一、转账案例

1.01 【原始版】创建工程并导入相关jar包

```
<dependencies>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.46</version>
    </dependency>
    <dependency>
        <groupId>c3p0</groupId>
        <artifactId>c3p0</artifactId>
        <version>0.9.1.2</version>
    </dependency>
    <dependency>
        <groupId>commons-dbutils</groupId>
        <artifactId>commons-dbutils</artifactId>
        <version>1.6</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.1.6.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
    </dependency>
</dependencies>
```

1.02 【原始版】pojo对象

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

    //setter getter ……

}
```

1.03 【原始版】dao层代码

```
//接口
public interface AccountDao {

    public Account findByName(String name);

    public void update(Account account);

}

//实现类
@Repository
public class AccountDaoImpl implements AccountDao {

    @Autowired
    private QueryRunner queryRunner;

    @Override
    public Account findByName(String name) {
        Account account = null;
        try {
            account = queryRunner.query("select * from account where name=?", new
BeanHandler<Account>(Account.class), name);
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return account;
    }
    @Override
    public void update(Account account) {
        try {
            queryRunner.update("update account set money=? where id=?", account.getMoney(),
account.getId());
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

}
```

1.04 【原始版】service层代码

```
//接口
public interface AccountService {
    public void transfer(String fromUser, String toUser, Double money);
}

//实现类
@Service
public class AccountServiceImpl implements AccountService {
```

```
    @Autowired
    private AccountDao accountDao;

    @Override
    public void transfer(String fromUser, String toUser, Double money) {
        //查询出转账双方的账户
        Account fromAccount = accountDao.findByName(fromUser);
        Account toAccount = accountDao.findByName(toUser);
        //修改双方的金额
        fromAccount.setMoney(fromAccount.getMoney()-money);
        toAccount.setMoney(toAccount.getMoney()+money);
        //更新双方的账户
        accountDao.update(fromAccount);
        //模拟异常
        //int i=1/0;
        accountDao.update(toAccount);

    }

}
```

1.05 【原始版】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"
        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">

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

    <!--把数据库连接池交给IOC容器-->
    <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">
        <property name="driverClass" value="com.mysql.jdbc.Driver"/>
        <property name="jdbcUrl" value="jdbc:mysql:///spring"/>
        <property name="user" value="root"/>
        <property name="password" value="root"/>
    </bean>

    <!--把QueryRunner对象交给IOC容器-->
    <bean id="queryRunner" class="org.apache.commons.dbutils.QueryRunner">
        <constructor-arg name="ds" ref="dataSource"/>
    </bean>
```

```
</beans>
```

1.06 【原始版】测试

```
public class AccountTest {
    @Test
    public void transfer(){
        ApplicationContext ac = new ClassPathXmlApplicationContext("applicationContext.xml");
        AccountService accountService = ac.getBean(AccountService.class);
        accountService.transfer("aaa", "bbb", 100d);
    }

}
```

1.07 【原始版】问题分析

此时事务在dao层，每次数据库操作都是一个独立的事务，但实际开发，应该把一次业务逻辑控制在一个事务中，所以应该将事务挪到service层。

1.08 【传统事务版】编写生产Connection对象的工具类

```
package com.itheima.utils;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.SQLException;

@Component
public class ConnectionUtils {
    //使Connection对象与当前线程绑定
    private ThreadLocal<Connection> tl = new ThreadLocal<>();

    @Autowired
    private DataSource dataSource;

    public Connection getConnection(){
        //从线程的局部变量中获取Connection对象
        Connection conn = tl.get();
        //如果没有获取到
        if(conn==null){
            try {
                //从数据库连接池中获取一个Connection对象
                conn = dataSource.getConnection();
                //让Datasource对象与当前线程绑定
                tl.set(conn);

            } catch (SQLException e) {
```

```
                e.printStackTrace();
            }
        }
        return conn;
    }

    //让Connection对象与当前线程解绑
    public void remove(){
        tl.remove();
    }

}
```

1.09【传统事务版】修改dao代码

```
@Repository
public class AccountDaoImpl implements AccountDao {

    @Autowired
    private QueryRunner queryRunner;

    @Autowired
    private ConnectionUtils connectionUtils;

    @Override
    public Account findByName(String name) {
        Account account = null;
        try {
            account = queryRunner.query(connectionUtils.getConnection(),
                    "select * from account where name=?", new BeanHandler<Account>
(Account.class), name);
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return account;
    }
    @Override
    public void update(Account account) {
        try {
            queryRunner.update(connectionUtils.getConnection(),
                    "update account set money=? where id=?", account.getMoney(),
account.getId());
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

}
```

