Spring 的循环依赖的理论依据基于 Java 的引用传递，当获得对象的引用时，对象的属性是可以延后设置的，但是构造器必须是在获取引用之前

Spring通过setXxx或者@Autowired方法解决循环依赖其实是通过提前暴露一个ObjectFactory对象来完成的，简单来说ClassA在调用构造器完成对象初始化之后，在调用ClassA的setClassB方法之前就把ClassA实例化的对象通过ObjectFactory提前暴露到Spring容器中。

- Spring容器初始化ClassA通过构造器初始化对象后提前暴露到Spring容器。

```
boolean earlySingletonExposure = (mbd.isSingleton() &&
this.allowCircularReferences &&
        isSingletonCurrentlyInCreation(beanName));
    if (earlySingletonExposure) {
      if (logger.isDebugEnabled()) {
        logger.debug("Eagerly caching bean '" + beanName +
            "' to allow for resolving potential circular references");
      }
      //将初始化后的对象提前已ObjectFactory对象注入到容器中
      addSingletonFactory(beanName, new ObjectFactory<Object>() {
        @Override
        public Object getObject() throws BeansException {
          return getEarlyBeanReference(beanName, mbd, bean);
        }
      });
    }
```

- ClassA调用setClassB方法，Spring首先尝试从容器中获取ClassB，此时ClassB不存在Spring容器中。
- Spring容器初始化ClassB，同时也会将ClassB提前暴露到Spring容器中
- ClassB调用setClassA方法，Spring从容器中获取ClassA ，因为第一步中已经提前暴露了ClassA，因此可以获取到ClassA实例
  - ClassA通过spring容器获取到ClassB，完成了对象初始化操作。
- 这样ClassA和ClassB都完成了对象初始化操作，解决了循环依赖问题。

