一、Spring中JdbcTemplate的使用

1.1 xml版

1.1.1 pojo对象

```
public class Account {
    private Integer id;
    private String name;
    private Double money;

    // setter getter ……
}
```

1.1.2 dao代码

```
//接口
public interface AccountDao {

    public Account findByName(String name);

    public void update(Account account);

    public List<Account> findAll();

    public void save(Account account);

}
//实现类
public class AccountDaoImpl extends JdbcDaoSupport implements AccountDao {

    @Override
    public Account findByName(String name) {
        return getJdbcTemplate().queryForObject("select * from account where name=?",
                new BeanPropertyRowMapper<Account>(Account.class), name);
    }

    @Override
    public void update(Account account) {
        getJdbcTemplate().update("update account set money=? where id=?",
                account.getMoney(), account.getId());
    }

    @Override
    public List<Account> findAll() {
        return getJdbcTemplate().query("select * from account",

                new BeanPropertyRowMapper<Account>(Account.class));
```

```
    }

    @Override
    public void save(Account account) {
        getJdbcTemplate().update("insert into account (name, money) values (?, ?)",
                account.getName(), account.getMoney());
    }
}
```

1.1.3 service代码

```
//接口
public interface AccountService {

    public Account findByName(String name);

    public void update(Account account);

    public List<Account> findAll();

    public void save(Account account);

}
//实现类
public class AccountServiceImpl implements AccountService {

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

    @Override
    public Account findByName(String name) {
        return accountDao.findByName(name);
    }

    @Override
    public void update(Account account) {
        accountDao.update(account);
    }

    @Override
    public List<Account> findAll() {
        return accountDao.findAll();
    }

    @Override
    public void save(Account account) {
        accountDao.save(account);
    }
}
```

1.1.4 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">


    <!--创建数据库连接池，这里选择使用spring提供的连接池对象-->
    <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql:///spring"/>
        <property name="username" value="root"/>
        <property name="password" value="root"/>
    </bean>

    <!--把dao交给IOC容器-->
    <bean id="accountDao" class="com.itheima.dao.impl.AccountDaoImpl">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <!--把service对象交给IOC容器-->
    <bean id="accountService" class="com.itheima.service.impl.AccountServiceImpl">
        <property name="accountDao" ref="accountDao"/>
    </bean>

</beans>
```

1.1.5 测试

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

    @Autowired
    private AccountService accountService;

    @Test
    public void save(){
        Account account = new Account();
        account.setName("eee");
```

```
        account.setMoney(100d);
        accountService.save(account);
    }

    @Test
    public void findByName(){
        Account account = accountService.findByName("eee");
        System.out.println(account.getName());
    }

    @Test
    public void findAll(){
        List<Account> list = accountService.findAll();
        for (Account account : list) {
            System.out.println(account.getName());
        }
    }

}
```

1.2 常用注解版

1.2.1 复制xml版做修改

略

1.2.2 修改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">

    <!--开启组件扫描-->
    <context:component-scan base-package="com.itheima"/>

    <!--创建数据库连接池，这里选择使用spring提供的连接池对象-->
    <bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql:///spring"/>

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

```
        <property name="password" value="root"/>
    </bean>

</beans>
```

1.2.3 修改dao

```
@Repository
public class AccountDaoImpl extends JdbcDaoSupport implements AccountDao {

    //注：@Autowired放在方法上表示spring会自动执行当前方法
    @Autowired
    public void suibianxie(DataSource dataSource){
        super.setDataSource(dataSource);
    }

    @Override
    public Account findByName(String name) {
        return getJdbcTemplate().queryForObject("select * from account where name=?",
                new BeanPropertyRowMapper<Account>(Account.class), name);
    }

    @Override
    public void update(Account account) {
        getJdbcTemplate().update("update account set money=? where id=?",
                account.getMoney(), account.getId());
    }

    @Override
    public List<Account> findAll() {
        return getJdbcTemplate().query("select * from account",
                new BeanPropertyRowMapper<Account>(Account.class));
    }

    @Override
    public void save(Account account) {
        getJdbcTemplate().update("insert into account (name, money) values (?, ?)",
                account.getName(), account.getMoney());
    }
}
```