1.10【传统事务版】编写事务管理器

```java
package com.itheima.utils;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import java.sql.SQLException;

@Component
public class TxUtils {
    @Autowired
    private ConnectionUtils connectionUtils;
    //开启事务
    public void openTx(){
        try {
            connectionUtils.getConnection().setAutoCommit(false);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    //提交事务
    public void commitTx(){
        try {
            connectionUtils.getConnection().commit();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    //回滚事务
    public void rollbackTx(){
        try {
            connectionUtils.getConnection().rollback();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    //关闭事务
    public void closeTx(){
        try {
            connectionUtils.getConnection().close();
            connectionUtils.remove();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

}
```

1.11【传统事务版】修改service代码

```
package com.itheima.service.impl;

import com.itheima.dao.AccountDao;
import com.itheima.domain.Account;
import com.itheima.service.AccountService;
import com.itheima.utils.TxUtils;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class AccountServiceImpl implements AccountService {

    @Autowired
    private AccountDao accountDao;

    @Autowired
    private TxUtils txUtils;

    @Override
    public void transfer(String fromUser, String toUser, Double money) {
        try {
            //开启事务
            txUtils.openTx();
            //查询出转账双方的账户
            Account fromAccount = accountDao.findByName(fromUser);
            Account toAccount = accountDao.findByName(toUser);
            //修改双方的金额
            fromAccount.setMoney(fromAccount.getMoney()-money);
            toAccount.setMoney(toAccount.getMoney()+money);
            //更新双方的账户
            accountDao.update(fromAccount);
            int i=1/0;
            accountDao.update(toAccount);
            //提交事务
            txUtils.commitTx();
        }catch (Exception e){
            e.printStackTrace();
            //回滚事务
            txUtils.rollbackTx();
        }finally {
            //关闭事务
            txUtils.closeTx();
        }
    }

}
```

1.12【传统事务版】问题分析

此时事务虽然控制住了，但是业务层代码与事务管理器耦合太紧密了，违背了面向对象开发的思想。

1.13【AOP事务版】恢复service为原始版的模样

```
package com.itheima.service.impl;

import com.itheima.dao.AccountDao;
import com.itheima.domain.Account;
import com.itheima.service.AccountService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service
public class AccountServiceImpl implements AccountService {

    @Autowired
    private AccountDao accountDao;
    @Override
    public void transfer(String fromUser, String toUser, Double money) {
        //查询出转账双方的账户
        Account fromAccount = accountDao.findByName(fromUser);
        Account toAccount = accountDao.findByName(toUser);
        //修改双方的金额
        fromAccount.setMoney(fromAccount.getMoney()-money);
        toAccount.setMoney(toAccount.getMoney()+money);
        //更新双方的账户
        accountDao.update(fromAccount);
        int i=1/0;
        accountDao.update(toAccount);

    }
}
```

1.14【AOP事务版】jdk动态代理实现事务增强

```
package com.itheima.factory;

import com.itheima.service.AccountService;
import com.itheima.utils.TxUtils;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.context.annotation.Bean;
import org.springframework.stereotype.Component;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

@Component

public class JdkProxyFactory {
```

```
    @Autowired
    @Qualifier("accountServiceImpl")
    private AccountService accountService;

    @Autowired
    private TxUtils txUtils;

    @Bean
    public AccountService createJdkProxyAccountService(){
        return (AccountService)
Proxy.newProxyInstance(accountService.getClass().getClassLoader(),
            accountService.getClass().getInterfaces(),
            new InvocationHandler() {
                @Override
                public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
                    Object obj = null;
                    try {
                        //开启事务
                        txUtils.openTx();
                        obj = method.invoke(accountService, args);
                        //提交事务
                        txUtils.commitTx();
                    }catch (Exception e){
                        e.printStackTrace();
                        txUtils.rollbackTx();
                    }finally {
                        txUtils.closeTx();
                    }
                    return obj;
                }
            });
    }
}
```

