Spring 第四天

今天知识点：

1. JdbcTemplate 实现 CURD
2. Spring 声明式事务管理(xml 和注解)

今天的主要内容：

1. JdbcTemplate 实现 CURD (单表)
2. Spring 的事务管理机制（3 个核心接口对象）
3. 声明式事务管理案例-转账（xml-tx、aop、注解@Transactional）

课程目标：

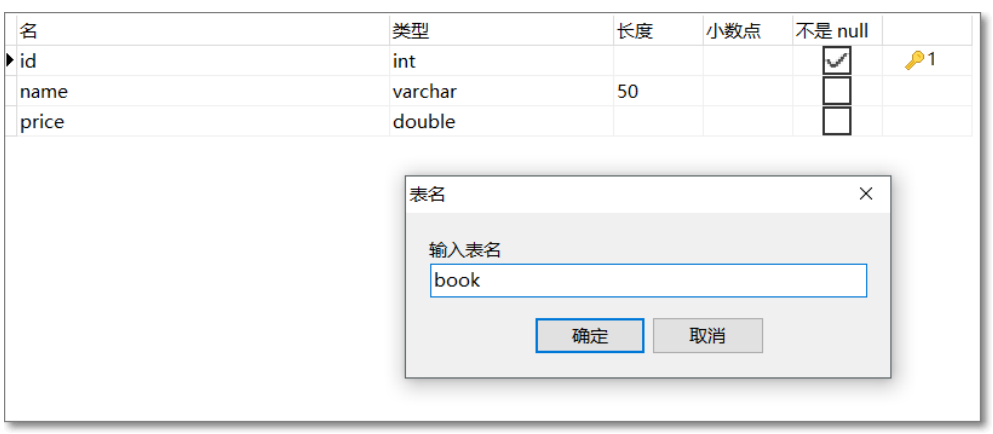
1. 声明式事务的配置编写（事务的传播行为等几个概念，xml，注解写法）
2. Jdbctemplate 的使用

1.1. 基于 JdbcTemplate 实现 DAO（CURD）

方案一：在 dao 中注入 jdbctemplate 成员变量操作数据库

第一步：创建一个表 book：

```
CREATE TABLE `book` (
   `id` int(11) NOT NULL AUTO_INCREMENT,
   `name` varchar(50) DEFAULT NULL,
   `price` double DEFAULT NULL,
   PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```



第二步：创建 cn.itcast.spring.domain 包，创建 Book 类，类中的属性用来对应 book 表的字段

```
//实体类
public class Book {

    private Integer id;
    private String name;
    private Double price;

    public Integer getId() {
        return id;
    }
    public void setId(Integer id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Double getPrice() {
        return price;
```

```
    }
    public void setPrice(Double price) {
        this.price = price;
    }

    public String toString() {
        return "Book [id=" + id + ", name=" + name + ", price=" + price + "]";
    }

}
```

第三步：编写 Dao 类

```
package cn.itcast.spring.dao;

import org.springframework.jdbc.core.JdbcTemplate;

import cn.itcast.spring.domian.Book;

public class BookDao {

    // 注入jdbctemplate
    private JdbcTemplate jdbcTemplate;

    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    // 添加用户
    public void saveBook(Book book) {
        String sql = "insert into book values(null,?,?)";
        this.jdbcTemplate.update(sql, book.getName(), book.getPrice());
    }

}
```

第四步: 在配置文件中配置 dao 并将 jdbctemplate 注入到 dao 对象中:

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

    <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <property name="driverClass" value="${driver}" />
        <property name="jdbcUrl" value="${url}" />
        <property name="user" value="${username}" />
        <property name="password" value="${password}" />
    </bean>



    <bean id="jdbctemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource" />
    </bean>


    <bean id="bookDao" class="cn.itcast.spring.dao.BookDao">
        <property name="jdbcTemplate" ref="jdbctemplate"></property>
    </bean>
```

第五步: 编写测试方法

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "classpath:applicationContext.xml")
public class SpringTest {


    @Autowired
    private BookDao bookDao;

    @Test
    public void test(){
        Book book = new Book();
        book.setName("从入门到精通");
        book.setPrice(99d);
```

```
        bookDao.saveBook(book);
    }
}
```

方案二：继承 Spring 框架封装的 JdbcDaoSupport 类获得 jdbctemplate 对象操作数据库

为了方便 Dao 中注入 JdbcTemplate，Spring 为每一个持久化技术都提供了支持类，如图

ORM 持久化技术	支持类
JDBC	org.springframework.jdbc.core.support.JdbcDaoSupport
Hibernate 5.0	org.springframework.orm.hibernate5.support.HibernateDaoSupport
iBatis	org.springframework.orm.ibatis.support.SqlMapClientDaoSupport

如果想编写 DAO 实现 CURD，只需要继承 Spring 提供 JdbcDAOSupport 支持类 ！

源代码分析 JdbcDaoSupport：不难发现，需要注入数据源

```
public abstract class JdbcDaoSupport extends DaoSupport {

    private JdbcTemplate jdbcTemplate;


    /**
     * Set the JDBC DataSource to be used by this DAO.
     */
    public final void setDataSource(DataSource dataSource) {
        if (this.jdbcTemplate == null || dataSource != this.jdbcTemplate.getDataSource()) {
            this.jdbcTemplate = createJdbcTemplate(dataSource);
            initTemplateConfig();
        }
    }
```

而且只要注入 datasource，就有了 jdbcTemplate，相当于也注入了 jdbcTemplate

```
    /**
     * Create a JdbcTemplate for the given DataSource.
     * Only invoked if populating the DAO with a DataSource reference!
     * <p>Can be overridden in subclasses to provide a JdbcTemplate instance
     * with different configuration, or a custom JdbcTemplate subclass.
     * @param dataSource the JDBC DataSource to create a JdbcTemplate for
     * @return the new JdbcTemplate instance
     * @see #setDataSource
     */
    protected JdbcTemplate createJdbcTemplate(DataSource dataSource) {
        return new JdbcTemplate(dataSource);
    }
```

编写的 Dao 类继承 JdbcDaoSupport

```
//图书操作的 dao 层
//JdbcDaoSupport 简化 JdbcTemplate 的代码开发。
public class BookDao extends JdbcDaoSupport {

    //注入 jdbctempate
//  private JdbcTemplate jdbcTemplate;
//  public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
//      this.jdbcTemplate = jdbcTemplate;
```

```
//  }

    //保存图书
    public void save(Book book){
        String sql="insert into book values(null,?,?)";
        //调用 jdbctemplate
//      jdbcTemplate.update(sql, book.getName(),book.getPrice());
        super.getJdbcTemplate().update(sql, book.getName(),book.getPrice());
    }
}
```

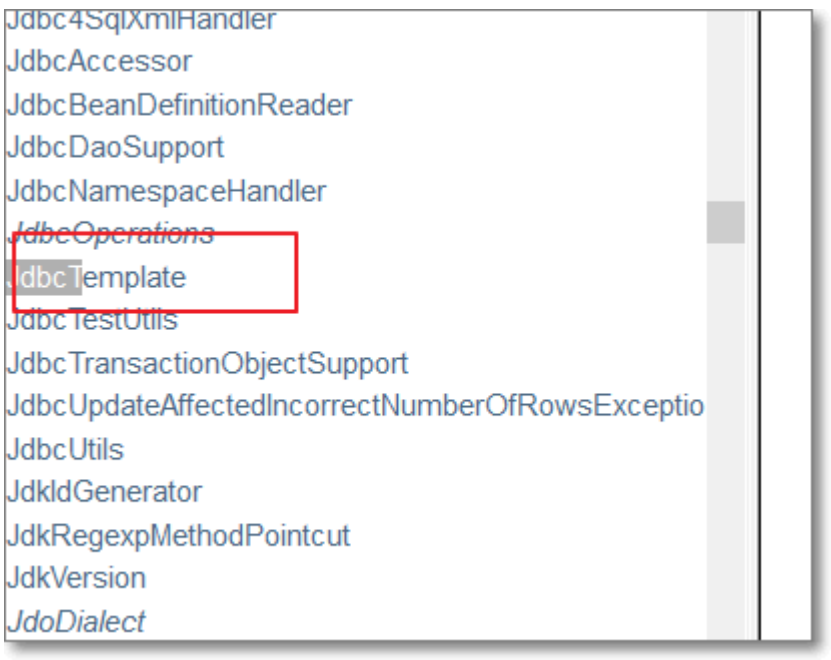
配置 spring 核心配置文件，注入 jdbcTemplate 到 BookDao：

