课程目标

- 理解 AOP 是什么，为什么使用 AOP，Spring 与它是什么关系。
- 理解为什么要 Spring 集成 MyBatis，集成本质是什么，利用 Spring 什么功能做什么。
- 掌握完成 Spring 集成 MyBatis。
- 了解 Spring 对事务的支持体验在哪里。
- 掌握利用 AOP 给业务方法添加事务。

一、AOP 思想和重要术语（理解）

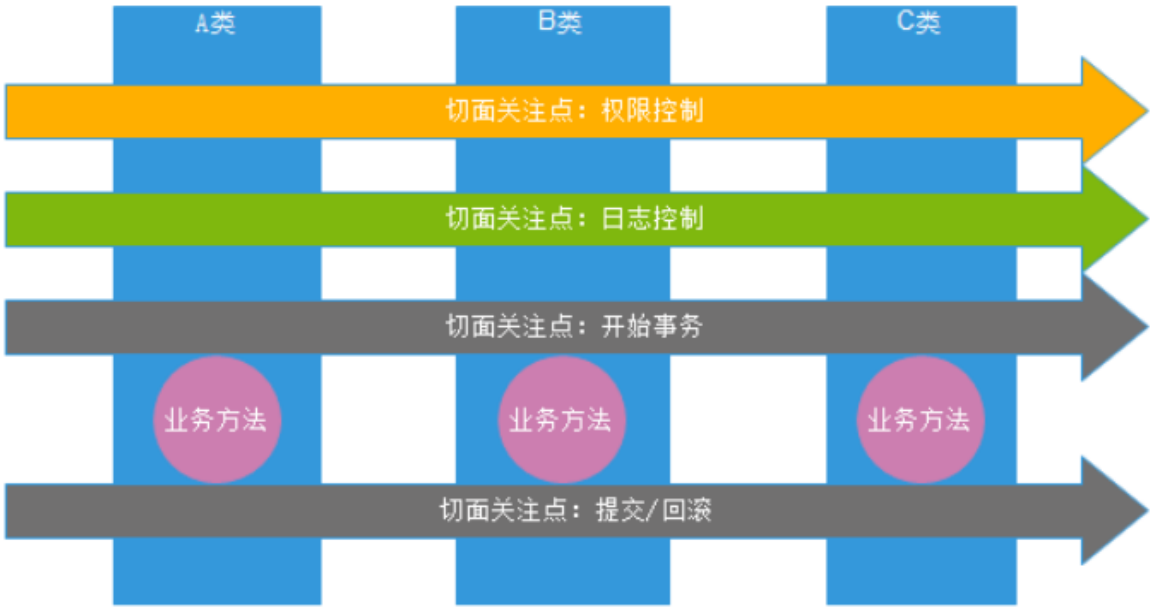
1、需求问题

在开发中，为了给业务方法中增加日志记录，权限检查，事务控制等功能，此时我们需要去修改业务方法代码，考虑到代码的重用性，我们可以考虑使用 OOP 的继承或组合关系来消除重复，但是无论怎么样，我们都会在业务方法中纵向地增加这些功能方法的调用代码。此时，既不遵循开闭原则，也会为后期系统的维护带来很大的麻烦。（即不管怎样都得修改到原来的代码）

为了解决该问题，OOP 思想是不行了，得使用 AOP 思想。

2、AOP

AOP（Aspect Oriented Programming），是面向切面编程的技术，把一个个的横切关注点放到某个模块中去，称之为切面。那么每一个的切面都能影响业务的某一种功能，切面的目的就是功能增强，如日志切面就是一个横切关注点，应用中许多方法需要做日志记录的只需要插入日志的切面即可。（动态代理就可以实现 AOP），这种面向切面编程的思想就是 AOP 思想了。



