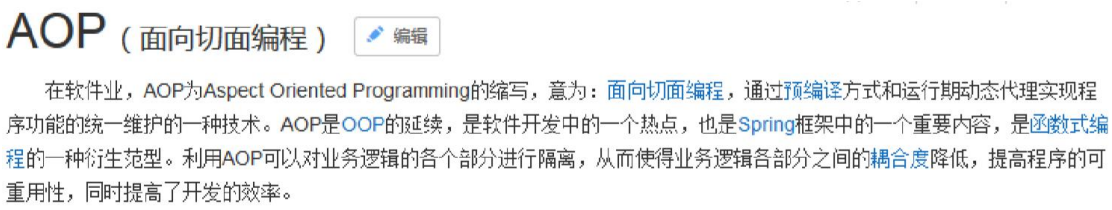
第五讲 Spring 框架之 AOP

一、AOP 概述

1. 什么是 AOP

AOP：全称是 Aspect Oriented Programming 即：面向切面编程。


AOP（面向切面编程） 编辑

在软件业，AOP为Aspect Oriented Programming的缩写，意为：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OOP的延续，是软件开发中的一个热点，也是Spring框架中的一个重要内容，是函数式编程的一种衍生范型。利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。

简单的说它就是把我们程序重复的代码抽取出来，在需要执行的时候，使用动态代理的技术，在不修改源码的基础上，对我们的已有方法进行增强。

2. AOP 的作用及优势

作用： 在程序运行期间，不修改源码对已有方法进行增强。

优势：

减少重复代码

提高开发效率

维护方便

3. AOP 的具体应用

① 案例中问题

- 客户的业务层实现类

```
/**
 * 账户的业务层实现类
*/
```

```
public class AccountServiceImpl implements IAccountService {
    private IAccountDao accountDao;

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

 @Override
public void saveAccount(Account account) throws SQLException {
    accountDao.save(account);
  }

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

 @Override
  public void deleteAccount(Integer accountId) throws SQLException{
    accountDao.delete(accountId);
 }

 @Override
  public Account findAccountById(Integer accountId) throws SQLException {
    return accountDao.findById(accountId);
  }

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

- 客户的业务层实现类

```
/**
 * 账户的业务层实现类
```

```
 */
public class AccountServiceImpl implements IAccountService {
    private IAccountDao accountDao;

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

@Override
 public void saveAccount(Account account) throws SQLException {
   accountDao.save(account);
 }

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

@Override
 public void deleteAccount(Integer accountId) throws SQLException{
   accountDao.delete(accountId);
}

@Override
 public Account findAccountById(Integer accountId) throws SQLException {
   return accountDao.findById(accountId);
 }

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

问题：

事务被自动控制了。换言之，我们使用了 connection 对象的 setAutoCommit(true)

此方式控制事务，如果我们每次都执行一条 sql 语句，没有问题，但是如果业务方法一次

要执行多条 sql 语句，这种方式就无法实现功能了。

业务层接口

```
/**
 * 转账  * @param sourceName  * @param targetName  * @param money  */
void transfer(String sourceName,String targetName,Float money);

业务层实现类：  @Override public void transfer(String sourceName, String targetName, Float money) {  //根据名称查询两个账户信息  Account source = accountDao.findByName(sourceName);  Account target = accountDao.findByName(targetName);
 //转出账户减钱，转入账户加钱  source.setMoney(source.getMoney()-money);
target.setMoney(target.getMoney()+money);
 //更新两个账户  accountDao.update(source);
 int i=1/0; //模拟转账异常  accountDao.update(target); }
```

在执行时，由于执行有异常，转账失败。但是因为每次执行持久层方法都是独立事务，导致无法实现事务控制（不符合事务的一致性）

解决办法： 让业务层来控制事务的提交和回滚。

② 改造后的业务层实现类

注：此处没有使用 spring 的 IoC.

```
/**
 * 账户的业务层实现类
 */
 public class AccountServiceImpl implements IAccountService {
 private IAccountDao accountDao = new AccountDaoImpl();

 @Override
  public void saveAccount(Account account) {
    try {
      TransactionManager.beginTransaction();
      accountDao.save(account);
    TransactionManager.commit();
    } catch (Exception e) {
```

```
    TransactionManager.rollback();
    e.printStackTrace();
   }finally {
    TransactionManager.release();
  }
   }

@Override
 public void updateAccount(Account account) {
  try {
    TransactionManager.beginTransaction();
    accountDao.update(account);
    TransactionManager.commit();
  } catch (Exception e) {
    TransactionManager.rollback();
    e.printStackTrace();
  }finally {
    TransactionManager.release();
  }
 }

@Override
 public void deleteAccount(Integer accountId) {
 try {
    TransactionManager.beginTransaction();
  accountDao.delete(accountId);
    TransactionManager.commit();
  } catch (Exception e) {
    TransactionManager.rollback();
  e.printStackTrace();
  }finally {
    TransactionManager.release();
  }
 }
@Override
 public Account findAccountById(Integer accountId) {
```

```
    Account account = null;
    try {
      TransactionManager.beginTransaction();
      account = accountDao.findById(accountId);
      TransactionManager.commit();
    return account;
    } catch (Exception e) {
      TransactionManager.rollback();
      e.printStackTrace();
    }finally {
      TransactionManager.release();
   }
    return null;
   }

  @Override
   public List<Account> findAllAccount() {
    List<Account> accounts = null;
    try {
      TransactionManager.beginTransaction();
      accounts = accountDao.findAll();
      TransactionManager.commit();
      return accounts;
    } catch (Exception e) {
      TransactionManager.rollback();
    e.printStackTrace();
    }finally {
      TransactionManager.release();
    }
    return null;
   }

  @Override
   public void transfer(String sourceName, String targetName, Float money) {
    try {
      TransactionManager.beginTransaction();
```

```
    Account source = accountDao.findByName(sourceName);
     Account target = accountDao.findByName(targetName);
     source.setMoney(source.getMoney()-money);
    target.setMoney(target.getMoney()+money);
     accountDao.update(source);
    int i=1/0;
    accountDao.update(target);
     TransactionManager.commit();
    } catch (Exception e) {
     TransactionManager.rollback();
    e.printStackTrace();
    }finally {
     TransactionManager.release();
   }
 }
}
```