```
<!-- jdbctemplate 对象 -->
    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <!-- 注入数据源 -->
        <property name="dataSource" ref="dataSource"/>
    </bean>
<!-- 配置 dao，注入 jdbctemplate -->
    <bean id="bookDao" class="cn.itcast.spring.dao.BookDao">
        <!-- 方案一：在 BookDao 中提供 jdbcTempate 属性，通过 set 方法注入 jdbcTemplate-->
        <!-- <property name="jdbcTemplate" ref="jdbcTemplate"/> -->
        <!-- 方案二：BookDao 类继承 JdbcDaoSupport，直接注入数据源，就拥有了 jdbctempate 对象 -->
        <property name="dataSource" ref="dataSource"/>
    </bean>
```

1.1.1. 实现增加、删除、修改功能

通过 jdbcTemplate 提供 update 一个方法就可以 增删改

参看 api 文档：spring-framework-4.2.4.RELEASE/docs/javadoc-api/index.html



int	**update(String sql, Object... args)** Issue a single SQL update operation (such as an insert, update or delete statement) via a prepared statement, binding the given arguments.

创建 cn.itcast.spring.dao 包，创建 BookDao 类

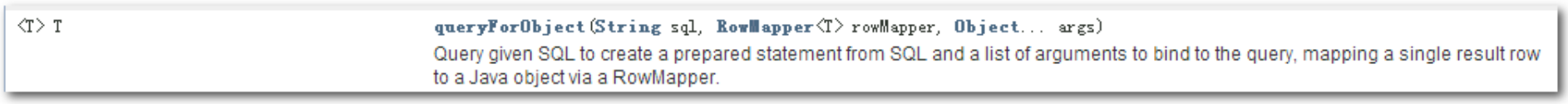
编写 BookDao 类：

```
//图书操作的 dao 层
//JdbcDaoSupport 简化 JdbcTemplate 的代码开发。
public class BookDao extends JdbcDaoSupport {

    //注入 jdbctempate
//  private JdbcTemplate jdbcTemplate;
//  public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
//      this.jdbcTemplate = jdbcTemplate;
//  }

    //保存图书
    public void save(Book book){
        String sql="insert into book values(null,?,?)";
        //调用 jdbctemplate
//      jdbcTemplate.update(sql, book.getName(),book.getPrice());
        super.getJdbcTemplate().update(sql, book.getName(),book.getPrice());
    }

    //更新
    public void update(Book book){
```

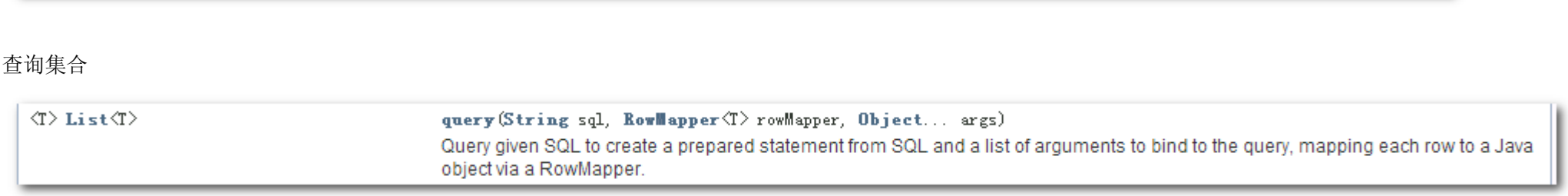
```
        String sql="update book set name =? ,price =? where id =?";
        super.getJdbcTemplate().update(sql, book.getName(),book.getPrice(),book.getId());
    }

    //删除
    public void delete(Book book){
        super.getJdbcTemplate().update("delete from book where id =?", book.getId());
    }
}
```

1.1.2. 查询

查询单个对象



查询集合




```
//根据id查询
    public Book findById(Integer id){
        String sql ="select * from book where id = ?";
        return super.getJdbcTemplate().queryForObject(sql,BeanPropertyRowMapper.newInstance(Book.class), id);
    }

    //查询所有

    public List<Book> findAll( ){
        String sql ="select * from book";
        return super.getJdbcTemplate().query(sql,BeanPropertyRowMapper.newInstance(Book.class));
    }

    //条件查询: 根据图书名称模糊查询信息
    public List<Book> findByNameLike(String name ){
        String sql ="select * from book where name like ?";
        return super.getJdbcTemplate().query(sql,BeanPropertyRowMapper.newInstance(Book.class),"%"+name+"%");
    }
```

创建包 cn.itcast.spring.test

创建 SpringTest.java 进行测试：

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations="classpath:applicationContext.xml")
public class SpringTest {
    //测试 dao
    @Autowired
    private BookDao bookDao;

    /**保存测试*/
    @Test
    public void testSave(){
        Book book = new Book();
        book.setName("如来神掌");
        book.setPrice(1998d);
        bookDao.save(book);
    }

    /**更新测试*/
    @Test
    public void testUpdate(){
        Book book = new Book();
        book.setId(1);
        book.setName("降龙十八掌");
        book.setPrice(298d);
        bookDao.update(book);
    }
```

```
    /**保存更新*/
    @Test
    public void testDelete(){
        Book book = new Book();
        book.setId(2);
        bookDao.delete(book);
    }

    /**使用主键 ID 查询测试*/
    @Test
    public void testFindById(){
        Integer id = 3;
        Book book = bookDao.findById(id);
        System.out.println(book);
    }


    /**查询测试*/
    @Test
    public void testFindAll(){
        List<Book> list = bookDao.findAll();
        System.out.println(list);
    }

    /**查询条件查询测试*/
    @Test
    public void testFindCondition(){
        Book book = new Book();
        book.setName("如来神掌");
        book.setPrice(1998d);
        List<Book> list = bookDao.findByCondition(book);
        System.out.println(list);
    }
}
```

2. Spring 的事务管理机制

Spring 事务管理高层抽象主要包括 3 个接口，Spring 的事务主要是由他们共同完成的：

- PlatformTransactionManager：事务管理器—主要用于平台相关事务的管理
- TransactionDefinition： 事务定义信息(隔离、传播、超时、只读)—通过配置如何进行事务管理。
- TransactionStatus：事务具体运行状态—事务管理过程中，每个时间点事务的状态信息。

2.1.PlatformTransactionManager 事务管理器

参考：spring-framework-4.2.4.RELEASE/docs/javadoc-api/index.html

All Methods | Instance Methods | Abstract Methods

Modifier and Type	Method and Description
void	commit(TransactionStatus status) Commit the given transaction, with regard to its status.
TransactionStatus	getTransaction(TransactionDefinition definition) Return a currently active transaction or create a new one, according to the specified propagation behavior.
void	rollback(TransactionStatus status) Perform a rollback of the given transaction.

该接口提供三个方法：

- commit：提交事务
- rollback：回滚事务
- getTransaction：获取事务状态

Spring 为不同的持久化框架提供了不同 PlatformTransactionManager 接口实现：

事务	说明
org.springframework.jdbc.datasource.DataSourceTransactionManager	使用 Spring JDBC 或 iBatis 进行持久化数据时使用
org.springframework.orm.hibernate5.HibernateTransactionManager	使用 Hibernate5.0 版本进行持久化数据时使用
org.springframework.orm.jpa.JpaTransactionManager	使用 JPA 进行持久化时使用
org.springframework.jdo.JdoTransactionManager	当持久化机制是 Jdo 时使用
org.springframework.transaction.jta.JtaTransactionManager	使用一个 JTA 实现来管理事务，在一个事务跨越多个资源时必须使用

- DataSourceTransactionManager 针对 JdbcTemplate、MyBatis 事务控制 ，使用 Connection（连接）进行事务控制 ：

开启事务 connection.setAutoCommit(false);
提交事务 connection.commit();
回滚事务 connection.rollback();

事务管理器的选择？
用户根据选择和使用的持久层技术，来选择对应的事务管理器。

2.2.TransactionDefinition 事务定义信息

用来定义事务相关的属性的，给事务管理器用。
参考：spring-framework-4.2.4.RELEASE/docs/javadoc-api/index.html

All Methods | Instance Methods | Abstract Methods

Modifier and Type	Method and Description
int	getIsolationLevel() Return the isolation level.
String	getName() Return the name of this transaction.
int	getPropagationBehavior() Return the propagation behavior.
int	getTimeout() Return the transaction timeout.
boolean	isReadOnly() Return whether to optimize as a read-only transaction.