AOP 能够将那些与业务无关，却为业务模块所共同调用的逻辑或责任（例如事务处理、日志管理、权限控制等）封装起来，便于减少系统的重复代码，降低模块间的耦合度，并有利于未来的可拓展性和可维护性。说人话：把业务方法中与业务无关的操作抽离到不同的对象中，最后使用动态代理的方式组合起来，动态地为类增加功能。

3、AOP 术语

- Joinpoint：连接点，一般指需要被增强的方法。where：去哪里做增强。
- Pointcut：切入点，哪些包中的哪些类中的哪些方法，可认为是连接点的集合。where：去哪些地方做增强。
- Advice：增强，当拦截到 Joinpoint 之后，在方法执行的什么时机（when）做什么样（what）的增强。根据时机分为：前置增强、后置增强、异常增强、最终增强、环绕增强。
- Aspect：切面，Pointcut + Advice，去哪些地方 + 在什么时候 + 做什么增强。
- Target：被代理的目标对象。
- Weaving：织入，把 Advice 加到 Target 上之后，创建出 Proxy 对象的过程。
- Proxy：一个类被 AOP 织入增强后，产生的代理类。

二、AOP 实现及 Pointcut 表达式（了解）

1、AOP 规范及实现

AOP 的规范本应该由 SUN 公司提出，是被 AOP 联盟捷足先登。AOP 联盟制定 AOP 规范。

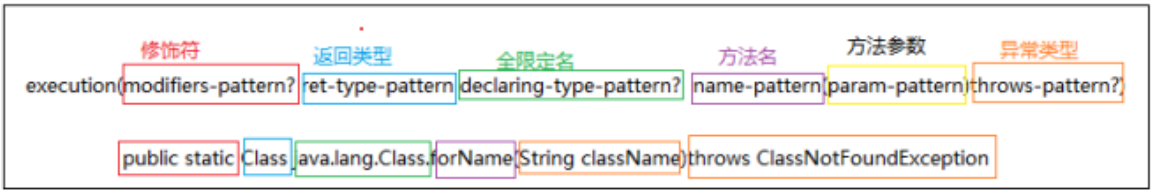
既然 AOP 是一个思想，就类似 IoC 思想，都是一个理论，那谁来实现这个 AOP 思想？AOP 思想的实现有 Spring AOP 和 AspectJ，都是开源的。那么如何选用呢，既然用 Spring 肯定要选用 Spring AOP，因为能与 Spring 无缝整合。那么为什么要使用提 AspectJ 呢? 那是因为用原始 Spring AOP 编程很麻烦，之后 Spring AOP 有些东西借鉴了 AspectJ（比如切入点表达式），简化 AOP 的配置与开发。

2、AspectJ

AspectJ 是一个面向切面的框架，它扩展了Java 语言（即使用 Java 对 AOP 进行了实现）。

3、AspectJ 切入点语法（掌握）

怎么表示 Pointcut，即怎么表示哪些包中的哪些类中的哪些方法？AspectJ 提供了表示切入点的语法。



3.1、切入点语法通配符

- *：匹配任何部分，只能表示一个单词。
- ..：可用于全限定名中和方法参数中，分别表示子包和 0 到 N 个参数。

3.2、切入点语法案例

注意第一个星符号后面有空格。

```
execution(* cn.wolfcode.ssm.service.impl.*ServiceImpl.*(..))
```

三、使用 XML 配置 AOP（掌握）

1、需求

想给业务方法加模拟的事务功能，又不想修改原来的代码

2、使用 AOP

2.1、拷贝之前动态代理的项目

修改项目名为 spring-xml-aop 再导入项目，删除其中动态代理有关的代码。

2.2、添加依赖

```
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.8.13</version>
</dependency>
```