TransactionManager 类的代码

```
public class TransactionManager {
 //定义一个 DBAssit
private static DBAssit dbAssit =   new DBAssit(C3P0Utils.getDataSource(),true);



 //开启事务
  public static void beginTransaction() {
   try {
    dbAssit.getCurrentConnection().setAutoCommit(false);
   } catch (SQLException e) {
    e.printStackTrace();
   }
  }
 //提交事务
  public static void commit() {
   try {
    dbAssit.getCurrentConnection().commit();
   } catch (SQLException e) {
```

```
      e.printStackTrace();
    }
  }

 //回滚事务
 public static void rollback() {
    try {
      dbAssit.getCurrentConnection().rollback();
    } catch (SQLException e) {
      e.printStackTrace();
  }
  }
 //释放资源
  public static void release() {
    try {
      dbAssit.releaseConnection();
    } catch (Exception e) {
      e.printStackTrace();
  }
  }
}
```

新的问题

上一小节的代码，通过对业务层改造，已经可以实现事务控制了，但是由于我们添加了事务控制，也产生了一个新的问题： 业务层方法变得臃肿了，里面充斥着很多重复代码。并且业务层方法和事务控制方法耦合了。 试想一下，如果我们此时提交，回滚，释放资源中任何一个方法名变更，都需要修改业务层的代码，况且这还只是一个业务层实现类，而实际的项目中这种业务层实现类可能有十几个甚至几十个。

4. 动态代理

① 动态代理的特点

字节码随用随创建，随用随加载。

它与静态代理的区别也在于此。因为静态代理是字节码一上来就创建好，并完成加载。

装饰者模式就是静态代理的一种体现。

② 动态代理常用两种方式

- 基于接口的动态代理

提供者：JDK 官方的 Proxy 类。

要求：被代理类最少实现一个接口。

- 基于子类的动态代理

提供者：第三方的 CGLib，如果报 asmxxxx 异常，需要导入 asm.jar。

要求：被代理类不能用 final 修饰的类（最终类）。

③ 使用 JDK 官方的 Proxy 类创建代理对象

此处使用的是一个演员的例子：

在很久以前，演员和剧组都是直接见面联系的。没有中间人环节。

而随着时间的推移，产生了一个新兴职业：经纪人（中间人），这个时候剧组再想找演员就需要通过经纪 人来找了。下面我们就用代码演示出来。

```
/**
 * 一个经纪公司的要求:
  *    能做基本的表演和危险的表演
*/
 public interface IActor {
   /**
   * 基本演出
   * @param money
   */
   public void basicAct(float money);
   /**
   * 危险演出
    * @param money
    */
   public void dangerAct(float money); }

/**
 * 一个演员
```

```
   */
//实现了接口，就表示具有接口中的方法实现。即：符合经纪公司的要求
public class Actor implements IActor{
      public void basicAct(float money){
   System.out.println("拿到钱，开始基本的表演："+money);
 }
      public void dangerAct(float money){
   System.out.println("拿到钱，开始危险的表演："+money);
 }
}
```

```
public class Client {
      public static void main(String[] args) {
   //一个剧组找演员：
    final Actor actor = new Actor();//直接
         /**
    * 代理：
     *   间接。
     * 获取代理对象：
     *   要求：
     *     被代理类最少实现一个接口
     * 创建的方式
     *     Proxy.newProxyInstance(三个参数)
     * 参数含义：
     *   ClassLoader：和被代理对象使用相同的类加载器。
     *   Interfaces：和被代理对象具有相同的行为。实现相同的接口。
     *   InvocationHandler：如何代理。
     *      策略模式：使用场景是：
    *          数据有了，目的明确。
    *          如何达成目标，就是策略。
     *
            */
    IActor proxyActor = (IActor) Proxy.newProxyInstance(
                actor.getClass().getClassLoader(),
                actor.getClass().getInterfaces(),
                new InvocationHandler() {
```

```
        /**
        * 执行被代理对象的任何方法，都会经过该方法。
         * 此方法有拦截的功能。
         *
        * 参数：
         *   proxy：代理对象的引用。不一定每次都用得到
        *   method：当前执行的方法对象
         *   args：执行方法所需的参数
        * 返回值：
         *   当前执行方法的返回值
         */
       @Override
        public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
        String name = method.getName();
        Float money = (Float) args[0];
        Object rtValue = null;
        //每个经纪公司对不同演出收费不一样，此处开始判断
         if("basicAct".equals(name)){
          //基本演出，没有 2000 不演
          if(money > 2000){
           //看上去剧组是给了 8000，实际到演员手里只有 4000
           //这就是我们没有修改原来 basicAct 方法源码，对方法进行了增强
           rtValue = method.invoke(actor, money/2);
         }
        }
        if("dangerAct".equals(name)){
         //危险演出,没有 5000 不演
         if(money > 5000){
          //看上去剧组是给了 50000，实际到演员手里只有 25000
          //这就是我们没有修改原来 dangerAct 方法源码，对方法进行了增强
           rtValue = method.invoke(actor, money/2);
         }
        }
        return rtValue;
        }
```

```
    });
    //没有经纪公司的时候，直接找演员。
  //    actor.basicAct(1000f);
//    actor.dangerAct(5000f);
    //剧组无法直接联系演员，而是由经纪公司找的演员
      proxyActor.basicAct(8000f);
      proxyActor.dangerAct(50000f);
    }
}
```