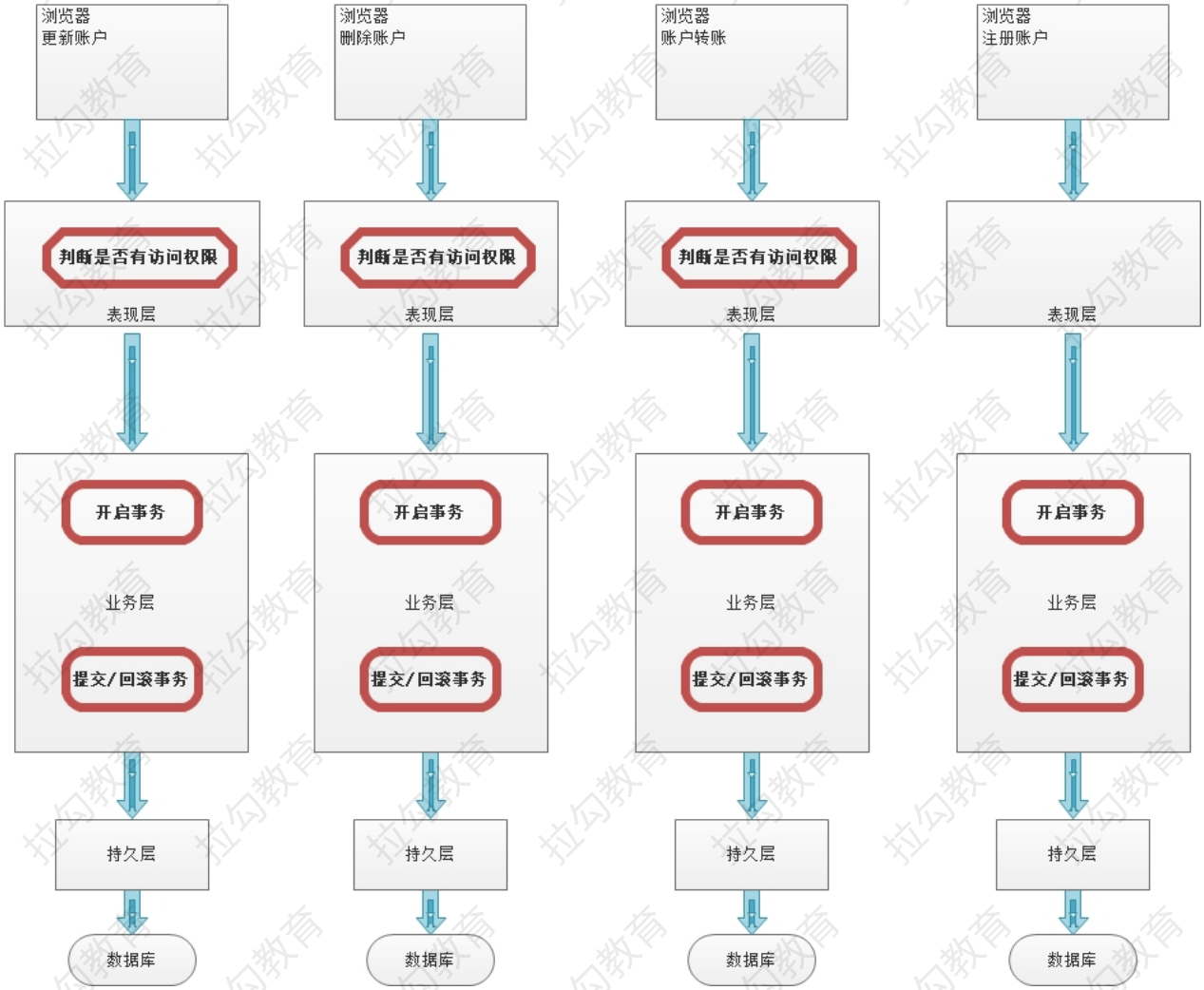
第六部分 Spring AOP 应用

AOP本质：在不改变原有业务逻辑的情况下增强横切逻辑，横切逻辑代码往往是权限校验代码、日志代码、事务控制代码、性能监控代码。

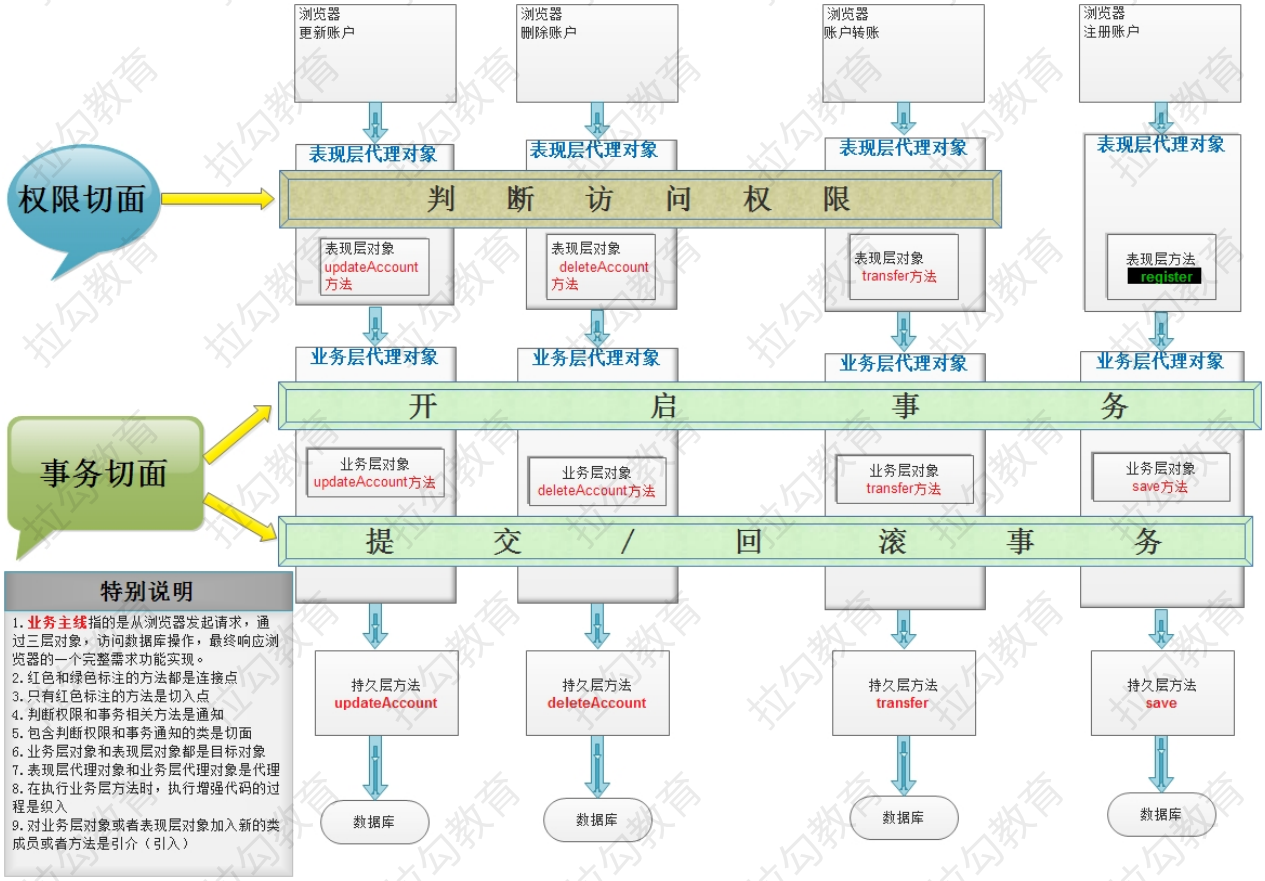
第1节 AOP 相关术语

1.1 业务主线

在讲解AOP术语之前，我们先来看一下下面这两张图，它们就是第三部分案例需求的扩展（针对这些扩展的需求，我们只进行分析，在此基础上去进一步回顾AOP，不进行实现）



上图描述的就是未采用AOP思想设计的程序，当我们红色框中圈定的方法时，会带来大量的重复劳动。程序中充斥着大量的重复代码，使我们程序的独立性很差。而下图中是采用了AOP思想设计的程序，它把红框部分的代码抽取出来的同时，运用动态代理技术，在运行期对需要使用的业务逻辑方法进行增强。


浏览器
更新账户
浏览器
删除账户
浏览器
账户转账
浏览器
注册账户
表现层代理对象
表现层代理对象
表现层代理对象
表现层代理对象
权限切面
判 断 访 问 权 限
表现层对象
updateAccount
方法
表现层对象
deleteAccount
方法
表现层对象
transfer方法
表现层方法
register
业务层代理对象
业务层代理对象
业务层代理对象
业务层代理对象
事务切面
开 启 事 务
业务层对象
updateAccount方法
业务层对象
deleteAccount方法
业务层对象
transfer方法
业务层对象
save方法
提 交 / 回 滚 事 务
特别说明
1. 业务主线指的是从浏览器发起请求，通过三层对象，访问数据库操作，最终响应浏览器的一个完整需求功能实现。
2. 红色和绿色标注的方法都是连接点
3. 只有红色标注的方法是切入点
4. 判断权限和事务相关方法是通知
5. 包含判断权限和事务通知的类是切面
6. 业务层对象和表现层对象都是目标对象
7. 表现层代理对象和业务层代理对象是代理
8. 在执行业务层方法时，执行增强代码的过程是织入
9. 对业务层对象或者表现层对象加入新的类成员或者方法是引介（引入）
持久层方法
updateAccount
持久层方法
deleteAccount
持久层方法
transfer
持久层方法
save
数据库
数据库
数据库
数据库