2.3、编写 AOP 配置

在 applicationContext.xml，配置如下：

```
<!-- 配置真实对象 -->
<bean id="employeeService"
class="cn.wolfcode.service.impl.EmployeeServiceImpl"/>

<!-- 配置是事务管理器， WHAT -->
<bean id="transactionManager" class="cn.wolfcode.tx.MyTransactionManager"/>

<!-- 配置 AOP -->
<aop:config>
    <aop:aspect ref="transactionManager">
        <!-- 定义切入点 WHERE -->
```

```
        <aop:pointcut expression="execution(*
cn.wolfcode.service.impl.*ServiceImpl.*(..))"
            id="txPointcut"/>
        <!-- 上面切入点切到方法，在方法执行之前，加  transactionManager.begin() -->
        <aop:before pointcut-ref="txPointcut" method="begin"/>
        <!-- 上面切入点切到方法，在方法正常执行完，加  transactionManager.commit() -->
        <aop:after-returning pointcut-ref="txPointcut" method="commit"/>
        <!-- 上面切入点切到方法，在方法执行抛出异常时，加
transactionManager.rollback() -->
        <aop:after-throwing pointcut-ref="txPointcut" method="rollback"/>
    </aop:aspect>
</aop:config>
```

2.4、修改单元测试类

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
public class EmployeeServiceTest {
    @Autowired
    private IEmployeeService employeeService;

    @Test
    public void testSave() {
        System.out.println(employeeService.getClass());
        employeeService.save("罗老师", "666");
    }
}
```

3、变更使用 CGLIB

Spring AOP 不做配置的话且目标对象实现接口的话，默认使用 JDK 的动态代理。若想强制使用 CGLIB，则在 applicationContext.xml 配置如下：

```
<aop:config proxy-target-class="true">
    <!-- 省略 ..... -->
</aop:config>
```

四、使用注解配置 AOP（掌握）

1、拷贝之前使用 XML 配置 AOP 的项目

修改项目名为 spring-anno-aop 再导入项目，删除其中配置 AOP 的 XML 代码。

2、对比 XML 配置来使用注解的方式来配置

```
package cn.wolfcode.tx;

@Component
@Aspect
public class MyTransactionManager {

    // 定义切入点 WHERE
```

```
    @Pointcut("execution(* cn.wolfcode.service.impl.*ServiceImpl.*(..))")
    public void txPointcut() {}

    @Before("txPointcut()")
    public void begin() {
        System.out.println("开启事务");
    }

    @AfterReturning("txPointcut()")
    public void commit() {
        System.out.println("提交事务");
    }

    @AfterThrowing("txPointcut()")
    public void rollback() {
        System.out.println("回滚事务");
    }
}
```

3、修改配置文件

在 applicationContext.xml，配置如下：

```
<!-- 配置真实对象 -->
<!-- 配置是管理器，WHAT -->
<context:component-scan base-package="cn.wolfcode"/>

<!-- AOP 注解解析器，让这些 AOP 注解起作用 -->
<aop:aspectj-autoproxy/>
```

4、变更使用 CGLIB

在 applicationContext.xml，配置如下

```
<aop:aspectj-autoproxy proxy-target-class="true"/>
```

五、集成准备（掌握）

1、集成作用及本质

- 使用框架，在别人的基础上开发，提高效率；
- 集成 MyBatis 和 业务层，即业务对象、Mapper 对象等都交由 Spring 容器管理：
  - 使用 Spring IoC 和 DI 来完成对象创建及其属性注入。
  - 后面再使用AOP来配置事务。

2、项目搭建

新建 Maven 项目，**打包方式是 war**，为后面项目做铺垫。**需求：做个转账功能。**

2.1、修改项目打包方式

在 pom.xml，追加一段配置，配置如下：

```
<packaging>war</packaging>
```

2.2、编写 web.xml

手动在项目的 main 目录下新建 webapp/WEB-INF/web.xml，配置如下：