④ 使用 CGLib 的 Enhancer 类创建代理对象

还是那个演员的例子，只不过不让他实现接口。

```
/**
 * 一个演员 */
public class Actor{//没有实现任何接口
       public void basicAct(float money){
    System.out.println("拿到钱，开始基本的表演："+money);
    }
      public void dangerAct(float money){
    System.out.println("拿到钱，开始危险的表演："+money);
  }
}


public class Client {
    /**
    * 基于子类的动态代理
      *    要求：
      *      被代理对象不能是最终类
      *    用到的类：
      *      Enhancer
    *    用到的方法：
      *      create(Class, Callback)
    *    方法的参数：
      *      Class：被代理对象的字节码
```

```
    *     Callback：如何代理
    * @param args
    */
   public static void   main(String[] args) {
    final Actor actor = new Actor();
        Actor cglibActor = (Actor) Enhancer.create(actor.getClass(),
         new MethodInterceptor() {
     /**
     * 执行被代理对象的任何方法，都会经过该方法。在此方法内部就可以对被代理对象的任何 方法进行增强。
      *
     * 参数：
      *   前三个和基于接口的动态代理是一样的。
      *   MethodProxy：当前执行方法的代理对象。
      * 返回值：
      *   当前执行方法的返回值
      */
     @Override
     public Object intercept(Object proxy, Method method, Object[] args, MethodProxy methodProxy) throws Throwable {
       String name = method.getName();
       Float money = (Float) args[0];
     Object rtValue = null;
     if("basicAct".equals(name)){
      //基本演出
        if(money > 2000){
          rtValue = method.invoke(actor, money/2);
       }
       }
```

解决案例中的问题

```
/**
 * 用于创建客户业务层对象工厂（当然也可以创建其他业务层对象，只不过我们此处不做那么繁琐）
*/
 public class BeanFactory {
```

```
    /**
  * 创建账户业务层实现类的代理对象
   * @return
   */
  public static IAccountService getAccountService() {
   //1.定义被代理对象
   final IAccountService accountService = new AccountServiceImpl();
   //2.创建代理对象
   IAccountService proxyAccountService = (IAccountService)
Proxy.newProxyInstance(accountService.getClass().getClassLoader(),
          accountService.getClass().getInterfaces(),new InvocationHandler()
{          /**
          * 执行被代理对象的任何方法，都会经过该方法。
           * 此处添加事务控制
           */
          @Override
          public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
           Object rtValue = null;
           try {
           //开启事务
            TransactionManager.beginTransaction();
           //执行业务层方法
            rtValue = method.invoke(accountService, args);
           //提交事务
            TransactionManager.commit();
          }catch(Exception e) {
           //回滚事务
            TransactionManager.rollback();
          e.printStackTrace();
           }finally {
           //释放资源
            TransactionManager.release();
           }
           return rtValue;
          }
```

```
        });
    return proxyAccountService;
}
}
```

改造完成之后，业务层用于控制事务的重复代码就都可以删掉了。

二、Spring 中的 AOP

学习 spring 的 aop，就是通过配置的方式，实现上一章节的功能。

关于代理的选择

在 spring 中，框架会根据目标类是否实现了接口来决定采用哪种动态代理的方式。

1. AOP 相关术语

- Joinpoint(连接点): 所谓连接点是指那些被拦截到的点。在 spring 中,这些点指的是方法,因为 spring 只支持方法类型的 连接点。
- Pointcut(切入点): 所谓切入点是指我们要对哪些 Joinpoint 进行拦截的定义
- Advice(通知/增强): 所谓通知是指拦截到 Joinpoint 之后所要做的事情就是通知。通知的类型：前置通知,后置通知,异常通知,最终通知,环绕通知。
- Introduction(引介): 引介是一种特殊的通知在不修改类代码的前提下, Introduction 可以在运行期为类动态地添加一些方 法或 Field。
- Target(目标对象): 代理的目标对象。
- Weaving(织入): 是指把增强应用到目标对象来创建新的代理对象的过程。spring 采用动态代理织入，而 AspectJ 采用编译期织入和类装载期织入。
- Proxy（代理）： 一个类被 AOP 织入增强后，就产生一个结果代理类。
- Aspect(切面): 是切入点和通知（引介）的结合。