1.15【AOP事务版】cglib动态代理实现事务增强

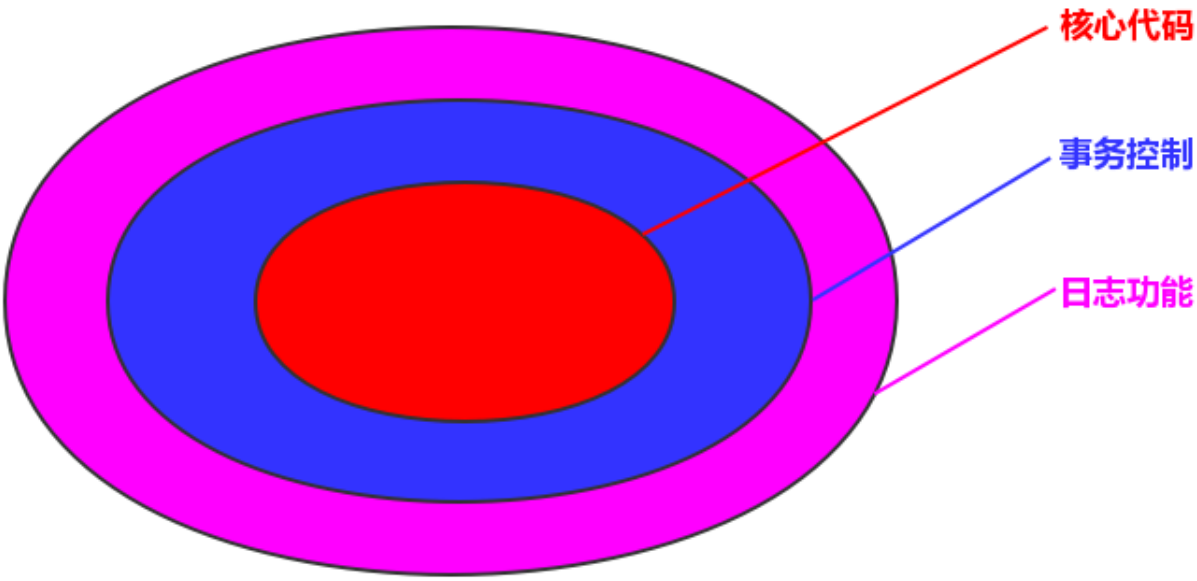
```
package com.itheima.factory;

import com.itheima.service.AccountService;
import com.itheima.utils.TxUtils;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.cglib.proxy.Enhancer;
import org.springframework.cglib.proxy.MethodInterceptor;
import org.springframework.cglib.proxy.MethodProxy;
import org.springframework.context.annotation.Bean;
import org.springframework.stereotype.Component;

import java.lang.reflect.Method;
```

```
@Component
public class CglibProxyFactory {

    @Autowired
    @Qualifier("accountServiceImpl")
    private AccountService accountService;

    @Autowired
    private TxUtils txUtils;

    @Bean
    public AccountService createCglibProxyAccountService(){
        //参数一：目标对象的类型
        //参数二：动作类
        return (AccountService) Enhancer.create(accountService.getClass(), new
MethodInterceptor() {
            /**
             * 前三个参数和jdk动态代理里面的是一样的
             * @param proxy
             * @param method
             * @param args
             * @param methodProxy 增强后的方法
             */
            @Override
            public Object intercept(Object proxy, Method method, Object[] args, MethodProxy
methodProxy) throws Throwable {
                Object obj = null;
                try {
                    //开启事务
                    txUtils.openTx();
                    obj = method.invoke(accountService, args);
                    //提交事务
                    txUtils.commitTx();
                }catch (Exception e){
                    e.printStackTrace();
                    txUtils.rollbackTx();
                }finally {
                    txUtils.closeTx();
                }
                return obj;
            }
        });
    }
}
```

二、初识AOP

AOP（Aspect Oriented Programming）面向切面编程。

- 通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术。
- AOP是一种编程范式，是OOP的延续，在OOP基础之上进行横向开发。
- AOP研究的不是每层内部如何开发，而是同一层面上各个模块之间的共性功能。
  比如：事务、日志、统计。。。



AOP中的专业术语:

连接点(Join point)

> 程序运行中的一些时间点, 例如一个方法的执行, 或者是一个异常的处理。
>
> 在 Spring AOP 中, join point 总是方法的执行点, 即只有方法连接点.