```
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
    version="3.0">
</web-app>
```

2.3、设置编译版本及添加依赖和插件

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
</properties>

<dependencies>
    <!-- 数据库驱动 -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.45</version>
        <scope>runtime</scope>
    </dependency>
    <!-- 数据库连接池 -->
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>druid</artifactId>
        <version>1.1.9</version>
    </dependency>

    <!-- MyBatis 相关 -->
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.4.5</version>
    </dependency>
    <!-- Spring 集成 MyBatis 的依赖 -->
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis-spring</artifactId>
        <version>1.3.1</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-log4j12</artifactId>
        <version>1.7.25</version>
    </dependency>
```

```
    <!-- Spring 相关 -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
        <version>5.0.8.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.aspectj</groupId>
        <artifactId>aspectjweaver</artifactId>
        <version>1.8.13</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>5.0.8.RELEASE</version>
    </dependency>

    <!-- 测试相关 -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-test</artifactId>
        <version>5.0.8.RELEASE</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.12</version>
        <scope>test</scope>
    </dependency>

    <!-- web 项目共用 -->
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.16.22</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>3.0.1</version>
        <scope>provided</scope>
    </dependency>

    <!-- 页面标签 -->
    <dependency>
        <groupId>org.apache.taglibs</groupId>
        <artifactId>taglibs-standard-spec</artifactId>
        <version>1.2.5</version>
    </dependency>
    <dependency>
        <groupId>org.apache.taglibs</groupId>
        <artifactId>taglibs-standard-impl</artifactId>
        <version>1.2.5</version>
    </dependency>
</dependencies>
```

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.tomcat.maven</groupId>
            <artifactId>tomcat7-maven-plugin</artifactId>
            <version>2.1</version>
            <configuration>
                <port>80</port> <!-- 端口 -->
                <path>/</path> <!-- 上下路径 -->
                <uriEncoding>UTF-8</uriEncoding> <!-- 针对 GET 方式乱码处理 -->
            </configuration>
        </plugin>
    </plugins>
</build>
```

2.4、准备数据库

在数据库新建一个名为 ssm 的库，建如下的表：

```
CREATE TABLE `account` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `balance` decimal(10,2) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

2.5、编写实体类

```
package cn.wolfcode.domain;

@Setter
@Getter
public class Account {
    private Long id;
    private BigDecimal balance;
}
```

六、配置 SqlSessionFactory（掌握）

1、使用 MyBatis 框架的问题

```
public void testOld() throws Exception {
    SqlSessionFactory ssf = new
SqlSessionFactoryBuilder().build(Resources.getResourceAsStream("mybatis-
config.xml"));
    SqlSession session = ssf.openSession();
    AccountMapper accountMapper = session.getMapper(AccountMapper.class);
    // ......
    session.close();
}
```

发现要创建 SqlSessionFactory 对象，而且应用中只需要一个，而创建对象这个工作交由 Spring 创建并管理多好。

2、拷贝配置文件

从给的课件中 practise/config，拷贝 db.properties、log4j.properties 和 mybatis-config.xml 到项目的 resources 目录下。

3、配置 SqlSessionFactory

查看 SqlSessionFactoryBean，其实现 FactoryBean 接口，通过配置在 Spring 配置文件中，会往容器里面创建和存放 SqlSessionFactory 类型对象。配置的时候想想配置些什么？且对比之前 MyBatis 主配置文件。

在 resources 目录下新建 applicationContext.xml，配置如下：

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

<!-- 配置 DataSource -->
<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"
    init-method="init" destroy-method="close">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>

<!-- 配置 SqlSessionFactory -->
<bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
    <!-- 数据源 -->
    <property name="dataSource" ref="dataSource"/>
    <!-- 配置别名，若在 Mapper XML 文件中不想使用别名也可以不用配置 -->
    <property name="typeAliasesPackage" value="cn.wolfcode.domain"/>
    <!-- 配管关联 MyBatis 主配置文件 可以不配置，后期使用 mybatis 插件的时候需要配置
        <property name="configLocation" value="classpath:mybatis-config.xml">
</property>
    -->
    <!-- 配管关联 MyBatis Mapper XML 文件 ，因为两者编译之后在一起
        <property name="mapperLocations"
value="classpath:cn/wolfcode/mapper/*Mapper.xml"></property>
    -->
</bean>
```

七、配置 Mapper 对象（掌握）

1、编写 AccountMapper 接口和 AccountMapper.xml

对应提供加钱和减钱的方法及对应 SQL。