2. 基于 XML 的 AOP 配置

示例：

我们在学习 spring 的 aop 时，采用账户转账作为示例。

并且把 spring 的 ioc 也一起应用进来。

① 环境搭建

沿用上一章节中的代码：包含了实体类，业务层和持久层代码

② 创建 spring 的配置文件并导入约束

```
<?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: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/aop
http://www.springframework.org/schema/aop/spring-aop.xsd">
</beans>
```

③ 配置 spring 的 ioc

```
<!-- 配置 service -->
 <bean id="accountService"
class="com.itheima.service.impl.AccountServiceImpl">
 <property name="accountDao" ref="accountDao"></property>
 </bean>

<!-- 配置 dao -->
 <bean id="accountDao" class="com.itheima.dao.impl.AccountDaoImpl">
 <property name="dbAssit" ref="dbAssit"></property>
</bean>
<!-- 配置数据库操作对象 -->
 <bean id="dbAssit" class="com.itheima.dbassit.DBAssit">
 <property name="dataSource" ref="dataSource"></property>
 <!-- 指定 connection 和线程绑定 -->
  <property name="useCurrentConnection" value="true"></property>
</bean>
```

```
<!-- 配置数据源 -->
 <bean                                                    id="dataSource"
class="com.mchange.v2.c3p0.ComboPooledDataSource">
  <property name="driverClass" value="com.mysql.jdbc.Driver"></property>
  <property name="jdbcUrl" value="jdbc:mysql:///spring_day02"></property>
 <property name="user" value="root"></property>
  <property name="password" value="1234"></property>
 </bean>
```

④ 抽取公共代码制作成通知

```
/**
 * 事务控制类
 */
 public class TransactionManager {
 //定义一个 DBAssit
  private DBAssit dbAssit ;

 public void setDbAssit(DBAssit dbAssit) {
    this.dbAssit = dbAssit;
  }

 //开启事务
  public void beginTransaction() {
    try {
      dbAssit.getCurrentConnection().setAutoCommit(false);
    } catch (SQLException e) {
      e.printStackTrace();
  }
  }
    //提交事务
  public void commit() {
    try {
      dbAssit.getCurrentConnection().commit();
    } catch (SQLException e) {
```

```
      e.printStackTrace();
    }
  }

 //回滚事务
  public void rollback() {
    try {
      dbAssit.getCurrentConnection().rollback();
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }
 //释放资源
  public void release() {
    try {
      dbAssit.releaseConnection();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

⑤ 配置步骤

- 第一步：把通知类用 bean 标签配置起来

```
<!-- 配置通知 -->
 <bean id="txManager" class="com.itheima.utils.TransactionManager">
  <property name="dbAssit" ref="dbAssit"></property>
 </bean>
```

- 第二步：使用 aop:config 声明 aop 配置

aop:config:

作用：用于声明开始 aop 的配置

```
<aop:config>
 <!-- 配置的代码都写在此处 -->
  </aop:config>
```

- 第三步：使用 aop:aspect 配置切面

aop:aspect:

作用：　用于配置切面。

属性：

id：给切面提供一个唯一标识。

ref：引用配置好的通知类 bean 的 id。

```
<aop:aspect id="txAdvice" ref="txManager">
  <!--配置通知的类型要写在此处-->
 </aop:aspect>
```

- 第四步：使用 aop:pointcut 配置切入点表达式

aop:pointcut：

作用：　用于配置切入点表达式。就是指定对哪些类的哪些方法进行增强。

属性：

expression：用于定义切入点表达式。

id：用于给切入点表达式提供一个唯一标识