该接口主要提供的方法：

- getIsolationLevel：隔离级别获取
- getPropagationBehavior：传播行为获取
- getTimeout：获取超时时间（事务的有效期）
- isReadOnly 是否只读(保存、更新、删除—对数据进行操作-变成可读写的，查询-设置这个属性为 true，只能读不能写)，事务管理器能够根据这个返回值进行优化。

这些事务的定义信息，都可以在配置文件中配置和定制。

2.2.1. 事务的隔离级别 IsolationLevel

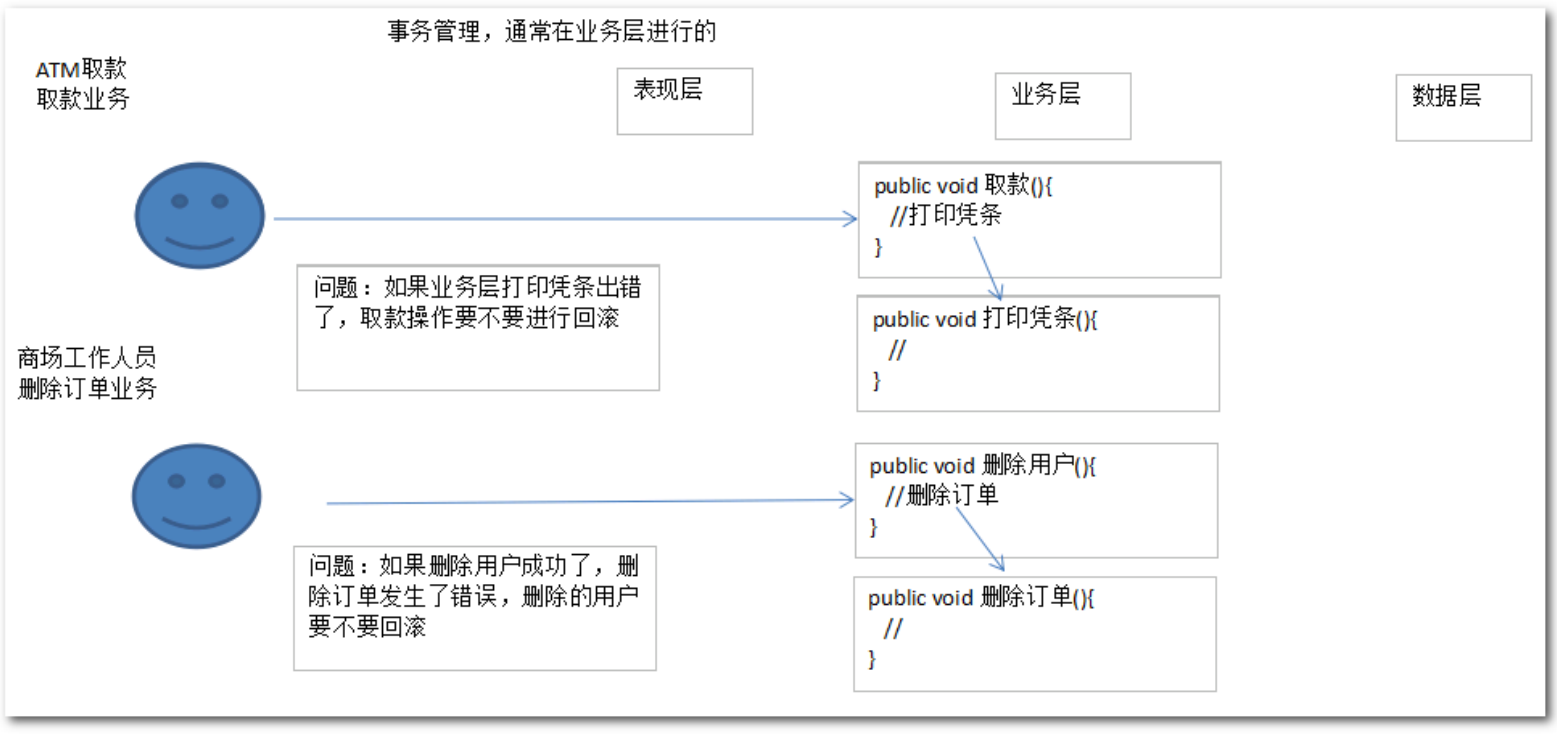
隔离级别	含义
DEFAULT	使用后端数据库默认的隔离级别(spring 中的的选择项)
READ_UNCOMMITED	允许你读取还未提交的改变了的数据。可能导致脏、幻、不可重复读
READ_COMMITTED	允许在并发事务已经提交后读取。可防止脏读，但幻读和 不可重复读仍可发生
REPEATABLE_READ	对相同字段的多次读取是一致的，除非数据被事务本身改变。可防止脏、不可重复读，但幻读仍可能发生。
SERIALIZABLE	完全服从 ACID 的隔离级别，确保不发生脏、幻、不可重复读。这在所有的隔离级别中是最慢的，它是典型的通过完全锁定在事务中涉及的数据表来完成的。

脏读:一个事务读取了另一个事务改写但还未提交的数据,如果这些数据被回滚，则读到的数据是无效的。
不可重复读：在同一事务中，多次读取同一数据返回的结果有所不同。换句话说就是，后续读取可以读到另一事务已提交的更新数据。相反，“可重复读”在同一事务中多次读取数据时，能够保证所读数据一样，也就是，后续读取不能读到另一事务已提交的更新数据。
幻读：一个事务读取了几行记录后，另一个事务插入一些记录，幻读就发生了。再后来的查询中，第一个事务就会发现有些原来没有的记录。

事务四大特性 ACID ---隔离性引发问题 ---- 解决事务的隔离问题 隔离级别
Mysql 默认隔离级别 REPEATABLE_READ
Oracle 默认隔离级别 READ_COMMITTED

2.2.2. 事务的传播行为 PropagationBehavior

什么是事务的传播行为？ 有什么作用？
事务传播行为用于解决两个被事务管理的方法互相调用问题



业务层两个方法面临的事务问题：

* 有些时候需要处于同一个事务（删除用户删除完成之后，需要同时删除用户对应的订单，需要事务回滚，例如商场工作人员删除订单业务），
* 有些时候不能在同一个事务（取款是一个事务操作，打印凭条是一个事务操作，例如 ATM 取款业务） ！

事务的传播行为的 7 种类型：

事务传播行为类型	说明
PROPAGATION_REQUIRED	支持当前事务，如果不存在 就新建一个
PROPAGATION_SUPPORTS	支持当前事务，如果不存在，就不使用事务
PROPAGATION_MANDATORY	支持当前事务，如果不存在，抛出异常
PROPAGATION_REQUIRES_NEW	如果有事务存在，挂起当前事务，创建一个新的事务
PROPAGATION_NOT_SUPPORTED	以非事务方式运行，如果有事务存在，挂起当前事务
PROPAGATION_NEVER	以非事务方式运行，如果有事务存在，抛出异常
PROPAGATION_NESTED	如果当前事务存在，则嵌套事务执行 只对 DataSourceTransactionManager 起效

主要分为三大类：

//开启事务
A
//提交事务

//开启事务
B
//提交事务

- PROPAGATION_REQUIRED(默认值)、PROPAGATION_SUPPORTS、PROPAGATION_MANDATORY
  支持当前事务， A 调用 B，如果 A 事务存在，B 和 A 处于同一个事务 。
  事务默认传播行为 REQUIRED。最常用的。

- PROPAGATION_REQUIRES_NEW、PROPAGATION_NOT_SUPPORTED、PROPAGATION_NEVER
  不会支持原来的事务 ，A 调用 B， 如果 A 事务存在， B 肯定不会和 A 处于同一个事务。
  常用的事务传播行为：PROPAGATION_REQUIRES_NEW

- PROPAGATION_NESTED
  嵌套事务 ，只对 DataSourceTransactionManager 有效 ，底层使用 JDBC 的 SavePoint 机制，允许在同一个事务设置保存点，回滚保存点