1.2 AOP 术语

名词	解释
Joinpoint(连接点)	它指的是那些**可以**用于把增强代码加入到业务主线中的点，那么由上图中我们可以看出，这些点指的就是方法。在方法执行的前后通过动态代理技术加入增强的代码。在Spring框架AOP思想的技术实现中，也只支持方法类型的连接点。
Pointcut(切入点)	它指的是那些已经把增强代码加入到业务主线进来之后的连接点。由上图中，我们看出表现层 `transfer` 方法就只是连接点，因为判断访问权限的功能并没有对其增强。
Advice(通知/增强)	它指的是切面类中用于提供增强功能的方法。并且不同的方法增强的时机是不一样的。比如，开启事务肯定要在业务方法执行之前执行；提交事务要在业务方法正常执行之后执行，而回滚事务要在业务方法执行产生异常之后执行等等。那么这些就是通知的类型。其分类有：**前置通知 后置通知 异常通知 最终通知 环绕通知**。
Target(目标对象)	它指的是代理的目标对象。即被代理对象。
Proxy(代理)	它指的是一个类被AOP织入增强后，产生的代理类。即代理对象。
Weaving(织入)	它指的是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。
Aspect(切面)	它指定是增强的代码所关注的方面，把这些相关的增强代码定义到一个类中，这个类就是切面类。例如，事务切面，它里面定义的方法就是和事务相关的，像开启事务，提交事务，回滚事务等等，不会定义其他与事务无关的方法。我们前面的案例中 `TrasnactionManager` 就是一个切面。

连接点：方法开始时、结束时、正常运行完毕时、方法异常时等这些特殊的时机点，我们称之为连接点，项目中每个方法都有连接点，连接点是一种候选点

切入点：指定AOP思想想要影响的具体方法是哪些，描述感兴趣的方法

Advice增强：

第一个层次：指的是横切逻辑

第二个层次：方位点（在某一些连接点上加入横切逻辑，那么这些连接点就叫做方位点，描述的是具体的特殊时机）

Aspect切面：切面概念是对上述概念的一个综合

Aspect切面= 切入点+增强

= 切入点（锁定方法） + 方位点（锁定方法中的特殊时机）+ 横切逻辑

众多的概念，目的就是为了锁定要在哪个地方插入什么横切逻辑代码

第2节 Spring中AOP的代理选择

Spring 实现AOP思想使用的是动态代理技术

默认情况下，Spring会根据被代理对象是否实现接口来选择使用JDK还是CGLIB。当被代理对象没有实现任何接口时，Spring会选择CGLIB。当被代理对象实现了接口，Spring会选择JDK官方的代理技术，不过我们可以通过配置的方式，让Spring强制使用CGLIB。

第3节 Spring中AOP的配置方式

在Spring的AOP配置中，也和IoC配置一样，支持3类配置方式。

第一类：使用XML配置

第二类：使用XML+注解组合配置

第三类：使用纯注解配置

第4节 Spring中AOP实现

需求：横切逻辑代码是打印日志，希望把打印日志的逻辑织入到目标方法的特定位置(service层transfer方法)

4.1 XML 模式

Spring是模块化开发的框架，使用aop就引入aop的jar

- 坐标

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aop</artifactId>
    <version>5.1.12.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.9.4</version>
</dependency>
```

- AOP 核心配置

```
<!--
  Spring基于XML的AOP配置前期准备：
    在spring的配置文件中加入aop的约束
  xmlns:aop="http://www.springframework.org/schema/aop"
      http://www.springframework.org/schema/aop
https://www.springframework.org/schema/aop/spring-aop.xsd


  Spring基于XML的AOP配置步骤：
```

```
            第一步：把通知Bean交给Spring管理
            第二步：使用aop:config开始aop的配置
            第三步：使用aop:aspect配置切面
            第四步：使用对应的标签配置通知的类型
                    入门案例采用前置通知，标签为aop:before
    -->
    <!--把通知bean交给spring来管理-->
    <bean id="logUtil" class="com.lagou.utils.LogUtil"></bean>

    <!--开始aop的配置-->
    <aop:config>
     <!--配置切面-->
        <aop:aspect id="logAdvice" ref="logUtil">
      <!--配置前置通知-->
        <aop:before method="printLog" pointcut="execution(public *
com.lagou.service.impl.TransferServiceImpl.updateAccountByCardNo(com.lagou
.pojo.Account))"></aop:before>
        </aop:aspect>
    </aop:config>
```

- 细节
  - 关于切入点表达式

    上述配置实现了对 TransferServiceImpl 的 updateAccountByCardNo 方法进行增强，在其执行之前，输出了记录日志的语句。这里面，我们接触了一个比较陌生的名称：**切入点表达式**，它是做什么的呢？我们往下看。

    - 概念及作用

      切入点表达式，也称之为AspectJ切入点表达式，**指的是遵循特定语法结构的字符串，其作用是用于对符合语法格式的连接点进行增强。**它是AspectJ表达式的一部分。

    - 关于AspectJ

      AspectJ是一个基于Java语言的AOP框架，Spring框架从2.0版本之后集成了AspectJ框架中切入点表达式的部分，开始支持AspectJ切入点表达式。

    - 切入点表达式使用示例