```
<aop:pointcut expression="execution( public void com.itheima.service.impl.AccountServiceImpl.transfer( java.lang.String, java.lang.String, java.lang.Float) )" id="pt1"/>
```

- 第五步：使用 aop:xxx 配置对应的通知类型

➢ **aop:before**

作用：　用于配置前置通知。指定增强的方法在切入点方法之前执行

属性：

method:用于指定通知类中的增强方法名称

ponitcut-ref：用于指定切入点的表达式的引用

poinitcut：用于指定切入点表达式

执行时间点：　切入点方法执行之前执行

<aop:before method="beginTransaction" pointcut-ref="pt1"/>

➢ **aop:after-returning**

作用：　用于配置后置通知

属性：

method：指定通知中方法的名称。

pointct：定义切入点表达式

pointcut-ref：指定切入点表达式的引用

执行时间点：　切入点方法正常执行之后。它和异常通知只能有一个执行

<aop:after-returning method="commit" pointcut-ref="pt1"/>

- **aop:after-throwing**

  作用：　用于配置异常通知

  属性：

  method：指定通知中方法的名称。

  pointct：定义切入点表达式

  pointcut-ref：指定切入点表达式的引用

  执行时间点：　切入点方法执行产生异常后执行。它和后置通知只能执行一个

  <aop:after-throwing method="rollback" pointcut-ref="pt1"/>

- **aop:after**

  作用：　用于配置最终通知

  属性：

  method：指定通知中方法的名称。

  pointct：定义切入点表达式

  pointcut-ref：指定切入点表达式的引用

  执行时间点：　无论切入点方法执行时是否有异常，它都会在其后面执行。

  <aop:after method="release" pointcut-ref="pt1"/>

⑥ 切入点表达式说明

execution:匹配方法的执行(常用)

execution(表达式)

表达式语法：execution([修饰符] 返回值类型 包名.类名.方法名(参数))

写法说明：

全匹配方式：

public void com.itheima.service.impl.AccountServiceImpl.saveAccount(com.itheima.domain.Account)

访问修饰符可以省略

void com.itheima.service.impl.AccountServiceImpl.saveAccount(com.itheima.domain.Account)

返回值可以使用*号，表示任意返回值

* com.itheima.service.impl.AccountServiceImpl.saveAccount(com.itheima.domain.Account)

包名可以使用*号，表示任意包，但是有几级包，需要写几个*

* *.*.*.*.AccountServiceImpl.saveAccount(com.itheima.domain.Account)

使用..来表示当前包，及其子包

* com..AccountServiceImpl.saveAccount(com.itheima.domain.Account)

类名可以使用*号，表示任意类

* com..*.saveAccount(com.itheima.domain.Account)

方法名可以使用*号，表示任意方法 * com..*.*(com.itheima.domain.Account)

参数列表可以使用*，表示参数可以是任意数据类型，但是必须有参数 * com..*.*(*)

参数列表可以使用..表示有无参数均可，有参数可以是任意类型 * com..*.*(..)

全通配方式： * *..*.*(..)

注： 通常情况下，我们都是对业务层的方法进行增强，所以切入点表达式都是切到业务层实现类。 execution(* com.itheima.service.impl.*.*(..))

⑦ 环绕通知

- 配置方式:

```
<aop:config>
  <aop:pointcut expression="execution(* com.itheima.service.impl.*.*(..))" id="pt1"/>
    <aop:aspect id="txAdvice" ref="txManager">
  <!-- 配置环绕通知 -->
    <aop:around method="transactionAround" pointcut-ref="pt1"/>
 </aop:aspect>
</aop:config>
```

- aop:around:

  作用： 用于配置环绕通知

  属性:

  method：指定通知中方法的名称。

  pointct：定义切入点表达式

  pointcut-ref：指定切入点表达式的引用

  说明:

  它是 spring 框架为我们提供的一种可以在代码中手动控制增强代码什么时候执行的方式。

  注意： 通常情况下，环绕通知都是独立使用的

```
/**
```

```
 * 环绕通知
 * @param pjp
 *  spring 框架为我们提供了一个接口：ProceedingJoinPoint，它可以作为环绕通知的方法参数。
 *  在环绕通知执行时，spring 框架会为我们提供该接口的实现类对象，我们直接使用就行。
 * @return
  */
 public Object transactionAround(ProceedingJoinPoint pjp) {
 //定义返回值
  Object rtValue = null;
  try {
   //获取方法执行所需的参数
   Object[] args = pjp.getArgs();
  //前置通知：开启事务
   beginTransaction();
  //执行方法
   rtValue = pjp.proceed(args);
  //后置通知：提交事务
   commit();
  }catch(Throwable e) {
   //异常通知：回滚事务
   rollback();
   e.printStackTrace();
  }finally {
  //最终通知：释放资源
   release();
  }
  return rtValue;
}
```

3. 基于注解的 AOP 配置

① 环境搭建

- 第一步 拷贝上一小节的工程
- 第二步 在配置文件中导入 context 的名称空间

```
<?xml version="1.0" encoding="UTF-8"?> <beans
xmlns="http://www.springframework.org/schema/beans"
 xmlns:aop="http://www.springframework.org/schema/aop"
xmlns:context="http://www.springframework.org/schema/context"
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
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop.xsd
        http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

 <!-- 配置数据库操作对象 -->
  <bean id="dbAssit" class="com.itheima.dbassit.DBAssit">
   <property name="dataSource" ref="dataSource"></property>
  <!-- 指定 connection 和线程绑定 -->
   <property name="useCurrentConnection" value="true"></property>  </bean>
 <!-- 配置数据源 -->
  <bean id="dataSource"
class="com.mchange.v2.c3p0.ComboPooledDataSource">
  <property name="driverClass" value="com.mysql.jdbc.Driver"></property>
   <property name="jdbcUrl" value="jdbc:mysql:///spring_day02"></property>
   <property name="user" value="root"></property>
   <property name="password" value="1234"></property>
  </bean>
 </beans>
```

- 第三步：把资源使用注解配置

```
/**
```

```
 * 账户的业务层实现类
 */
@Service("accountService")
 public class AccountServiceImpl implements IAccountService {
       @Autowired
   private IAccountDao accountDao;
}