附录：嵌套事务的示例：

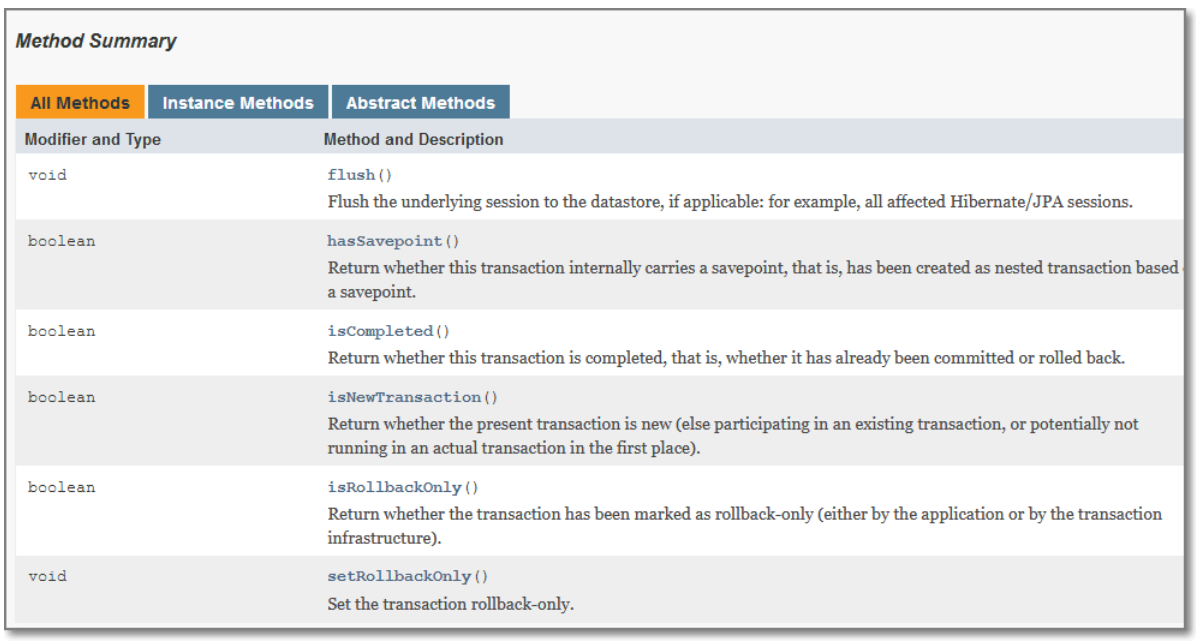
```
Connection conn = null;
try {
    conn.setAutoCommit(false);
    Statement stmt = conn.createStatement();
    stmt.executeUpdate("update person set name='888' where id=1");

    Savepoint savepoint = conn.setSavepoint();
    try{
            conn.createStatement().executeUpdate("update person set name='222' where sid=2");
    }catch(Exception ex){
            conn.rollback(savepoint);
```

```
        }
          stmt.executeUpdate("delete from person where id=9");
          conn.commit();
            stmt.close();
       } catch (Exception e) {
              conn.rollback();
        }finally{
                        try {
            if(null!=conn && !conn.isClosed()) conn.close();
                         } catch (SQLException e) { e.printStackTrace(); }
        }
}
```

【面试题】REQUIRED、RE NESTED QUIRES_NEW、区分
REQUIRED：只有一个事务(默认，推荐)
REQUIRES_NEW：存在两个事务 ，如果事务存在，挂起事务，重新又开启了一个新的事务
NESTED 嵌套事务，事务可以设置保存点，回滚到保存点 ，选择提交或者回滚

2.3.TransactionStatus 事务状态

事务运行过程中，每个时间点 事务状态信息 ！



Method Summary

All Methods | Instance Methods | Abstract Methods

Modifier and Type	Method and Description
void	flush() Flush the underlying session to the datastore, if applicable: for example, all affected Hibernate/JPA sessions.
boolean	hasSavepoint() Return whether this transaction internally carries a savepoint, that is, has been created as nested transaction based a savepoint.
boolean	isCompleted() Return whether this transaction is completed, that is, whether it has already been committed or rolled back.
boolean	isNewTransaction() Return whether the present transaction is new (else participating in an existing transaction, or potentially not running in an actual transaction in the first place).
boolean	isRollbackOnly() Return whether the transaction has been marked as rollback-only (either by the application or by the transaction infrastructure).
void	setRollbackOnly() Set the transaction rollback-only.

flush()，给 hibernate 使用，底层发出 sql 的
hasSavepoint()：判断是否有保留点
isCompleted()：判断事务是否结束
isNewTransaction()：判断当前事务是否是新开的一个事务。
isRollbackOnly()：判断事务是否只能回滚
setRollbackOnly()：设置事务是否回滚

事务的结束：必须通过 commit 确认事务提交， rollback 作用标记为回滚。
数据库操作中，如果只是回滚，后面不操作，数据库在关闭连接的时候，自动发出了 commit。

```
try {
      操作
} catch (){
      rollback
} finally {
      commit
}
```

【三个事务超级接口对象之间的关系】
1）首先用户管理事务，需要先配置 TransactionManager（事务管理器）进行事务管理
2）然后根据 TransactionDefinition(事务定义信息)，通过 TransactionManager（事务管理器）进行事务管理；
3）最后事务运行过程中，每个时刻都可以通过获取 TransactionStatus（事务状态）来了解事务的运行状态。

2.4.Spring 事务管理两种方式

Spring 支持两种方式事务管理
- 一：编程式的事务管理

通过 TransactionTemplate 手动管理事务
在实际应用中很少使用，原因是要修改原来的代码，加入事务管理代码 （侵入性 ）

参考文档：http://www.yiibai.com/spring/programmatic_management.html

```
public void create(String name, Integer age, Integer marks, Integer year){

   TransactionDefinition def = new DefaultTransactionDefinition();
   TransactionStatus status = transactionManager.getTransaction(def);

   try {
      String SQL1 = "insert into Student (name, age) values (?, ?)";
      jdbcTemplateObject.update( SQL1, name, age);

      // Get the latest student id to be used in Marks table
      String SQL2 = "select max(id) from Student";
      int sid = jdbcTemplateObject.queryForInt( SQL2 );

      String SQL3 = "insert into Marks(sid, marks, year) " +
                    "values (?, ?, ?)";
      jdbcTemplateObject.update( SQL3, sid, marks, year);

      System.out.println("Created Name = " + name + ", Age = " + age);
      transactionManager.commit(status);
   } catch (DataAccessException e) {
      System.out.println("Error in creating record, rolling back");
      transactionManager.rollback(status);
      throw e;
   }
   return;
}
```

- 二：使用 XML 或注解配置声明式事务

* Spring 的声明式事务是通过 AOP 实现的（环绕通知）

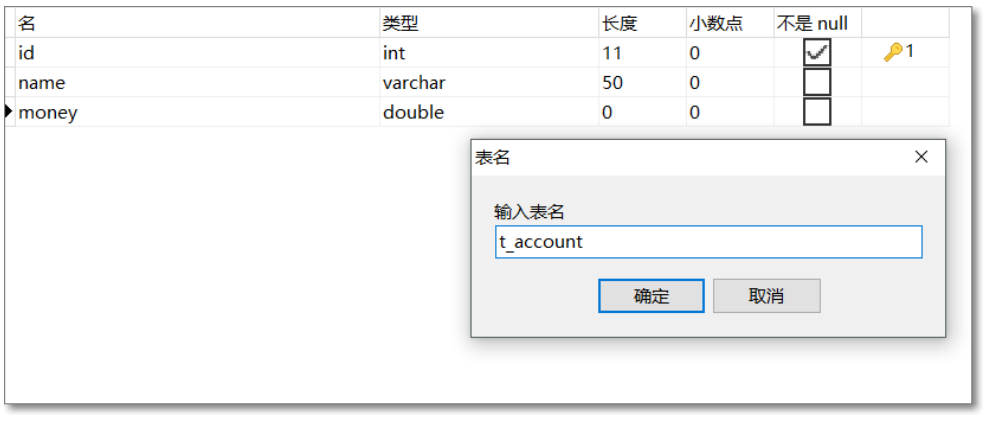
*代开发中经常使用（码侵入性最小）--推荐使用！

3. 声明式事务管理案例-转账（xml、注解）

3.1.编写转账案例，引出事务管理问题

需求：账号转账，Tom 账号取出 1000 元，存放到 Jack 账号上

数据表和测试数据准备：



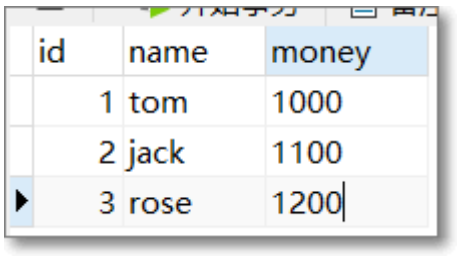
建表脚本（MySQL）：

第一步：创建表 t_account

```
CREATE TABLE `t_account` (
   `id` int(11) NOT NULL AUTO_INCREMENT,
   `name` varchar(50) DEFAULT NULL,
   `money` double DEFAULT NULL,
   PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8 ROW_FORMAT=DYNAMIC;
```