1.2.4 修改service

```
@Service
public class AccountServiceImpl implements AccountService {

    @Autowired
    private AccountDao accountDao;

    @Override
    public Account findByName(String name) {
```

```
        return accountDao.findByName(name);
    }

    @Override
    public void update(Account account) {
        accountDao.update(account);
    }

    @Override
    public List<Account> findAll() {
        return accountDao.findAll();
    }

    @Override
    public void save(Account account) {
        accountDao.save(account);
    }
}
```

二、Spring中的事务控制

2.1 Spring中的事务控制方式

Spring的事务控制可以分为编程式和声明式。官方大力推荐使用声明式事务。

- 声明式事务是建立在 AOP 的基础上的。

  其本质是在方法前后进行拦截，在方法开始之前创建事务，在方法结束后根据执行情况提交或回滚事务。
- 声明式事务的优点：可以将事务代码和业务代码完全分离开发，然后通过配置的方式实现运行时组装运行。
- 声明式事务的不足：只能作用到方法级别，无法像编程式事务那样可以作用到代码块级别。

2.2 Spring中事务管理的核心类和方法

- PlatformTransactionManager 平台事务管理器
  - TransactionStatus getTransaction(TransactionDefinition def) 获取事务的状态信息
  - void commit(TransactionStatus status) 提交事务
  - void rollback(TransactionStatus status) 回滚事务
- TransactionStatus 事务运行状态
  - boolean isNewTransaction() 是否是新事物
  - boolean hasSavepoint() 是否有回滚点
  - void setRollbackOnly() 设置为只回滚事务
  - boolean isRollbackOnly() 是否是只回滚事务
  - void flush() 刷新事务状态
  - boolean isCompleted() 事务是否完成
- TransactionDefinition 事务的定义信息（事务隔离级别、事务传播行为等等）
  - 事务隔离级别相关【不设置事务隔离级别，可能引发脏读、不可重复读、幻读】
    - ISOLATION_READ_UNCOMMITTED 读未提交

        - ISOLATION_READ_COMMITTED 读已提交
        - ISOLATION_REPEATABLE_READ 可重复度
        - ISOLATION_SERIALIZABLE 串行化
    - 事务传播行为

      > 事务传播行为指的就是当一个业务方法被另一个业务方法调用时，应该如何进行事务控制。

        - PROPAGATION_REQUIRED(必须有事务，这是默认值)

          如果存在一个事务，则加入到当前事务。如果没有事务则开启一个新的事务。
        - PROPAGATION_REQUIRES_NEW(必须有新的)

          总是开启一个新的事务。如果存在一个事务，则将这个存在的事务挂起,再来一个新的。
        - PROPAGATION_SUPPORTS(支持有事务)

          如果存在一个事务，则加入到当前事务。如果没有事务则非事务运行。
        - PROPAGATION_NOT_SUPPORTED(不支持有事务)

          总是非事务地执行，并挂起任何存在的事务。
        - PROPAGATION_MANDATORY(强制有事务,自己还不负责创建)

          如果存在一个事务，则加入到当前事务。如果没有事务，则抛出异常。
        - PROPAGATION_NEVER(强制不要事务,自己还不负责挂起):

          总是非事务地执行，如果存在一个活动事务，则抛出异常。
        - PROPAGATION_NESTED(嵌套事务)

          如果一个活动的事务存在，则运行在一个嵌套的事务中。如果没有活动事务, 则开启一个新的事务。

          内层事务依赖于外层事务。外层事务失败时，会回滚内层事务所做的动作。

          而内层事务操作失败并不会引起外层事务的回滚。
    - 只读事务

      isReadOnly：配置事务是否是只读事务
    - 事务超时

      TIMEOUT_DEFAULT 事务的超时时间，需要底层数据库支持才能使用此配置，-1代表无限制。

总结：Spring中的事务控制主要就是通过这三个API实现的

- PlatformTransactionManager 负责事务的管理，他是个接口，其子类负责具体工作
- TransactionDefinition 定义了事务的一些相关参数
- TransactionStatus 代表事务运行的一个实时状态

