pointcut-ref=*"pt"*/>

</aop:config>

</beans>

## 在web.xml中配置核心监听器

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<web-app xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns=*"http://java.sun.com/xml/ns/javaee"* xsi:schemaLocation=*"http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"* version=*"2.5"*>

<display-name>spring4\_day04\_listener</display-name>

<welcome-file-list>

<welcome-file>index.html</welcome-file>

<welcome-file>index.htm</welcome-file>

<welcome-file>index.jsp</welcome-file>

<welcome-file>default.html</welcome-file>

<welcome-file>default.htm</welcome-file>

<welcome-file>default.jsp</welcome-file>

</welcome-file-list>

**<!-- 指定spring核心配置文件路径 -->**

**<context-param>**

**<param-name>contextConfigLocation</param-name>**

**<param-value>classpath:applicationContext.xml</param-value>**

**</context-param>**

**<!-- 配置监听器，当工程启动时，创建ioc容器，存放在ServletContext中 -->**

**<listener>**

**<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>**

**</listener>**

</web-app>

## 创建Servlet

**public** **class** AccountServlet **extends** HttpServlet {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**public** AccountServlet() {

**super**();

}

**protected** **void** doGet(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {

//从ServletContext中获取唯一的IOC容器

ApplicationContext ac = WebApplicationContextUtils.*getWebApplicationContext*(**this**.getServletContext());

//从IOC容器中获取AccountService

AccountService accountService = (AccountService) ac.getBean("accountService");

List<Account> accounts = accountService.findAllAccount();

request.setAttribute("accounts", accounts);

request.getRequestDispatcher("/accounts.jsp").forward(request, response);

}

**protected** **void** doPost(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {

doGet(request, response);

}

}

在web.xml中配置Servlet：

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<web-app xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns=*"http://java.sun.com/xml/ns/javaee"* xsi:schemaLocation=*"http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd"* version=*"2.5"*>

<display-name>spring4\_day04\_listener</display-name>

<welcome-file-list>

<welcome-file>index.html</welcome-file>

<welcome-file>index.htm</welcome-file>

<welcome-file>index.jsp</welcome-file>

<welcome-file>default.html</welcome-file>

<welcome-file>default.htm</welcome-file>

<welcome-file>default.jsp</welcome-file>

</welcome-file-list>

<!-- 指定spring核心配置文件路径 -->

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:applicationContext.xml</param-value>

</context-param>

<!-- 配置监听器，当工程启动时，创建ioc容器，存放在ServletContext中 -->

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<servlet>

<servlet-name>findAllAccount</servlet-name>

<servlet-class>cn.itcast.servlet.AccountServlet</servlet-class>

</servlet>

<servlet-mapping>

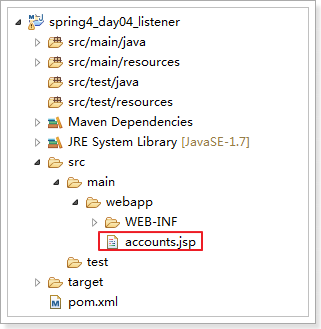
<servlet-name>findAllAccount</servlet-name>

<url-pattern>/findAllAccount</url-pattern>

</servlet-mapping>

</web-app>

## 创建页面



<%@ page language=*"java"* contentType=*"text/html; charset=UTF-8"*

pageEncoding=*"UTF-8"*%>

<%@ taglib uri=*"http://java.sun.com/jsp/jstl/core"* prefix=*"c"*%>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=UTF-8"*>

<title>Insert title here</title>

<style type=*"text/css"*>

*#customers* {

font-family: *"Trebuchet MS", Arial, Helvetica, sans-serif*;

width: *100%*;

border-collapse: *collapse*;

}

*#customers* **td,** *#customers* **th** {

font-size: *1em*;

border: *1px solid #98bf21*;

padding: *3px 7px 2px 7px*;

}

*#customers* **th** {

font-size: *1.1em*;

text-align: *left*;

padding-top: *5px*;

padding-bottom: *4px*;

background-color: *#A7C942*;

color: *#ffffff*;

}

*#customers* **tr***.alt* **td** {

color: *#000000*;

background-color: *#EAF2D3*;

}

</style>

</head>

<body>

<table id=*"customers"*>

<tr>

<th>编号</th>

<th>姓名</th>

<th>余额</th>

</tr>

<c:forEach items=*"*${accounts}*"* var=*"account"*>

<tr>

<td>${account.id}</td>

<td>${account.name}</td>

<td>${account.money}</td>

</tr>

</c:forEach>

</table>

</body>

</html>

## 测试