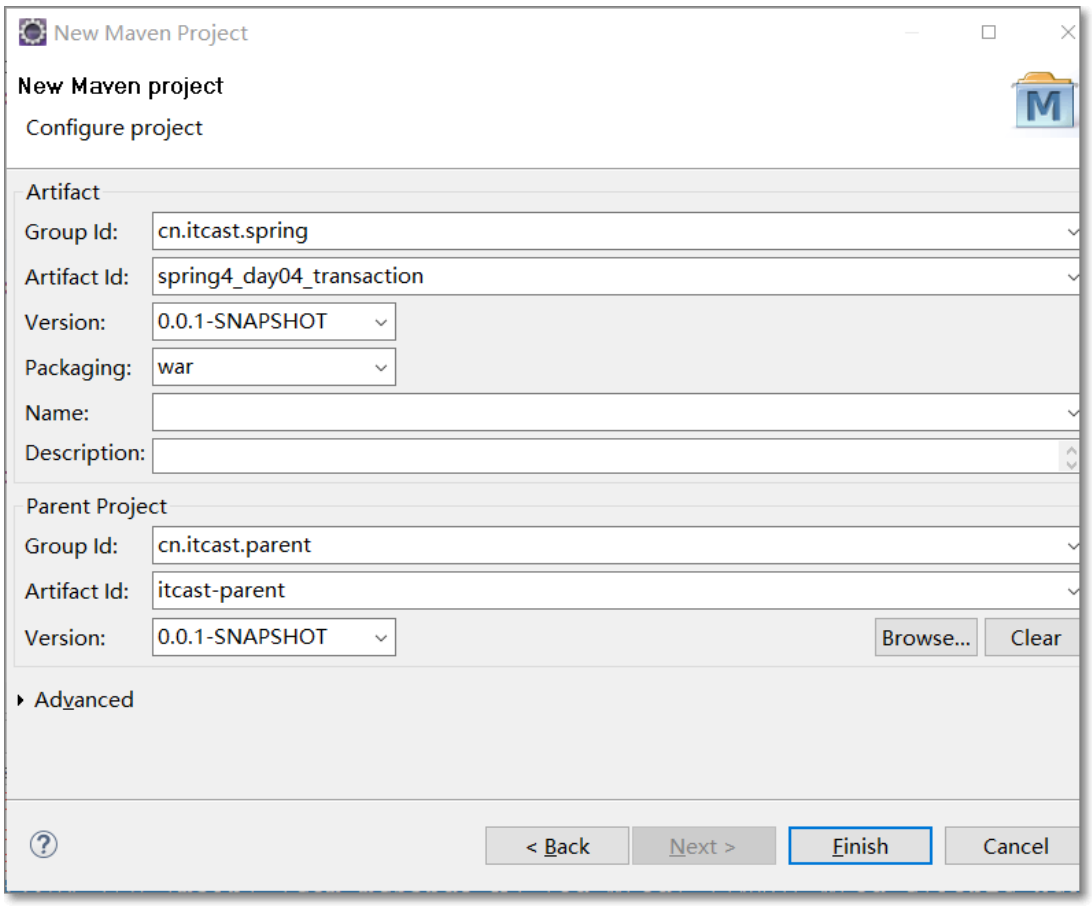
第二步：插入测试数据：

```
INSERT INTO `itcastspring`.`t_account` (`id`, `name`, `money`) VALUES ('1', 'tom', '1000');
INSERT INTO `itcastspring`.`t_account` (`id`, `name`, `money`) VALUES ('2', 'jack', '1100');
INSERT INTO `itcastspring`.`t_account` (`id`, `name`, `money`) VALUES ('3', 'rose', '1200');
```



id	name	money
1	tom	1000
2	jack	1100
3	rose	1200

第一步：新建 web 工程，spring4_day04_transaction



第二步：引入依赖:

```
<!-- spring核心依赖 -->
      <dependency>
         <groupId>org.springframework</groupId>
         <artifactId>spring-context</artifactId>
      </dependency>

      <!-- springaop相关包 -->
      <dependency>
         <groupId>org.springframework</groupId>
         <artifactId>spring-aspects</artifactId>
      </dependency>

      <!-- 单元测试 -->
      <dependency>
         <groupId>junit</groupId>
         <artifactId>junit</artifactId>
         <scope>test</scope>
      </dependency>

      <!-- 日志 -->
      <dependency>
         <groupId>org.slf4j</groupId>
         <artifactId>slf4j-log4j12</artifactId>
      </dependency>

      <!-- spring集成测试 -->
      <dependency>
         <groupId>org.springframework</groupId>
         <artifactId>spring-test</artifactId>
         <version>4.3.13.RELEASE</version>
      </dependency>

      <!-- 操作数据库 -->
      <dependency>
         <groupId>org.springframework</groupId>
         <artifactId>spring-jdbc</artifactId>
      </dependency>

      <!-- MySql -->
      <dependency>
         <groupId>mysql</groupId>
         <artifactId>mysql-connector-java</artifactId>
      </dependency>

      <!-- c3p0数据源 -->
      <dependency>
         <groupId>c3p0</groupId>
         <artifactId>c3p0</artifactId>
```

```
            <version>0.9.1.2</version>
        </dependency>
```

建包: cn.itcast.spring.dao

第三步：创建 IAccountDao 接口

```
public interface IAccountDao {

    //（存入）转入
    public void in(String name,Double money);

    //（取出）转出
    public void out(String name,Double money);
}
```

创建 AccounDaoImpl 实现类，实现了 IAccountDao 接口

```
//账户操作持久层
//技术方案：jdbctempate
public class AccountDaoImpl extends JdbcDaoSupport implements IAccountDao {

    //（存入）转入
    public void in(String name,Double money){
        String sql="update t_account set money = money+ ? where name = ?";
        super.getJdbcTemplate().update(sql, money,name);
    }

    //（取出）转出
    public void out(String name,Double money){
        String sql="update t_account set money = money- ? where name = ?";
        super.getJdbcTemplate().update(sql, money,name);
    }
}
```

第四步：建立 service 层，创建 IAccountService 接口，编写转账的业务代码：

建包: cn.itcast.spring.service

```
public interface IAccountService {

    void transfer(String outName,String inName,Double money);

}
```

创建 AccountServiceImpl 实现类，实现了 IAccountService 接口，编写转账的业务操作

```
//掌握操作的业务层
public class AccountServiceImpl implements IAccountService{

    //注入 dao
    private IAccountDao accountDao;

    public void setAccountDao(IAccountDao accountDao) {
        this.accountDao = accountDao;
    }


    //转账操作的业务逻辑
    public void transfer(String outName,String inName,Double money){
        //调用 dao 层
        //先取出
        accountDao.out(outName, money);
        //再转入

        accountDao.in(inName, money);
    }

}
```

配置 applicationContext.xml:

```
<!-- dao -->
        <bean id="accountDAO" class="cn.itcast.spring.dao.AccountDAOImpl">
            <property name="dataSource" ref="dataSource" />
        </bean>

        <!-- service -->
        <bean id="accountService" class="cn.itcast.spring.dao.AccountServiceImpl">
```

```
            <property name="accountDAO" ref="accountDAO" />
        </bean>
```

第五步：使用 SpringTest 进行测试

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations={"classpath:applicationContext.xml"})
public class SpringTest {
    //注入测试的 service
    @Autowired
    private IAccountService accountService;

    //需求：账号转账，Tom 账号取出 1000 元，存放到 Jack 账号上
    @Test
    public void testTransfer(){
        accountService.transfer("Tom", "Jack", 1000d);
        System.out.println("转账成功！");
    }

}
```

但是发现问题：

事务管理问题：在 Service 层没有事务的情况下，如果出现异常，则会转账不成功，数据异常。

如果不配置事务，那么每一个数据库的操作都是单独的一个事务。

3.2.XML 配置方式添加事务管理(tx、aop 元素)

【操作思路】：

1、 确定目标：需要对 AccountService 的 transfer 方法，配置切入点

2、 需要 Advice （环绕通知），方法前开启事务，方法后提交关闭事务

3、 配置切面和切入点

第一步：

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       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">
```

配置 Advice 通知：

Spring 为简化事务的配置，提供了<tx:advice>来配置事务管理，也可以理解为该标签是 spring 为你实现好了的事务的通知增强方案。

第二步：配置 spring 容器，applicationContext.xml 文件

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       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">

    <!-- 引入外部属性配置文件-->
```

```xml
<context:property-placeholder location="classpath:db.properties"/>

 <!-- 配置数据源 -->
<!-- c3p0连接池 -->
<bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
    <property name="driverClass" value="${jdbc.className}" />
    <property name="jdbcUrl" value="${jdbc.url}" />
    <property name="user" value="${jdbc.user}" />
    <property name="password" value="${jdbc.password}" />
</bean>