可以简单的理解三者的关系：**事务管理器**通过读取**事务定义参数**进行事务管理，然后会产生一系列的**事务状态**。

2.3 基于XML的声明式事务配置

首先明确声明式事务控制是采用AOP思想来实现的。

使用AOP就要有三部分内容：

- 核心业务代码(目标对象)
- 事务增强代码(Spring提供好了事务管理器)
- 切面配置

```
<?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: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">

    <!--开启组件扫描-->
    <context:component-scan base-package="com.itheima"/>

    <!--创建数据库连接池-->
    <bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql:///spring"/>
        <property name="username" value="root"/>
        <property name="password" value="root"/>
    </bean>

    <!--事务管理器配置-->
    <bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <!--配置事务的通知-->
    <!--
        id="advice" 表示IOC容器中真正的通知对象的id。
        transaction-manager="transactionManager" 表示指定当前要对哪个事务管理器进行配置
        如果事务管理器在IOC容器中的id为transactionManager，此配置可以省略。
    -->
    <tx:advice id="advice" transaction-manager="transactionManager">
        <tx:attributes>
            <!--
                <tx:method 指定目标对象中切入点的方法名
                name="*" 表示目标对象中所有方法都是切入点，都要有事务。
                name="save*" 目标对象中所有以save开头的方法要走事务。
                name="update*" 目标对象中所有以update开头的方法要走事务。
                name="delete*" 目标对象中所有以delete开头的方法要走事务。
                name="find*" 所有以find开头的方法不走事务。
```

```
                propagation="REQUIRED" 指定事务的传播行为，默认REQUIRED表示有且只有一个事务，
SUPPORTS表示有无事务均可
                read-only="true"  是否只读，true表示只能做查询操作，没有事务
                rollback-for=""  指定一个异常，出现此异常才回滚，其余异常都不回滚。不配置默认表示
所有异常都回滚
                no-rollback-for=""  指定一个异常，出现此异常不回滚，其余异常都回滚。不配置默认表
示所有异常都回滚
                timeout="-1"  指定超时时间，默认-1，表示永不超时
                isolation=""  指定事务的隔离级别，默认使用当前数据库默认的事务隔离级别
            -->
            <tx:method name="save*" propagation="REQUIRED"/>
            <tx:method name="update*" propagation="REQUIRED"/>
            <tx:method name="delete*" propagation="REQUIRED"/>
            <tx:method name="find*" read-only="true"/>
            <tx:method name="*"/>
        </tx:attributes>
    </tx:advice>

    <!--进行AOP配置-->
    <aop:config>
        <!--切入点表达式-->
        <aop:pointcut id="pointcut" expression="execution(* com.itheima.service.impl.*.*
(..))"/>
        <!--配置切面-->
        <!--<aop:advisor只有在spring的声明式事务配置时才能使用-->
        <aop:advisor advice-ref="advice" pointcut-ref="pointcut"/>
    </aop:config>

</beans>
```

2.4 基于注解的声明式事务配置

1）去掉配置文件中< tx:advice> 和 < aop:config> 相关配置，加入一个< tx:annotation-driven>

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

2）然后在需要加入事务的类或者方法上添加注解@Transactional

```
//此注解中的属性可以根据需求选择配置，方法上优先级高于类上。
@Transactional(transactionManager = "transactionManager",//事务管理器
               readOnly = false,//只读事务
               isolation = Isolation.READ_COMMITTED,//事务隔离级别
               propagation = Propagation.REQUIRED//事务传播行为
)
```

3) 也可以用配置类完成声明式事务零配置文件实现

```
@Configuration
@EnableTransactionManagement   //声明式事务注解驱动<tx:annotation-driven/>
@ComponentScan("com.itheima")  //组件扫描
public class Config {
}
```

三、作业

使用spring的声明式事务来实现转账案例，持久层要求使用JdbcTemplate。