```
全限定方法名    访问修饰符   返回值   包名.包名.包名.类名.方法名(参数列表)
    全匹配方式：
            public void
com.lagou.service.impl.TransferServiceImpl.updateAccountByCardNo(c
om.lagou.pojo.Account)
    访问修饰符可以省略
        void
com.lagou.service.impl.TransferServiceImpl.updateAccountByCardNo(c
om.lagou.pojo.Account)
    返回值可以使用*，表示任意返回值
        *
com.lagou.service.impl.TransferServiceImpl.updateAccountByCardNo(c
om.lagou.pojo.Account)
```

```
    包名可以使用.表示任意包，但是有几级包，必须写几个
        * ....TransferServiceImpl.updateAccountByCardNo(com.lagou.pojo.Account)
    包名可以使用..表示当前包及其子包
        * ..TransferServiceImpl.updateAccountByCardNo(com.lagou.pojo.Account)
        类名和方法名，都可以使用.表示任意类，任意方法
        * ...(com.lagou.pojo.Account)
        参数列表，可以使用具体类型
        基本类型直接写类型名称 ： int
        引用类型必须写全限定类名：java.lang.String
    参数列表可以使用*，表示任意参数类型，但是必须有参数
        * *..*.*(*)
        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
        * *..*.*(..)
        全通配方式：
        * *..*.*(..)
```

- 改变代理方式的配置

  在前面我们已经说了，Spring在选择创建代理对象时，会根据被代理对象的实际情况来选择的。被代理对象实现了接口，则采用基于接口的动态代理。当被代理对象没有实现任何接口的时候，Spring会自动切换到基于子类的动态代理方式。

  但是我们都知道，无论被代理对象是否实现接口，只要不是final修饰的类都可以采用cglib提供的方式创建代理对象。所以Spring也考虑到了这个情况，提供了配置的方式实现强制使用基于子类的动态代理（即cglib的方式），配置的方式有两种

  - 使用aop:config标签配置

    ```
    <aop:config proxy-target-class="true">
    ```

  - 使用aop:aspectj-autoproxy标签配置

    ```
    <!--此标签是基于XML和注解组合配置AOP时的必备标签，表示Spring开启注解配置AOP的支持-->
    <aop:aspectj-autoproxy proxy-target-class="true"></aop:aspectj-autoproxy>
    ```

- 五种通知类型

  - 前置通知

    配置方式：aop:before标签

```
<!--
  作用:
    用于配置前置通知。
  出现位置:
    它只能出现在aop:aspect标签内部
  属性:
    method:用于指定前置通知的方法名称
    pointcut:用于指定切入点表达式
    pointcut-ref:用于指定切入点表达式的引用
-->
<aop:before method="printLog" pointcut-ref="pointcut1">
</aop:before>
```

执行时机

前置通知永远都会在切入点方法（业务核心方法）执行之前执行。

细节

前置通知可以获取切入点方法的参数，并对其进行增强。

- 正常执行时通知

  配置方式

```
    <!--
      作用:
        用于配置正常执行时通知
      出现位置:
        它只能出现在aop:aspect标签内部
      属性:
        method:用于指定后置通知的方法名称
        pointcut:用于指定切入点表达式
        pointcut-ref:用于指定切入点表达式的引用
    -->
    <aop:after-returning method="afterReturningPrintLog" pointcut-
ref="pt1"></aop:after-returning>
```

  - 异常通知

  配置方式