切点(Point cut)

> AOP可以提供了一组规则，按照这些规则去切(匹配)连接点，匹配出来的就叫切点。

增强(Advice)

> 就是指的一个有具体意义的功能，比如说事务控制代码。

织入(Weaving)

> 这是一个动作。将增强代码加入到核心代码的过程就叫织入。

目标对象(Target)

> 需要被增强的对象，即切入点方法所在对象。

代理对象(Proxy)

> 目标对象被增强后成为代理对象

切面(Aspect)

> 这是一个设计概念，它包含了Advice 和 Pointcut。

> Advice定义了Aspect的任务和什么时候执行，Pointcut定义在哪里切入。
>
> 也就是说，Aspect定义了一个什么样的增强功能，切入到哪个核心方法的哪个位置。

三、Spring中AOP功能

> AOP是Spring的一个核心功能，在Spring底层是通过动态代理的方式实现的AOP。
>
> Spring对JDK和CGLIB代理都做了实现，它会根据被代理类是否实现了接口决定采用哪种动态代理方式。
>
> 如果被代理类实现了接口，就采用JDK动态代理；如果没接口，就采用CGLIB动态代理。

spring的AOP有五种通知类型

Spring的AOP支持的通知类型共有5种:

- before:前置通知　　方法执行前执行， 如果其中抛出异常，则不在运行
- afterReturning:后置通知　　方法正常返回后执行，如果方法中抛出异常，则不在运行
- afterThrowing:异常通知　　方法抛出异常后执行，如果方法没有抛出异常，则不运行
- after:最终通知　　方法执行完毕后执行，无论方法中是否出现异常都会执行
- around:环绕通知　　这是一种特殊的通知，以编程方式来实现上面四种通知

3.1 环境准备

3.1.1 引入坐标

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.1.5.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.aspectj</groupId>
        <artifactId>aspectjweaver</artifactId>
        <version>1.8.13</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
    </dependency>
</dependencies>
```

3.1.2 创建目标对象

```
//接口

public interface UserService {
```

```
    public void save();

    public Integer update();

    public Integer delete(Integer id);

}
//实现类
@Service
public class UserServiceImpl implements UserService {
    @Override
    public void save() {
        System.out.println("保存成功了！ ");
    }

    @Override
    public Integer update() {
        System.out.println("更新成功了！ ");
        return 1;
    }

    @Override
    public Integer delete(Integer id) {
        System.out.println("删除成功了！ "+id);
        return 1;
    }
}
```

3.2 spring四大通知基于xml使用

3.2.1 增强类

```
@Component
public class MyAdvice {

    //前置通知，就是在切入点方法前执行
    public void before(){
        System.out.println("前置通知！ ");
    }
    //后置通知，就是在切入点方法后执行
    public void afterReturning(){
        System.out.println("后置通知！ ");
    }
    //异常通知，就是在切入点方法出错时执行
    public void afterThrowing(){
        System.out.println("异常通知！ ");
    }
    //最终通知，就是在切入点方法执行完毕，不管成功与否

    public void after(){
```

```
        System.out.println("最终通知! ");
    }
}
```

3.2.2 spring配置文件中aop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"
        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">

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

    <!--aop四大通知基于xml配置-->
    <aop:config>
        <!--配置切入点表达式-->
        <!--
            execution(* *..*.*(..))是切入点的全统配方式，表示整个项目中所有类中所有方法都要被增强。
            最常用的切入点表达式必须精确到具体包路径。
            * com.itheima.service.impl.*.*(..)
        -->
        <aop:pointcut id="pointcut" expression="execution(* com.itheima.service.impl.*.*(..))"/>
        <!--
        <aop:aspect 配置一个切面
        id="aspect" 当前切面的唯一标示
        ref="myAdvice" 指定当前切面使用哪个通知
        -->
        <aop:aspect id="aspect" ref="myAdvice">
            <!--
                <aop:before 指定通知在切入点方法中执行的位置
                method="before" 指定当前位置要执行通知中的哪个方法
                pointcut 切入点表达式，指定通知要在哪些切入点方法上执行
            -->
            <aop:before method="before" pointcut-ref="pointcut"/>
            <aop:after-returning method="afterReturning" pointcut-ref="pointcut"/>
            <aop:after-throwing method="afterThrowing" pointcut-ref="pointcut"/>
            <aop:after method="after" pointcut-ref="pointcut"/>
        </aop:aspect>
    </aop:config>
</beans>
```