/**
 * 账户的持久层实现类
 */
@Repository("accountDao")
 public class AccountDaoImpl   implements IAccountDao {
 @Autowired
   private DBAssit dbAssit ;
 }
```

- 第四步：在配置文件中指定 spring 要扫描的包

```
<!-- 告知 spring，在创建容器时要扫描的包 -->
<context:component-scan
base-package="com.itheima"></context:component-scan>
```

② 配置步骤

- 第一步：把通知类也使用注解配置

```
/**
 * 事务控制类
 */
@Component("txManager")
public class TransactionManager {
 //定义一个 DBAssit
   @Autowired
   private DBAssit dbAssit ;
 }
```

- 第二步：在通知类上使用@Aspect 注解声明为切面

作用：　把当前类声明为切面类。

```
/**
 * 事务控制类
 */
 @Component("txManager")
@Aspect//表明当前类是一个切面类
public class TransactionManager {
 //定义一个 DBAssit
  @Autowired
  private DBAssit dbAssit ;
 }
```

- 第三步：在增强的方法上使用注解配置通知
  - @Before

作用：　把当前方法看成是前置通知。

属性：　value：用于指定切入点表达式，还可以指定切入点表达式的引用。

```
//开启事务
  @Before("execution(* com.itheima.service.impl.*.*(..))")
  public void beginTransaction() {
   try {
    dbAssit.getCurrentConnection().setAutoCommit(false);
   } catch (SQLException e) {
    e.printStackTrace();
   }
  }
```

  - @AfterReturning

作用：　把当前方法看成是后置通知。

属性：　value：用于指定切入点表达式，还可以指定切入点表达式的引用

```
//提交事务
  @AfterReturning("execution(* com.itheima.service.impl.*.*(..))")
  public void commit() {
  try {
    dbAssit.getCurrentConnection().commit();
   } catch (SQLException e) {
    e.printStackTrace();
   }
 }
```

- @AfterThrowing

作用：　把当前方法看成是异常通知。

属性：　value：用于指定切入点表达式，还可以指定切入点表达式的引用

```
//回滚事务
  @AfterThrowing("execution(* com.itheima.service.impl.*.*(..))")
  public void rollback() {
   try {
    dbAssit.getCurrentConnection().rollback();
   } catch (SQLException e) {
    e.printStackTrace();
   }
  }
```

- @After

作用：　把当前方法看成是最终通知。

属性：　value：用于指定切入点表达式，还可以指定切入点表达式的引用

```
//释放资源
  @After("execution(* com.itheima.service.impl.*.*(..))")
  public void release() {
   try {
    dbAssit.releaseConnection();
   } catch (Exception e) {
    e.printStackTrace();
   }
  }
```

- 第四步：在 spring 配置文件中开启 spring 对注解 AOP 的支持

```
<!-- 开启 spring 对注解 AOP 的支持 -->
<aop:aspectj-autoproxy/>
```

③ 环绕通知注解配置

@Around

作用：　把当前方法看成是环绕通知。

属性：　value：用于指定切入点表达式，还可以指定切入点表达式的引用。

```
/**
 * 环绕通知
```

```
 */
  @Around("execution(* com.itheima.service.impl.*.*(..))")
  public Object transactionAround(ProceedingJoinPoint pjp) {
  //定义返回值
    Object rtValue = null;
    try {
    //获取方法执行所需的参数
      Object[] args = pjp.getArgs();
    //前置通知：开启事务
      beginTransaction();
    //执行方法
      rtValue = pjp.proceed(args);
    //后置通知：提交事务
      commit();
    }catch(Throwable e) {
    //异常通知：回滚事务
      rollback();
      e.printStackTrace();
    }finally {
      //最终通知：释放资源
      release();
    }
    return rtValue;
  }
```

④ 切入点表达式注解

@Pointcut

作用：　指定切入点表达式

属性：　value：指定表达式的内容

```
@Pointcut("execution(* com.itheima.service.impl.*.*(..))")
 private void pt1() {}
```

引用方式：

```
/**
   * 环绕通知
```

```
 */
 @Around("pt1()")//注意：千万别忘了写括号
  public Object transactionAround(ProceedingJoinPoint pjp) {
    //定义返回值
    Object rtValue = null;
    try {
    //获取方法执行所需的参数
      Object[] args = pjp.getArgs();
    //前置通知：开启事务
      beginTransaction();
    //执行方法
      rtValue = pjp.proceed(args);
    //后置通知：提交事务
      commit();
    }catch(Throwable e) {
    //异常通知：回滚事务
      rollback();
      e.printStackTrace();
    }finally {
      //最终通知：释放资源
      release();
    }
    return rtValue;
  }
```

⑤ 不使用 XML 的配置方式

```
@Configuration
@ComponentScan(basePackages="com.itheima")
@EnableAspectJAutoProxy
public class SpringConfiguration {
 }
```

三、 Spring 中的事务控制

1. 基于 XML 的声明式事务控制

① 环境搭建

- 第一步：jar 包
- 第二步 创建 spring 的配置文件并导入约束

此处需要导入 aop 和 tx 两个名称空间