```
    <!--
  作用:
        用于配置异常通知。
  出现位置:
        它只能出现在aop:aspect标签内部
  属性:
        method:用于指定异常通知的方法名称
    pointcut:用于指定切入点表达式
        pointcut-ref:用于指定切入点表达式的引用

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

执行时机

异常通知的执行时机是在切入点方法（业务核心方法）执行产生异常之后，异常通知执行。如果切入点方法执行没有产生异常，则异常通知不会执行。

细节

异常通知不仅可以获取切入点方法执行的参数，也可以获取切入点方法执行产生的异常信息。

- 最终通知

  配置方式

```
<!--
  作用:
  用于指定最终通知。
  出现位置:
  它只能出现在aop:aspect标签内部
  属性:
  method:用于指定最终通知的方法名称
    pointcut:用于指定切入点表达式
  pointcut-ref:用于指定切入点表达式的引用
-->
<aop:after method="afterPrintLog" pointcut-ref="pt1"></aop:after>
```

执行时机

最终通知的执行时机是在切入点方法（业务核心方法）执行完成之后，切入点方法返回之前执行。换句话说，无论切入点方法执行是否产生异常，它都会在返回之前执行。

细节

最终通知执行时，可以获取到通知方法的参数。同时它可以做一些清理操作。

- 环绕通知

  配置方式

```
    <!--
  作用:
        用于配置环绕通知。
  出现位置:
        它只能出现在aop:aspect标签的内部
  属性:
        method:用于指定环绕通知的方法名称
    pointcut:用于指定切入点表达式
        pointcut-ref:用于指定切入点表达式的引用

    -->
<aop:around method="aroundPrintLog" pointcut-ref="pt1"></aop:around>
```

```
**特别说明**

环绕通知，它是有别于前面四种通知类型外的特殊通知。前面四种通知（前置，后置，异常和最终）它们都是指定何时增强的通知类型。而环绕通知，它是Spring框架为我们提供的一种可以通过编码的方式，控制增强代码何时执行的通知类型。它里面借助的ProceedingJoinPoint接口及其实现类，实现手动触发切入点方法的调用。

**ProceedingJoinPoint接口介绍**

类视图:

![image-20191205141201938]
(Spring%E9%AB%98%E7%BA%A7%E6%A1%86%E6%9E%B6%E8%AF%BE%E7%A8%8B%E8%AE%B2%E4%B9%89.assets/image-20191205141201938.png)
```

4.2 XML+注解模式

- XML 中开启 Spring 对注解 AOP 的支持

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

- 示例

```
/**
 * 模拟记录日志
 * @author 应癫
 */
@Component
@Aspect
public class LogUtil {

  /**
    * 我们在xml中已经使用了通用切入点表达式，供多个切面使用，那么在注解中如何使用呢?
```

```
    * 第一步：编写一个方法
    * 第二步：在方法使用@Pointcut注解
    * 第三步：给注解的value属性提供切入点表达式
    * 细节：
    *   1.在引用切入点表达式时，必须是方法名+()，例如"pointcut()"。
    *   2.在当前切面中使用，可以直接写方法名。在其他切面中使用必须是全限定方法名。
    */
    @Pointcut("execution(* com.lagou.service.impl.*.*(..))")
    public void pointcut(){}


    @Before("pointcut()")
    public void beforePrintLog(JoinPoint jp){
        Object[] args = jp.getArgs();
        System.out.println("前置通知：beforePrintLog，参数是："+
Arrays.toString(args));
    }

    @AfterReturning(value = "pointcut()",returning = "rtValue")
    public void afterReturningPrintLog(Object rtValue){
        System.out.println("后置通知：afterReturningPrintLog，返回值
是："+rtValue);
    }

    @AfterThrowing(value = "pointcut()",throwing = "e")
    public void afterThrowingPrintLog(Throwable e){
        System.out.println("异常通知：afterThrowingPrintLog，异常是："+e);
    }

    @After("pointcut()")
    public void afterPrintLog(){
        System.out.println("最终通知：afterPrintLog");
    }

    /**
     * 环绕通知
     * @param pjp
     * @return
     */
    @Around("pointcut()")
    public Object aroundPrintLog(ProceedingJoinPoint pjp){
        //定义返回值
        Object rtValue = null;
        try{
            //前置通知
            System.out.println("前置通知");

            //1.获取参数
            Object[] args = pjp.getArgs();
```

```
            //2.执行切入点方法
            rtValue = pjp.proceed(args);

            //后置通知
            System.out.println("后置通知");
        }catch (Throwable t){
            //异常通知
            System.out.println("异常通知");
            t.printStackTrace();
        }finally {
            //最终通知
            System.out.println("最终通知");
        }
        return  rtValue;
    }

}
```