<!-- 第一步：定义具体的平台事务管理器（DataSource 事务管理器） -->
<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <!-- 注入数据源 -->
    <property name="dataSource" ref="dataSource"/>
</bean>
<!-- 第二步：定义通知，通知中要处理的就是事务 -->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <!-- 配置事务的属性定义 -->
    <tx:attributes>
        <!-- 配置具体的方法的事务属性
        isolation//事务的隔离级别，默认是按数据库的隔离级别来
        propagation//事务的传播行为，默认是同一个事务
        timeout="-1":事务的超时时间，默认值使用数据库的超时时间。
        read-only="false":事务是否只读，默认可读写。
        rollback-for:遇到哪些异常就回滚，其他的都不回滚
        no-rollback-for：遇到哪些异常不回滚，其他的都回滚。和上面互斥的
         -->
        <tx:method name="transfer" isolation="DEFAULT" propagation="REQUIRED" timeout="-1" read-only="false"/>

        <!-- 支持通配符
            要求 service 中 方法名字必须符合下面的规则
         -->
        <tx:method name="save*"/>
        <tx:method name="update*"/>
        <tx:method name="delete*"/>
        <tx:method name="find*" read-only="true"/>
    </tx:attributes>
</tx:advice>
<!-- 第三步：配置切入点，让通知关联切入点，即事务控制业务层的方法 -->
<aop:config>
    <!-- 切入点 -->
    <aop:pointcut expression="bean(*Service)" id="txPointcut"/>
    <!-- 切面 -->
    <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointcut"/>
</aop:config>

<!-- dao -->
    <bean id="accountDao" class="cn.itcast.spring.dao.AccountDaoImpl">
        <!-- 注入数据源，才拥有 jdbctemplate -->
        <property name="dataSource" ref="dataSource"/>
    </bean>
    <!-- 业务层 -->
    <bean id="accountService" class="cn.itcast.spring.service.AccountServiceImpl">
        <!-- 注入 dao -->
        <property name="accountDao" ref="accountDao"/>
    </bean>

</beans>
```

使用 SpringTest.java 测试：

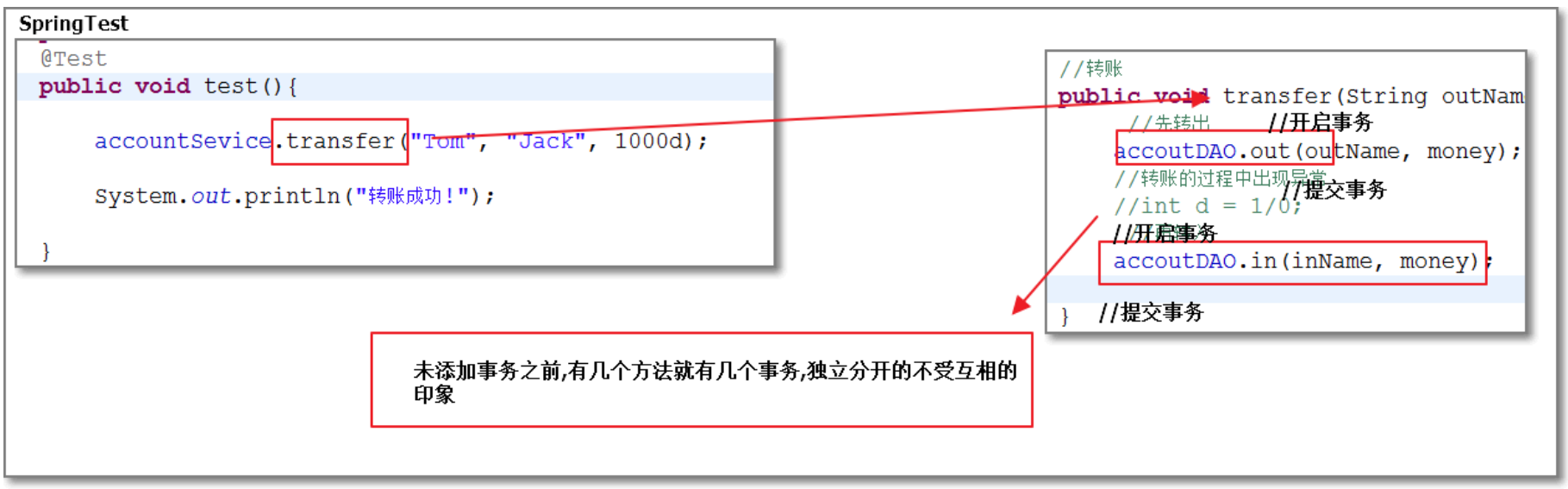
```java
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations={"classpath:applicationContext.xml"})
public class SpringTest {
    //注入测试的 service
    @Autowired
    private IAccountService accountService;

    //需求：账号转账，Tom 账号取出 1000 元，存放到 Jack 账号上
    @Test
```

```
    public void testTransfer(){
        accountService.transfer("Tom", "Jack", 1000d);
        System.out.println("转账成功！");
    }

}
```

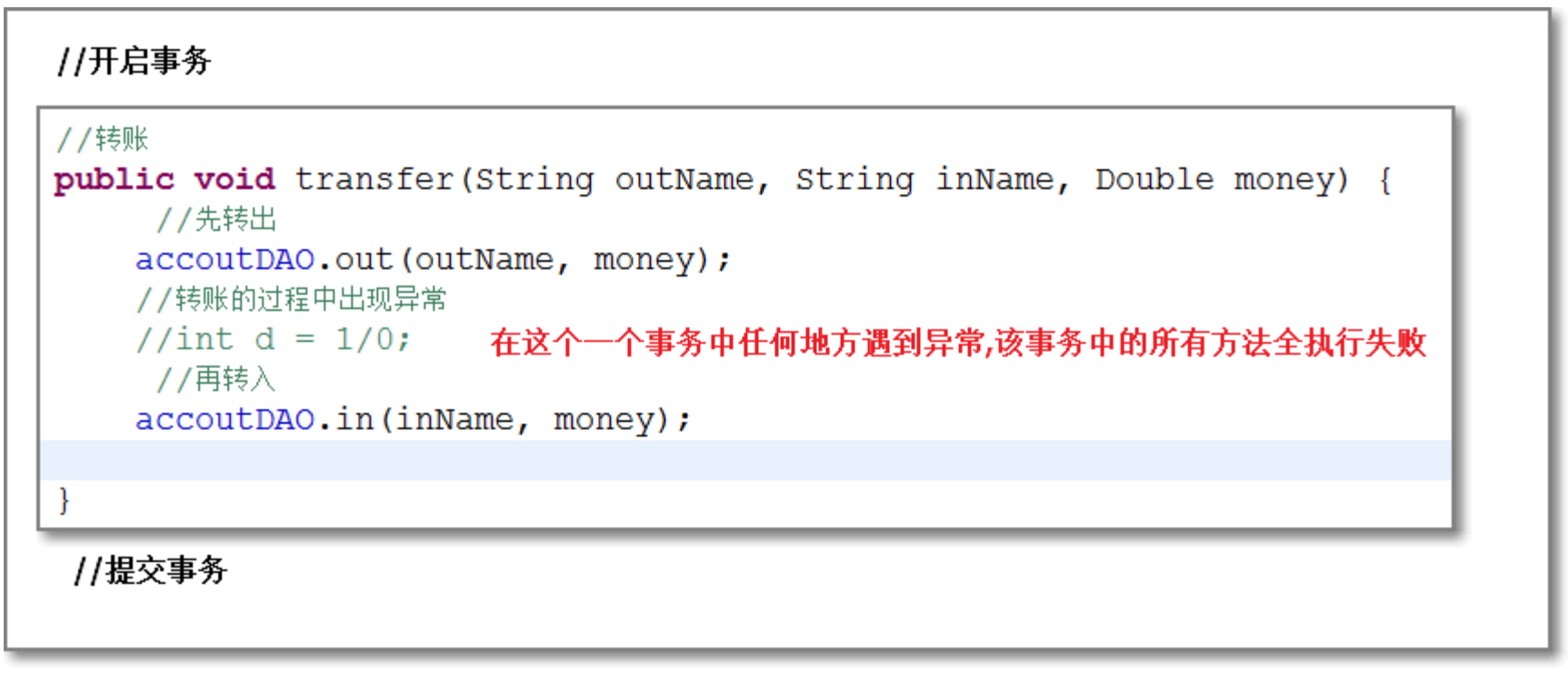
数据正常!

【声明式事务处理的原理图】

没有添加事务：



添加事务：

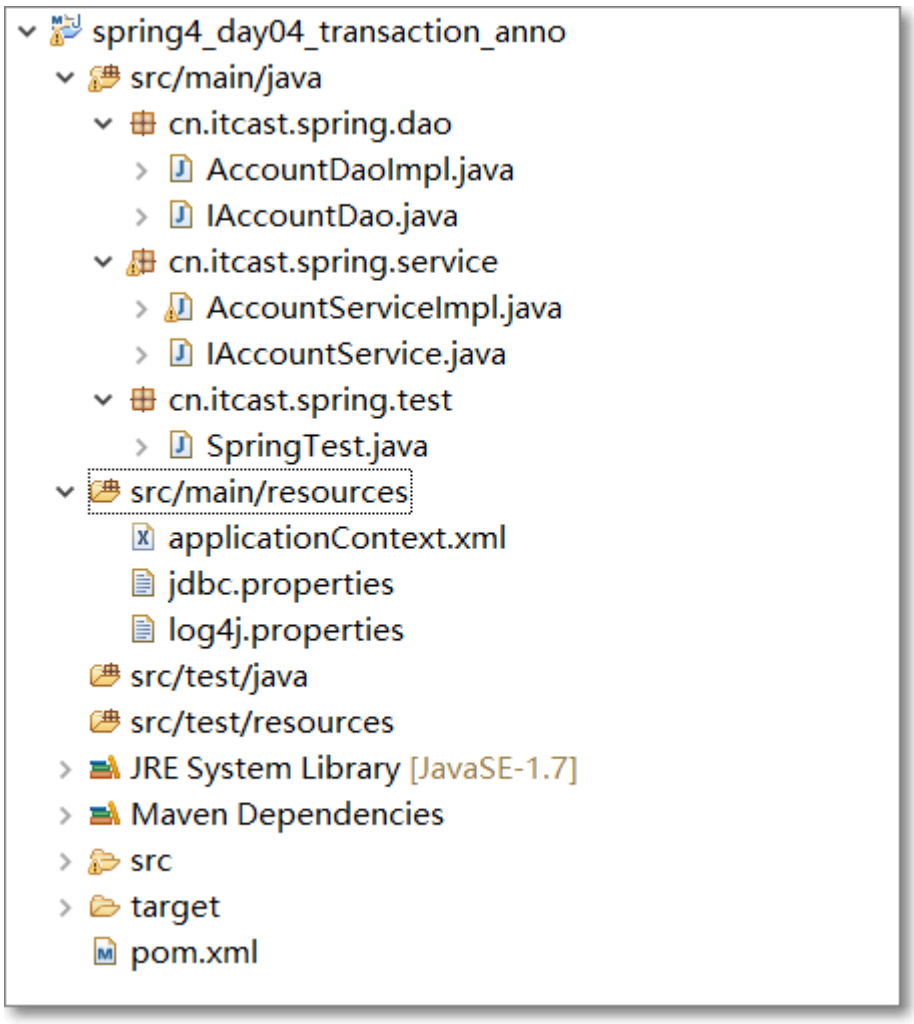


【注意】

如果不配置，则走默认的事务（默认事务是每个数据库操作都是一个事务，相当于没事务）,所以我们开发时需要配置事务。

3.3.注解配置方式添加事务管理 @Transactional

参考之前的项目代码创建 web 项目 spring4_day04_transaction_anno



步骤：

1．在需要管理事务的方法或者类上面 添加@Transactional 注解

2．配置注解驱动事务管理（事务管理注解生效的作用）（需要配置对特定持久层框架使用的事务管理器）

第一步：确定目标（bean 的方法）：

（1）IAccountDao.java 接口