3.3 spring环绕通知基于xml使用

3.3.1 增强类

```
@Component
public class MyAdvice {
    //需求：对所有的delete方法的参数加1。
    //环绕通知
    public Object around(ProceedingJoinPoint pjp){
        Object obj = null;
        try {
            System.out.println("前置通知");
            //得到当前目标对象的类型
            Class clazz = pjp.getTarget().getClass();
            //得到当前切入点方法名
            String methodName = pjp.getSignature().getName();
            //得到当前方法参数的数组
            Object[] args = pjp.getArgs();
            if(methodName.equals("delete")){
                Integer id = (Integer) args[0];
                //放行方法
                pjp.proceed(new Object[]{id+1});
            }else {
                pjp.proceed(args);
            }
            System.out.println("后置通知");
        }catch (Throwable t){
            t.printStackTrace();
            System.out.println("异常通知");
        }finally {
            System.out.println("最终通知");
        }
        return obj;
    }
}
```

3.3.2 spring配置文件中aop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"
        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">

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

```
    <!--aop基于xml配置-->
    <aop:config>
        <!--配置切入点表达式-->
        <aop:pointcut id="pointcut" expression="execution(* com.itheima.service.impl.*.*(..))"/>
        <!-- 配置切面 -->
        <aop:aspect id="aspect" ref="myAdvice">
            <aop:around method="around" pointcut-ref="pointcut"/>
        </aop:aspect>
    </aop:config>

</beans>
```

3.4 spring的AOP基于注解使用

3.4.1 spring配置文件中注解支持

```
<?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"
        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">

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

    <!--aop注解的支持-->
    <aop:aspectj-autoproxy/>

</beans>
```

3.4.2 spring四大通知基于注解使用

```
@Component
@Aspect
public class MyAdvice {

    //提取切入点表达式
    @Pointcut("execution(* com.itheima.service.impl.*.*(..))")
    public void pc(){}

    //前置通知，就是在切入点方法前执行
    @Before("pc()")
    public void before(){
        System.out.println("前置通知！");
```

```
    }

    //后置通知，就是在切入点方法后执行
    @AfterReturning("execution(* com.itheima.service.impl.*.*(..))")
    public void afterReturning(){
        System.out.println("后置通知！");
    }

    //异常通知，就是在切入点方法出错时执行
    @AfterThrowing("execution(* com.itheima.service.impl.*.*(..))")
    public void afterThrowing(){
        System.out.println("异常通知！");
    }

    //最终通知，就是在切入点方法执行完毕，不管成功与否
    @After("execution(* com.itheima.service.impl.*.*(..))")
    public void after(){
        System.out.println("最终通知！");
    }
}
```

3.4.3 spring环绕通知基于注解使用

```
@Component
@Aspect
public class MyAdvice {

    //需求：对所有的delete方法的参数加1。
    //环绕通知
    @Around("execution(* com.itheima.service.impl.*.*(..))")
    public Object around(ProceedingJoinPoint pjp){
        Object obj = null;
        try {
            System.out.println("前置通知");
            //得到当前目标对象的类型
            Class clazz = pjp.getTarget().getClass();
            //得到当前切入点方法名
            String methodName = pjp.getSignature().getName();
            //得到当前方法参数的数组
            Object[] args = pjp.getArgs();
            if(methodName.equals("delete")){
                Integer id = (Integer) args[0];
                //放行方法
                pjp.proceed(new Object[]{id+1});
            }else {
                pjp.proceed(args);
            }
            System.out.println("后置通知");
        }catch (Throwable t){
            t.printStackTrace();

            System.out.println("异常通知");
```

```
        }finally {
            System.out.println("最终通知");
        }
        return obj;
    }

}
```

3.5 AOP工作流程

- 开发阶段(开发者完成)
  - 开发共性功能，制作成**增强**
  - 开发非共性功能，制作成**切点**
  - 在配置文件中，声明切点与增强间的关系，即**切面**
- 运行阶段(AOP完成)
  - JVM读取配置文件中的信息，监控切入点方法的执行
  - 一旦监控到切入点方法被运行，使用代理机制，动态创建**目标对象**
  - 根据通知类别，在代理对象的对应位置，将对应的增强功能**织入**，完成完整的代码逻辑运行

四、spring中AOP零配置文件使用

```
@Configuration
@ComponentScan("com.itheima")  //替代 <context:component-scan base-package="" />
@EnableAspectJAutoProxy     //替代 <aop:aspectj-autoproxy />
public class AopConfig {

}
```

作业

将转账案例改成spring的AOP方式来完成，要求xml和注解分别实现。