4.3 注解模式

在使用注解驱动开发aop时，我们要明确的就是，是注解替换掉配置文件中的下面这行配置：

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

在配置类中使用如下注解进行替换上述配置

```
/**
 * @author 应癫
 */
@Configuration
@ComponentScan("com.lagou")
@EnableAspectJAutoProxy //开启spring对注解AOP的支持
public class SpringConfiguration {
}
```

第5节 Spring 声明式事务的支持

编程式事务：在业务代码中添加事务控制代码，这样的事务控制机制就叫做编程式事务

声明式事务：通过xml或者注解配置的方式达到事务控制的目的，叫做声明式事务

5.1 事务回顾

5.1.1 事务的概念

事务指逻辑上的一组操作，组成这组操作的各个单元，要么全部成功，要么全部不成功。从而确保了数据的准确与安全。

例如：A——B转帐，对应于如下两条sql语句:

```
/*转出账户减钱*/
update account set money=money-100 where name='a';
/**转入账户加钱*/
update account set money=money+100 where name='b';
```

这两条语句的执行，要么全部成功，要么全部不成功。

5.1.2 事务的四大特性

原子性（Atomicity） 原子性是指事务是一个不可分割的工作单位，事务中的操作要么都发生，要么都不发生。

从操作的角度来描述，事务中的各个操作要么都成功要么都失败

一致性（Consistency） 事务必须使数据库从一个一致性状态变换到另外一个一致性状态。

例如转账前A有1000，B有1000。转账后A+B也得是2000。

一致性是从数据的角度来说的，（1000，1000）（900，1100），不应该出现（900，1000）

隔离性（Isolation） 事务的隔离性是多个用户并发访问数据库时，数据库为每一个用户开启的事务，

每个事务不能被其他事务的操作数据所干扰，多个并发事务之间要相互隔离。

比如：事务1给员工涨工资2000，但是事务1尚未被提交，员工发起事务2查询工资，发现工资涨了2000块钱，读到了事务1尚未提交的数据（脏读）

持久性（Durability）

持久性是指一个事务一旦被提交，它对数据库中数据的改变就是永久性的，接下来即使数据库发生故障也不应该对其有任何影响。

5.1.3 事务的隔离级别

不考虑隔离级别，会出现以下情况：（以下情况全是错误的），也即为隔离级别在解决事务并发问题

脏读：一个线程中的事务读到了另外一个线程中**未提交**的数据。

不可重复读：一个线程中的事务读到了另外一个线程中已经提交的**update**的数据（前后内容不一样）

场景：

员工A发起事务1，查询工资，工资为1w，此时事务1尚未关闭

财务人员发起了事务2，给员工A张了2000块钱，**并且提交了事务**

员工A通过事务1再次发起查询请求，发现工资为1.2w，原来读出来1w读不到了，叫做不可重复读

虚读（幻读）：一个线程中的事务读到了另外一个线程中已经提交的insert或者delete的数据（前后条数不一样）

场景：

事务1查询所有工资为1w的员工的总数，查询出来了10个人，此时事务尚未关闭

事务2财务人员发起，新来员工，工资1w，向表中插入了2条数据，**并且提交了事务**

事务1再次查询工资为1w的员工个数，发现有12个人，见了鬼了

数据库共定义了四种隔离级别：

Serializable（串行化）：可避免脏读、不可重复读、虚读情况的发生。（串行化） 最高

Repeatable read（可重复读）：可避免脏读、不可重复读情况的发生。(幻读有可能发生) 第二

该机制下会对要update的行进行加锁

Read committed（读已提交）：可避免脏读情况发生。不可重复读和幻读一定会发生。 第三

Read uncommitted（读未提交）：最低级别，以上情况均无法保证。(读未提交) 最低

注意：级别依次升高，效率依次降低

MySQL的默认隔离级别是：REPEATABLE READ

查询当前使用的隔离级别：`select @@tx_isolation;`

设置MySQL事务的隔离级别：`set session transaction isolation level xxx;`（设置的是当前mysql连接会话的，并不是永久改变的）