```
public interface IAccountDao {

    //（存入）转入
    public void in(String name,Double money);

    //（取出）转出
    public void out(String name,Double money);
}
```

（2）AccountDaoImpl.java 类

```
//账户操作持久层
//技术方案：jdbctempate
/**
 * @Repository("accountDao")
 * 相当于容易中定义<bean id="accountDao" class="cn.itcast.spring.spring.dao.AccountDaoImpl"/>
 */
@Repository("accountDao")
public class AccountDaoImpl extends JdbcDaoSupport implements IAccountDao {

    //注入数据源
    ////@Autowired
    //private DataSource dataSource;//没有注入数据源成功~
    ////原理：放到属性上的的注解相当于，自动生成 setter 方法上加注解
    //@Autowired   //自动到 spring 的容器中寻找类型是参数类型（DataSource）的 bean
    //public void setDataSource(DataSource dataSource){
    //  this.dataSource=dataSource;
    //}

    @Autowired//当初始化 dao 的时候，会调用该方法啊，通过 set 方法的形参注入数据源
    //方法名无所谓
    public void setSuperDataSource(DataSource dataSource){
        //调用父类的方法
        super.setDataSource(dataSource);
    }

    //（存入）转入
    public void in(String name,Double money){
        String sql="update t_account set money = money+ ? where name = ?";
        super.getJdbcTemplate().update(sql, money,name);
```

```
    }

    //（取出）转出
    public void out(String name,Double money){
        String sql="update t_account set money = money- ? where name = ?";
        super.getJdbcTemplate().update(sql, money,name);
    }
}
```

（1）IAccountService 接口

```
public interface IAccountService {

    void transfer(String outName,String inName,Double money);

}
```

（2）AccountServiceImpl 类

```
//掌握操作的业务层
/**
 * @Service("accountService")
 * 相当于 spring 容器中定义：<bean id="accountService" class="cn.itcast.spring.spring.service.AccountServiceImpl">
 */
@Service("accountService")
@Transactional//会对该类中，所有的共有的方法，自动加上事务--全局的设置，默认是可写
public class AccountServiceImpl implements IAccountService{

    //注入 dao
    @Autowired
    private IAccountDao accountDao;

    //转账操作的业务逻辑
//  @Transactional//在方法上添加事务
    public void transfer(String outName,String inName,Double money){

        //调用 dao 层
        //先取出
        accountDao.out(outName, money);
        int d = 1/0;
        //再转入
        accountDao.in(inName, money);

    }

    @Transactional(readOnly=true)//使用局部覆盖全局的
    public void findAccount(){
        System.out.println("查询帐号的信息了");
    }

}
```

第二步：创建 applicationContext-tx.xml 在 applicationContext-tx.xml 中配置：

```
<?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:p="http://www.springframework.org/schema/p"
      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">

    <!-- 引入外部属性文件 -->
    <context:property-placeholder location="classpath:db.properties" />
    <!-- 配置数据源 -->
    <!-- c3p0连接池 -->
    <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <property name="driverClass" value="${jdbc.className}" />
        <property name="jdbcUrl" value="${jdbc.url}" />
```

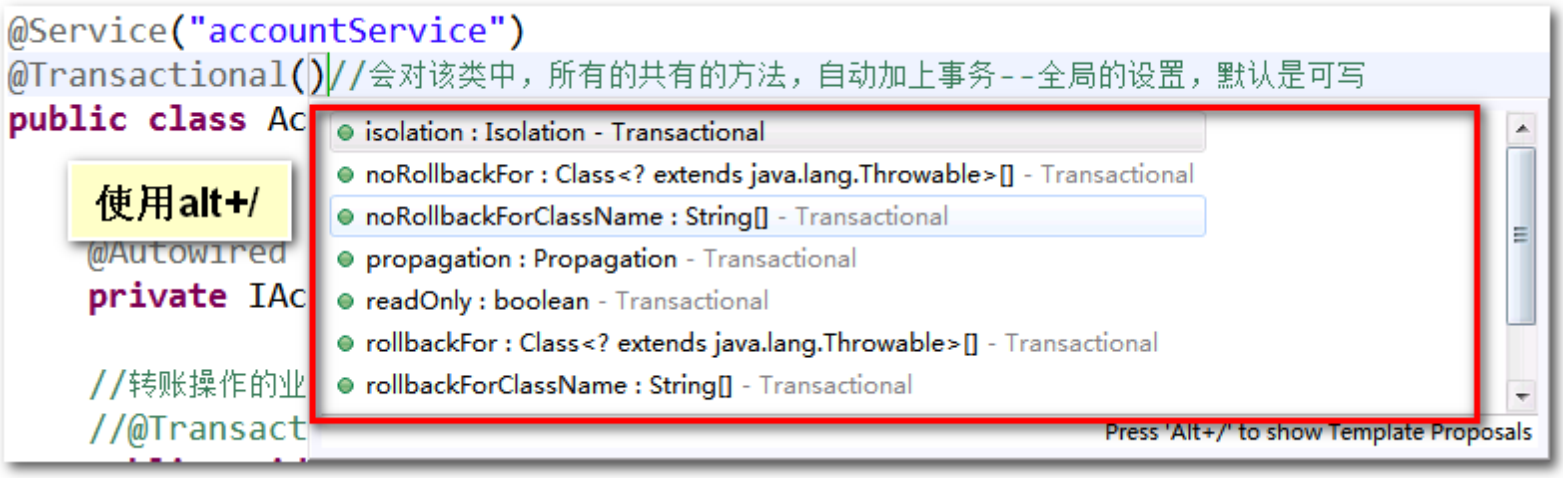
```
        <property name="user" value="${jdbc.user}" />
        <property name="password" value="${jdbc.password}" />
    </bean>
    <!-- 配置 bean 注解扫描 -->
    <context:component-scan base-package="cn.itcast.spring.anntx"/>