```
package cn.wolfcode.mapper;

public interface AccountMapper {
    // 加钱
    void addBalance(@Param("inId")Long inId, @Param("amount")BigDecimal amount);
    // 减钱
    void subtractBalance(@Param("outId")Long outId, @Param("amount")BigDecimal
amount);
}
```

在 resources 目录下新建 cn/wolfcode/mapper/AccountMapper.xml，文件内容如下：

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
  PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
  "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="cn.wolfcode.mapper.AccountMapper">
    <update id="addBalance">
        UPDATE account SET balance = balance + #{amount} WHERE id = #{inId}
    </update>
    <update id="subtractBalance">
        UPDATE account SET balance = balance - #{amount} WHERE id = #{outId}
    </update>
</mapper>
```

2、使用 MyBatis 框架的问题

```
public void testOld() throws Exception {
    SqlSessionFactory ssf = new
SqlSessionFactoryBuilder().build(Resources.getResourceAsStream("mybatis-
config.xml"));
    SqlSession session = ssf.openSession();
    AccountMapper accountMapper = session.getMapper(AccountMapper.class);
    // ......
    session.close();
}
```

获取 Mapper 接口的代理对象之前是由 MyBatis 管理，那么带来的问题，若想在业务对象中注入 Mappe r接口的代理对象就，必须改成交由 Spring 容器统一管理。

3、配置 AccountMapper 对象（了解）

对比以前获取 AccountMapper 接口代理对象的代码思考配置。在 applicationContext.xml，配置如下：

```
<!-- 配置 AccountMapper -->
<bean id="accountMapper" class="org.mybatis.spring.mapper.MapperFactoryBean">
    <!-- 借由容器中的 sqlSessionFactory 来创建 -->
    <property name="sqlSessionFactory" ref="sqlSessionFactory"/>
    <!-- 具体创建什么接口类型的代理对象 -->
    <property name="mapperInterface" value="cn.wolfcode.mapper.AccountMapper"/>
</bean>
```

八、配置业务层对象（掌握）

1、编写业务接口及其实现类

```
package cn.wolfcode.service;

public interface IAccountService {
    // 转账方法
    void transfer(Long outId, Long inId, BigDecimal amount);
}
```

```
package cn.wolfcode.service.impl;

public class AccountServiceImpl implements IAccountService {
    private AccountMapper accountMapper;
    public void setAccountMapper(AccountMapper accountMapper) {
        this.accountMapper = accountMapper;
    }

    @Override
    public void transfer(Long outId, Long inId, BigDecimal amount) {
        accountMapper.subtractBalance(outId, amount);
        accountMapper.addBalance(inId, amount);
    }
}
```

2、配置业务对象

在 applicationContext.xml，配置如下：

```
<bean id="accountService" class="cn.wolfcode.service.impl.AccountServiceImpl">
    <property name="accountMapper" ref="accountMapper"/>
</bean>
```

3、编写单元测试类

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
public class AccountServiceTest {
    @Autowired
    private IAccountService accountService;

    @Test
    public void testTransfer() {
        accountService.transfer(1L, 2L, BigDecimal.TEN);
    }
}
```

九、简化配置（掌握）

1、配置 Mapper 接口扫描器

之前配置多个 Mapper 接口代理对象是比较麻烦的，使用 Mapper 接口扫描器可以批量生成 Mapper 接口代理对象并注册到 Spring 容器中。

在 applicationContext.xml，删除原有单个配置 Mapper 接口代理对象的的代码，改配置如下：

```
<!-- 批量创建 Mapper 接口实现对象 -->
<bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
    <!-- 指定 Mapper 接口所在的包路径 -->
    <property name="basePackage" value="cn.wolfcode.mapper"/>
</bean>
```

2、使用注解方式配置业务对象

在业务类上贴 IoC 注解和 DI 注解。