```
<?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: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/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">
 </beans>
```

- 第三步：准备数据库表和实体类

创建数据库：

```
create database spring_day04; use spring_day04;
```

创建表：

```
create table account(
   id int primary key auto_increment,
   name varchar(40),
  money float
)character set utf8 collate utf8_general_ci;
```

```
/**
 * 账户的实体
  */
```

```
public class Account implements Serializable {
  private Integer id;
  private String name;
  private Float money;
  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 Float getMoney() {
    return money;
  }
  public void setMoney(Float money) {
    this.money = money;
  }
  @Override
  public String toString() {
    return "Account [id=" + id + ", name=" + name + ", money=" + money + "]";
  }
}
```

第四步：编写业务层接口和实现类

```
/**
 * 账户的业务层接口
  */
 public interface IAccountService {
     /**
  * 根据 id 查询账户信息
    * @param id
```

```
    * @return
    */
 Account findAccountById(Integer id);//查
     /**
  * 转账
    * @param sourceName 转出账户名称
    * @param targeName   转入账户名称
    * @param money     转账金额
    */
 void transfer(String sourceName,String targeName,Float money);//增删改
 }

/**
 * 账户的业务层实现类
  */
 public class AccountServiceImpl implements IAccountService {
    private IAccountDao accountDao;
     public void setAccountDao(IAccountDao accountDao) {
    this.accountDao = accountDao;
  }

 @Override
  public Account findAccountById(Integer id) {
    return accountDao.findAccountById(id);
  }

 @Override
  public void transfer(String sourceName, String targeName, Float money) {
  //1.根据名称查询两个账户
    Account source = accountDao.findAccountByName(sourceName);
    Account target = accountDao.findAccountByName(targeName);
  //2.修改两个账户的金额
    source.setMoney(source.getMoney()-money);//转出账户减钱
    target.setMoney(target.getMoney()+money);//转入账户加钱
    //3.更新两个账户
    accountDao.updateAccount(source);
```

```
    int i=1/0;
    accountDao.updateAccount(target);
  }
}
```

第五步：编写 Dao 接口和实现类

```
/**
 * 账户的持久层接口
  */
 public interface IAccountDao {
     /**
  * 根据 id 查询账户信息
   * @param id
   * @return
   */
  Account findAccountById(Integer id);

 /**
  * 根据名称查询账户信息
   * @return
  */
 Account findAccountByName(String name);
 /**
  * 更新账户信息
   * @param account
   */
  void updateAccount(Account account);
}
/**
 * 账户的持久层实现类
  * 此版本 dao，只需要给它的父类注入一个数据源
  */
 public class AccountDaoImpl extends JdbcDaoSupport implements IAccountDao {
 @Override
  public Account findAccountById(Integer id) {
   List<Account> list = getJdbcTemplate().query("select * from account where id
```

```
= ? ",new AccountRowMapper(),id);
    return list.isEmpty()?null:list.get(0);
  }

 @Override
  public Account findAccountByName(String name) {
   List<Account> list =  getJdbcTemplate().query("select * from account where name = ? ",new AccountRowMapper(),name);
    if(list.isEmpty()){
     return null;
    }
    if(list.size()>1){
     throw new RuntimeException("结果集不唯一，不是只有一个账户对象");
    }
    return list.get(0);
  }

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

/**
 * 账户的封装类 RowMapper 的实现类
 */
public class AccountRowMapper implements RowMapper<Account>{

 @Override
  public Account mapRow(ResultSet rs, int rowNum) throws SQLException {
   Account account = new Account();
   account.setId(rs.getInt("id"));
   account.setName(rs.getString("name"));
  account.setMoney(rs.getFloat("money"));
   return account;
```

```
 }
}
```

第六步：在配置文件中配置业务层和持久层对象

```
<!-- 配置 service -->
 <bean id="accountService"
class="com.itheima.service.impl.AccountServiceImpl">
   <property name="accountDao" ref="accountDao"></property>
</bean>
<!-- 配置 dao -->
 <bean id="accountDao" class="com.itheima.dao.impl.AccountDaoImpl">
 <!-- 注入 dataSource -->
   <property name="dataSource" ref="dataSource"></property>
 </bean>
<!-- 配置数据源 -->
 <bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
   <property name="driverClassName" value="com.mysql.jdbc.Driver"></property>
   <property name="url" value="jdbc:mysql:///spring_day04"></property>
   <property name="username" value="root"></property>
   <property name="password" value="1234"></property>
 </bean>
```

② 配置步骤

- 第一步：配置事务管理器

```
<!-- 配置一个事务管理器 -->
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
 <!-- 注入 DataSource -->
   <property name="dataSource" ref="dataSource"></property>
 </bean>
```

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.hibernate5.HibernateTransactionManager 使用 Hibernate 版本进行持久化数据时使用

- 第二步：配置事务的通知引用事务管理器

```
<!-- 事务的配置 -->
 <tx:advice id="txAdvice" transaction-manager="transactionManager">
</tx:advice>
```

- 第三步：配置事务的属性