    <!-- 定义具体的平台事务管理器（DataSource 事务管理器） -->
    <bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <!-- 注入数据源 -->
        <property name="dataSource" ref="dataSource"/>
    </bean>


    <!-- 配置事务注解驱动 ： 识别事务的注解@tr。。。
    transaction-manager:具体的平台事务管理器
    -->
    <!-- <tx:annotation-driven transaction-manager="transactionManager"/> -->
    <!-- 默认的平台事务管理器的名字叫：transactionManager，此时 transaction-manager="transactionManager"可以不写 -->
    <tx:annotation-driven transaction-manager="transactionManager"/>

</beans>
```

【注意】：数据源的注解注入 需要自己添加 set 方法

（1）在需要管理事务的方法或者类上面 添加@Transactional 注解

```
//转账操作的业务逻辑
@Transactional//在方法上添加事务
public void transfer(String outName,String inName,Double money){

    //调用dao层
    //先取出
    accountDao.out(outName, money);
    int d = 1/0;
    //再转入
    accountDao.in(inName, money);

}
```

（2）配置事务的定义属性信息，在注解中直接配置：



【扩展 1】

如果 @Transactional 标注在 Class 上面， 那么将会对这个 Class 里面所有的 public 方法都包装事务方法。等同于该类的每个公有方法都放上了@Transactional。

如果某方法需要单独的事务定义，则需要在方法上加@Transactional 来覆盖类上的标注声明。记住：方法级别的事务覆盖类级别的事务

```
//掌握操作的业务层
/**
 * @Service("accountService")
 * 相当于 spring 容器中定义：<bean id="accountService" class="cn.itcast.spring.spring.service.AccountServiceImpl">
 */
@Service("accountService")
@Transactional()//会对该类中，所有的共有的方法，自动加上事务--全局的设置，默认是可写
public class AccountServiceImpl implements IAccountService{

    //注入 dao
    @Autowired
    private IAccountDao accountDao;
```

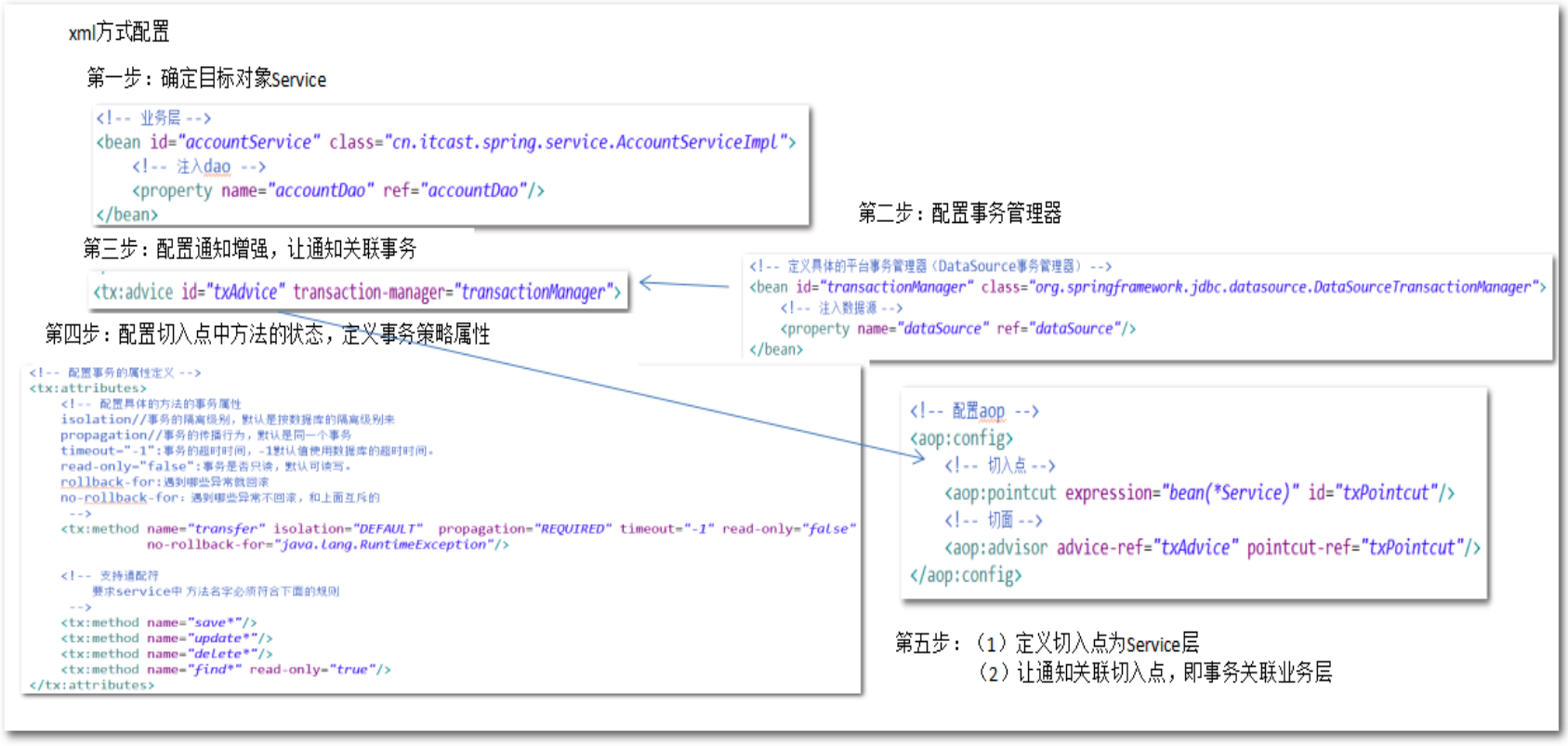
```
    //转账操作的业务逻辑
    @Transactional(readOnly=false)//在方法上添加事务
    public void transfer(String outName,String inName,Double money){

        //调用 dao 层
        //先取出
        accountDao.out(outName, money);
        int d = 1/0;
        //再转入
        accountDao.in(inName, money);

    }

    @Transactional(readOnly=true)//使用局部覆盖全局的
    public void findAccount(){
        System.out.println("查询帐号的信息了");
    }

}
```

3.4.小结-xml 和注解的选择

XML 配置方式和注解配置方式 进行事务管理 哪种用的多？
XML 方式，集中式维护，统一放置到 applicationContext.xml 文件中，缺点在于配置文件中的内容太多。
使用@Transactional 注解进行事务管理，配置太分散，使用 XML 进行事务管理，属性集中配置，便于管理和维护

```
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <!-- 配置事务的属性定义 -->
    <tx:attributes>
        <!-- 配置具体的方法的事务属性
        isolation//事务的隔离级别，默认是按数据库的隔离级别来
        propagation//事务的传播行为，默认是同一个事务
        timeout="-1":事务的超时时间，-1默认值使用数据库的超时时间。
        read-only="false":事务是否只读，默认可读写。
        rollback-for:遇到哪些异常就回滚
        no-rollback-for：遇到哪些异常不回滚，和上面互斥的
         -->
        <tx:method name="transfer" isolation="DEFAULT"  propagation="REQUIRED" timeout="-1" read-only="false"
                   no-rollback-for="java.lang.RuntimeException"/>

        <!-- 支持通配符
            要求service中 方法名字必须符合下面的规则
         -->
        <tx:method name="save*"/>
        <tx:method name="update*"/>
        <tx:method name="delete*"/>
        <tx:method name="find*" read-only="true"/>
    </tx:attributes>
</tx:advice>
```

注意：以后的 service 的方法名字的命名，必须是上面规则，否则，不能被 spring 事务管理。！！！！
即以 save 开头的方法，update 开头的方法，delete 开头的方法，表示增删改的操作，故事务为可写
以 find 开头的方法，表示查询，故事务为只读
（1）xml 方式小结



（2）注解方式小结

作业：

1、 JdbcTemplate 实现 CURD –大家要都用一遍

1、 声明式事务管理原理 （转账）(xml 和注解)