```
package cn.wolfcode.service.impl;

@Service
public class AccountServiceImpl implements IAccountService {
    @Autowired
    private AccountMapper accountMapper;

    @Override
    public void transfer(Long outId, Long inId, BigDecimal amount) {
        accountMapper.subtractBalance(outId, amount);
        accountMapper.addBalance(inId, amount);
    }
}
```

3、配置第三方解析程序

在 applicationContext.xml，配置如下：

```
<!--
    IoC DI 注解解析器
    让 Spring 帮你创建所有业务对象，给业务对象字段设置值
-->
<context:component-scan base-package="cn.wolfcode.service.impl"/>
```

十、银行转账案例分析（了解）

1、模拟代码运行时出了问题

在 AccountServiceImpl 中转账的方法，中间插入造成异常（模拟）的代码。再运行单元测试方法，发现转出人少了钱，但转入人没有收到钱。

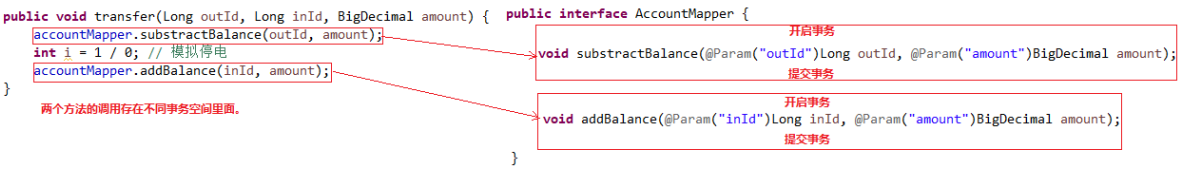
```
package cn.wolfcode.service.impl;

@Service
```

```
public class AccountServiceImpl implements IAccountService {
    @Autowired
    private AccountMapper accountMapper;

    @Override
    public void transfer(Long outId, Long inId, BigDecimal amount) {
        accountMapper.subtractBalance(outId, amount);
        int i = 1/0;
        accountMapper.addBalance(inId, amount);
    }
}
```

2、出现问题的原因



3、解决转账的事务问题

所以只要把转入和转出放在同一个事务就可以解决，解决思路（AOP）如下：