5.1.4 事务的传播行为

事务往往在service层进行控制，如果出现service层方法A调用了另外一个service层方法B，A和B方法本身都已经被添加了事务控制，那么A调用B的时候，就需要进行事务的一些协商，这就叫做事务的传播行为。

A调用B，我们站在B的角度来观察来定义事务的传播行为

PROPAGATION_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。这是最常见的选择。**
PROPAGATION_SUPPORTS	**支持当前事务，如果当前没有事务，就以非事务方式执行。**
PROPAGATION_MANDATORY	使用当前的事务，如果当前没有事务，就抛出异常。
PROPAGATION_REQUIRES_NEW	新建事务，如果当前存在事务，把当前事务挂起。
PROPAGATION_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。
PROPAGATION_NEVER	以非事务方式执行，如果当前存在事务，则抛出异常。
PROPAGATION_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，则执行与PROPAGATION_REQUIRED类似的操作。

5.2 Spring中事务的API

mybatis: sqlSession.commit();

hibernate: session.commit();

PlatformTransactionManager

```
public interface PlatformTransactionManager {

  /**
```

```
    * 获取事务状态信息
    */
   TransactionStatus getTransaction(@Nullable TransactionDefinition definition)
throws TransactionException;

   /**
    * 提交事务
    */
   void commit(TransactionStatus status) throws TransactionException;

   /**
    * 回滚事务
    */
   void rollback(TransactionStatus status) throws TransactionException;
}
```

作用

此接口是Spring的事务管理器核心接口。Spring本身并不支持事务实现，只是负责提供标准，应用底层支持什么样的事务，需要提供具体实现类。此处也是策略模式的具体应用。在Spring框架中，也为我们内置了一些具体策略，例如：DataSourceTransactionManager，HibernateTransactionManager等等。（和`HibernateTransactionManager`事务管理器在`spring-orm-5.1.12.RELEASE.jar`中）

Spring JdbcTemplate（数据库操作工具）、Mybatis（mybatis-spring.jar）————> DataSourceTransactionManager

Hibernate框架 ————————> HibernateTransactionManager

DataSourceTransactionManager 归根结底是横切逻辑代码，声明式事务要做的就是使用Aop（动态代理）来将事务控制逻辑织入到业务代码

5.3 Spring 声明式事务配置

- 纯xml模式
  - 导入jar

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.1.12.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.9.4</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>5.1.12.RELEASE</version>
```

```
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-tx</artifactId>
    <version>5.1.12.RELEASE</version>
</dependency>
```

- xml 配置

```
 <tx:advice id="txAdvice" transaction-manager="transactionManager">
        <!--定制事务细节，传播行为、隔离级别等-->
        <tx:attributes>
            <!--一般性配置-->
            <tx:method name="*" read-only="false"
propagation="REQUIRED" isolation="DEFAULT" timeout="-1"/>
            <!--针对查询的覆盖性配置-->
            <tx:method name="query*" read-only="true"
propagation="SUPPORTS"/>
        </tx:attributes>
    </tx:advice>

    <aop:config>
        <!--advice-ref指向增强=横切逻辑+方位-->
        <aop:advisor advice-ref="txAdvice" pointcut="execution(*
com.lagou.edu.service.impl.TransferServiceImpl.*(..))"/>
    </aop:config>
```

- 基于XML+注解
  - xml配置

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

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

  - 在接口、类或者方法上添加@Transactional注解

```
@Transactional(readOnly = true,propagation = Propagation.SUPPORTS)
```

- 基于纯注解