```
<!--在 tx:advice 标签内部 配置事务的属性 -->
 <tx:attributes>
<!-- 指定方法名称：是业务核心方法
    read-only：是否是只读事务。默认 false，不只读。
   isolation：指定事务的隔离级别。默认值是使用数据库的默认隔离级别。
    propagation：指定事务的传播行为。
   timeout：指定超时时间。默认值为：-1。永不超时。
   rollback-for：用于指定一个异常，当执行产生该异常时，事务回滚。产生其他异常，事务不回滚。 没有默认值，任何异常都回滚。
   no-rollback-for：用于指定一个异常，当产生该异常时，事务不回滚，产生其他异常时，事务回 滚。没有默认值，任何异常都回滚。
   -->
   <tx:method name="*" read-only="false" propagation="REQUIRED"/>
   <tx:method name="find*" read-only="true" propagation="SUPPORTS"/>
</tx:attributes>
```

事务的传播行为

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）

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

REQUERS_NEW:新建事务，如果当前在事务中，把当前事务挂起。

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。

是否是只读事务

建议查询时设置为只读。

- 第四步：配置 AOP 切入点表达式

```
<!-- 配置 aop -->
 <aop:config>
 <!-- 配置切入点表达式 -->
  <aop:pointcut expression="execution(* com.itheima.service.impl.*.*(..))"
id="pt1"/>
</aop:config>
```

- 第五步：配置切入点表达式和事务通知的对应关系

```
<!-- 在 aop:config 标签内部：建立事务的通知和切入点表达式的关系 -->
 <aop:advisor advice-ref="txAdvice" pointcut-ref="pt1"/>
```

2. 基于注解的配置方式

① 环境搭建

- 第一步： jar 包
- 第二步：创建 spring 的配置文件导入约束并配置扫描的包

```
<?xml version="1.0" encoding="UTF-8"?>
 <beans xmlns="http://www.springframework.org/schema/beans"
xmlns:aop="http://www.springframework.org/schema/aop"
xmlns:tx="http://www.springframework.org/schema/tx"
xmlns:context="http://www.springframework.org/schema/context"
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
        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/context
http://www.springframework.org/schema/context/spring-context.xsd">
 <!-- 配置 spring 创建容器时要扫描的包 -->
  <context:component-scan
```

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

 <!-- 配置 JdbcTemplate-->
 <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
  <property name="dataSource" ref="dataSource"></property>
  </bean>

 <!-- 配置 spring 提供的内置数据源 -->
  <bean                                                    id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName"
value="com.mysql.jdbc.Driver"></property>
    <property name="url"
value="jdbc:mysql://localhost:3306/spring_day02"></property>
    <property name="username" value="root"></property>
    <property name="password" value="1234"></property>
  </bean>
</beans>
```

- 第三步：创建数据库表和实体类

和基于 xml 的配置相同。略

- 第四步：创建业务层接口和实现类并使用注解让 spring 管理

```
/**
 * 账户的业务层实现类
  */
 @Service("accountService")
 public class AccountServiceImpl implements IAccountService {
  @Autowired
  private IAccountDao accountDao;

 //其余代码和基于 XML 的配置相同
 }
```

- 第五步：创建 Dao 接口和实现类并使用注解让 spring 管理

```
/**
 * 账户的持久层实现类      */ @Repository("accountDao") public class
AccountDaoImpl implements IAccountDao {
```

```
@Autowired
private JdbcTemplate jdbcTemplate;
//其余代码和基于 XML 的配置相同 }
```

② 配置步骤

- 第一步：配置事务管理器并注入数据源

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

- 第二步：在业务层使用@Transactional 注解

```
@Service("accountService")
@Transactional(readOnly=true,propagation=Propagation.SUPPORTS)
 public class AccountServiceImpl implements IAccountService {
 @Autowired
  private IAccountDao accountDao;

 @Override
  public Account findAccountById(Integer id) {
   return accountDao.findAccountById(id);
  }

 @Override   @Transactional(readOnly=false,propagation=Propagation.REQUIRED)
  public void transfer(String sourceName, String targeName, Float money) {
  //1.根据名称查询两个账户
   Account source = accountDao.findAccountByName(sourceName);
   Account target = accountDao.findAccountByName(targeName);
  //2.修改两个账户的金额
   source.setMoney(source.getMoney()-money);//转出账户减钱
   target.setMoney(target.getMoney()+money);//转入账户加钱
   //3.更新两个账户
```

```
    accountDao.updateAccount(source);
    //int i=1/0;
   accountDao.updateAccount(target);
 }
}
```

该注解的属性和 xml 中的属性含义一致。该注解可以出现在接口上，类上和方法上。

出现接口上，表示该接口的所有实现类都有事务支持。

出现在类上，表示类中所有方法有事务支持

出现在方法上，表示方法有事务支持。

以上三个位置的优先级：方法>类>接口

- 第三步：在配置文件中开启 spring 对注解事务的支持

```
<!-- 开启 spring 对注解事务的支持 -->
 <tx:annotation-driven transaction-manager="transactionManager"/>
```

③ 不使用 xml 的配置方式

```
@Configuration
@EnableTransactionManagement
public class SpringTxConfiguration {
 //里面配置数据源，配置 JdbcTemplate,配置事务管理器。在之前的步骤已经写过了。
 }
```