```
取消事务自动提交
获取连接对象
try{
    public void transfer(Long outId, Long inId, BigDecimal amount) {
        accountMapper.substractBalance(outId, amount);
        int i = 1 / 0; // 模拟停电
        accountMapper.addBalance(inId, amount);
    }
    提交事务
}catch(Exception e){
回滚事务
}
```

十一、Spring 对事务支持（了解）

1、Spring 对象事务支持的 API

PlatformTransactionManager：统一抽象处理事务操作相关的方法，是其他事务管理器的规范，根据实际情况选用不同的实现类。

- TransactionStatus getTransaction(TransactionDefinition definition)：根据事务定义信息从事务环境中返回一个已存在的事务，或者创建一个新的事务。
- void commit(TransactionStatus status)：根据事务的状态提交事务，如果事务状态已经标识为 rollback-only，该方法执行回滚事务的操作。
- void rollback(TransactionStatus status)：将事务回滚，当 commit 方法抛出异常时，rollback 会被隐式调用。

使用 Spring 管理事务的时候，针对不同的持久化技术选用不同的事务管理器：

- **JDBC 和 MyBatis 使用 DataSourceTransactionManager。**
- Hibernate 使用 HibernateTransactionManager。
- JPA 使用 JpaTransactionManager。

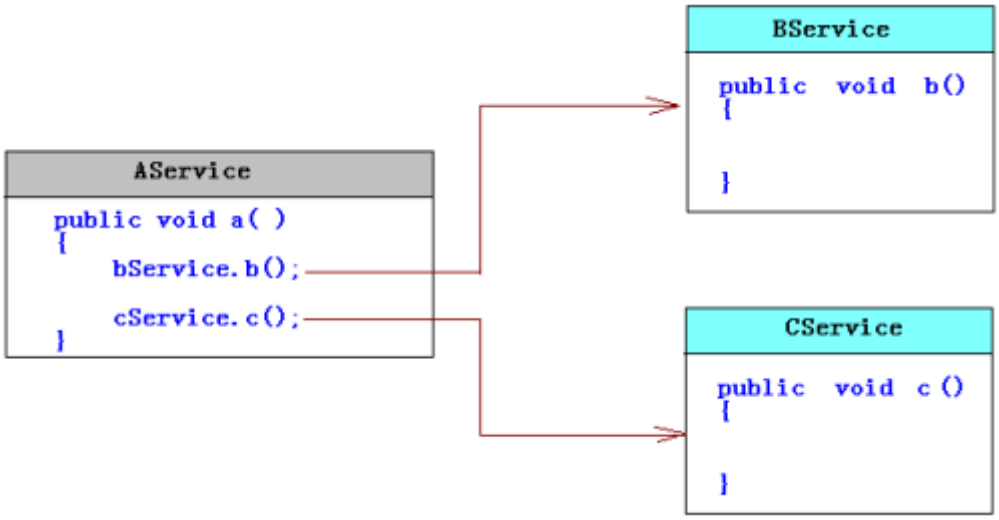
2、Spring实现事务的方式

- 编程式事务：通过编写代码来管理事务；
- 声明式事务：通过XML配置或注解来管理事务。

十二、事务的传播规则（了解）

1、事务传播规则

在一个事务方法中调用其他事务方法，此时事务该如何传播，按照什么规则传播，用谁的事务，还是都不用等等？



2、Spring共支持7种传播规则

- 情况一：遵从当前事务
  - **REQUIRED**：必须存在事务，如果当前存在一个事务，则加入该事务，否则将新建一个事务（缺省）。
  - **SUPPORTS**：支持当前事务，指如果当前存在事务，就加入到该事务，如果当前没有事务，就以非事务方式执行。
  - MANDATORY：必须有事务，使用当前事务执行，如果当前没有事务，则抛出异常IllegalTransactionStateException。
- 情况二：不遵从当前事务
  - **REQUIRES_NEW**：不管当前是否存在事务，每次都创建新的事务
  - NOT_SUPPORTRD：以非事务方式执行，如果当前存在事务，就把当前事务暂停，以非事务方式执行
  - NEVER：不支持事务，如果当前存在是事务，则抛出异常，IllegalTransactionStateException
- 情况三：寄生事务（外部事务和寄生事务）
  - NESTED ：寄生事务，如果当前存在事务，则在内部事务内执行，如果当前不存在事务，则创建一个新的事务，嵌套事务使用数据库中的保存点来实现，即嵌套事务回滚不影响外部事务，但外部事务回滚将导致嵌套事务回滚。

- 寄生事务：DataSourceTransactionManager默认支持，而 HibernateTransactionManager 默认不支持，需要手动开启。

十三、使用 XML 配置事务（了解）

1、编写配置

在 applicationContext.xml，配置如下：

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

<!-- 配置增强 WHEN，加关联上面 WHAT -->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <tx:attributes>
        <!-- 这里可以针对不同方法进行差异化配置 -->
        <tx:method name="transfer"/>
    </tx:attributes>
</tx:advice>

<aop:config>
    <!-- 配置切入点 WHERE -->
    <aop:pointcut expression="execution(*
cn.wolfcode.service.impl.*ServiceImpl.*(..))" id="txPoint"/>
    <!-- 关联上面的增强 -->
    <aop:advisor advice-ref="txAdvice" pointcut-ref="txPoint"/>
</aop:config>
```

2、测试

再运行单元测试看下效果。

十四、事务增强的属性配置（掌握）

1、事务方法属性配置

属性	必须	缺省值	描述
name	√		事务管理的方法，可使用通配符*
propagation	×	REQUIRED	事务传播规则，比如：SUPPORTS、REQUIRES_NEW
isolation	×	DEFAULT	事务隔离级别，DEFAULT:Spring 使用数据库默认的事务隔离级别、其他级别是 Spring 模拟的
read-only	×	false	事务是否只读，查询操作设置为只读事务
timeout	×	-1	事务超时时间（秒），若为-1，则有抵触事务系统决定
rollback-for	×	运行期异常	需要回滚的异常类型，多个使用逗号隔开
no-rollback-for	×	检查类型异常	不需要回滚的异常

- name：匹配到的方法，必须配置；可以使用通配符，*比如 get* 匹配所有 get 开头的方法。
- propagation：事务的传播规则。
- isolation：代表数据库事务隔离级别（就使用默认）
  - DEFAULT：让 Spring 使用数据库默认的事务隔离级别；
  - 其他：Spring 模拟，一般不用。
- timeout：缺省 -1，表示使用数据库底层的超时时间。
- read-only：若为 true，开启一个只读事务，只读事务的性能较高，但是不能在只读事务中操作 DML。
- no-rollback-for：如果遇到的异常是匹配的异常类型，就不回滚事务。
- rollback-for：如果遇到的异常是指定匹配的异常类型，才回滚事务。

Spring 默认情况下，业务方法抛出 RuntimeException 及其子类的异常才会去回滚，而抛出 Exception 和 Thowable 的异常不会回滚（自定义的业务异常会去继承 RuntimeException）。

2、通用事务配置

在 applicationContext.xml，增加对业务方法的差异配置，配置如下：

```
<tx:attributes>
    <tx:method name="get*" read-only="true"/>
    <tx:method name="list*" read-only="true"/>
    <tx:method name="query*" read-only="true"/>
    <tx:method name="count*" read-only="true"/>
    <tx:method name="*"/>
</tx:attributes>
```

- 其它方法的配置没有匹配到，就会使用最后一个通配方法配置匹配。
- 查询的方法名其实相对有限的，而其业务的方法则无数，所以才像上面那么配置。
- **方法名约定俗成，所以自己事务配置好了，业务方法名切记不要乱取了。**

十五、使用注解配置事务（掌握）

使用 XML 配置事务，配置过于繁琐。

1、为业务类贴注解

```java
package cn.wolfcode.service.impl;

@Service
// 贴事务注解
@Transactional
public class AccountServiceImpl implements IAccountService {
    @Autowired
    private AccountMapper accountMapper;

    @Override
    public void transfer(Long outId, Long inId, BigDecimal amount) {
        accountMapper.subtractBalance(outId, amount);
        int i = 1/0;
        accountMapper.addBalance(inId, amount);
    }
}
```

2、编写配置

在 applicationContext.xml 删除之前 XML 配置事务的代码，只保留配置事务管理器，配置具体如下：

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

<!-- 事务注解解析器 -->
<tx:annotation-driven transaction-manager="transactionManager"/>
```

3、测试

再运行单元测试看下效果。

4、Transactional 注解说明

Transactional 注解可以贴多个地方：

- 贴业务类或业务接口上，事务的配置是通用与整个类或接口的的方法；
- 贴业务方法上，即方法上的的事务的配置仅限于被贴的方法；
- 同时存在时，后者覆盖前者。

若想强制使用 CGLIB 动态代理，则修改事务注解解析器的配置，如下：

```xml
<tx:annotation-driven proxy-target-class="true" transaction-
manager="transactionManager"/>
```

练习

- 练习一，给所有业务方法执行之前加一个功能，打印当前的系统时间。

```
CREATE TABLE `account` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `balance` decimal(10,2) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

- 练习二，根据以上表结构和数据，提供转账功能，只需编写业务层和持久层的代码，持久层使用 MyBatis，这些层的对象都 Spring 管理，利用 AOP 给转账业务方法添加事务，使用 Spring Test 对转账业务方法进行测试。