Neuedu

东软睿道内部公开

文件编号： D000-

Spring框架技术

版本：3.6.0-0.0.0

第3章 AOP与事务处理

本章教学内容

Neuedu

节	知识点	掌握程度	难易程度	教学形式	对应微课
AOP基础知识	AOP是什么	理解	普通	线下	AOP是什么
	AOP应用范围	理解	普通	线下	AOP应用范围
	AOP术语	理解	普通	线下	AOP术语
	AOP开源框架介绍	了解	普通	线下	AOP开源框架介绍
	Spring对AOP的支持	掌握	普通	线下	Spring对AOP的支持
	Spring AOP代理机制	掌握	普通	线下	Spring AOP代理机制
	JDK动态代理	掌握	普通	线下	JDK动态代理
	CGlib动态代理	掌握	普通	线下	CGlib动态代理
Spring AOP编程	注解AOP编程流程	掌握	普通	线下	注解AOP编程流程
	execution表达式	掌握	普通	线下	execution表达式
	通知（advice）类型	掌握	普通	线下	通知（advice）类型
	XML配置AOP	掌握	普通	线下	XML配置AOP
Spring DAO支持	Spring对DAO的支持	掌握	普通	线下	Spring对DAO的支持
	Spring配置数据源	掌握	普通	线下	Spring配置数据源
	Spring JDBC概述	掌握	普通	线下	Spring JDBC概述
	JdbcTemplate	掌握	普通	线下	JdbcTemplate
	SpringJDBC与传统JDBC编程对比	掌握	普通	线下	SpringJDBC与传统JDBC编程对比
	JdbcTemplate主要回调接口	掌握	普通	线下	JdbcTemplate主要回调接口
	JdbcTemplate添加数据	掌握	普通	线下	JdbcTemplate添加数据
	JdbcTemplate查询数据	掌握	普通	线下	JdbcTemplate查询数据
Spring 事务处理	事务基础知识	掌握	普通	线下	事务基础知识
	事务并发问题	掌握	普通	线下	事务并发问题
	Spring对事务管理的支持	掌握	普通	线下	Spring对事务管理的支持
	Spring事务传播特性	掌握	普通	线下	Spring事务传播特性
	编程式事务	掌握	普通	线下	编程式事务
	XML配置声明式事务	掌握	普通	线下	XML配置声明式事务
	注解配置声明式事务	掌握	普通	线下	注解配置声明式事务

本章教学目标

- 理解AOP的概念、术语、基本原理；
- 理解JDK动态代理与CGLIB动态代理实现机制；
- 掌握使用注解、XML进行Spring AOP编程；
- 掌握JdbcTemplate对数据库访问操作；
- 掌握注解、XML进行Spring声明式事务配置；

CONTENTS

目录

本章内容

Neuedu

节	知识点	掌握程度	难易程度
AOP简介	AOP是什么	理解	难
	比较使用AOP和不使用AOP	理解	
	为何使用AOP	理解	
	AOP应用范围	了解	
核心概念	核心概念	理解	难
代理机制	静态代理	了解	
	动态代理	掌握	难
	基于注解方式的AOP编程	掌握	难
	基于配置方式的AOP编程	掌握	难

AOP是什么

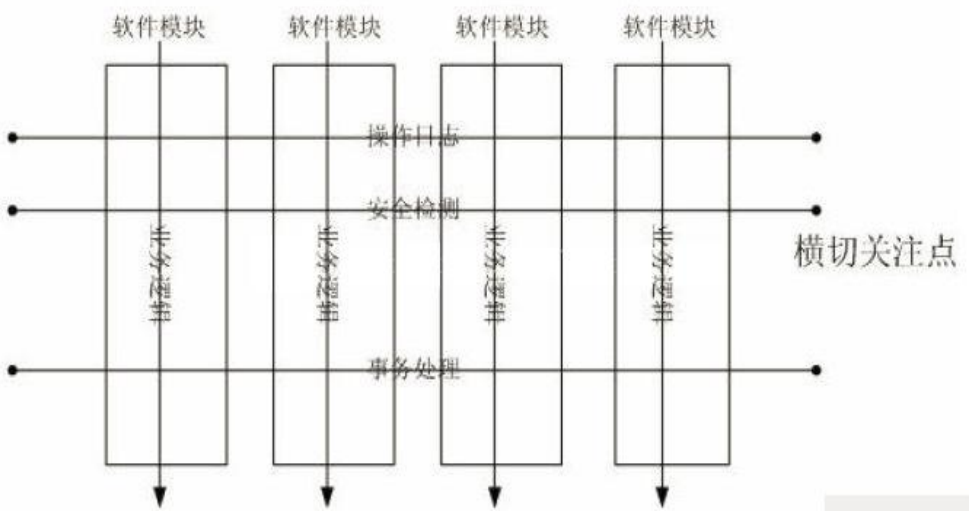
Neuedu

- AOP为Aspect Oriented Programming的缩写，意为：面向切面编程，类似的还有面向过程编程（Procedure-Oriented Programming），面向对象编程（Object-Oriented Programming），AOP的出现并不是要完全替代OOP，仅作为OOP的有益补充，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。
- AOP有特定的应用场合，它只适合具有横切逻辑的应用场合，如性能检测、访问控制、事务控制、日志记录等。就像面向对象编程，关注点在于对象，即类。而面向切面编程理所当然关注于切面，那么什么是切面？可以理解为程序执行时的某个节点，或更具体一点，在某个方法执行之前，执行之后，返回之后等其它节点。

- 比如我们往冰箱里放东西这个过程，非AOP的过程：开门-放冰箱里（大象）-关门，开门-放冰箱里（冰棍）-关门，应用AOP后，配置在调用放冰箱里前执行开门，在调用放冰箱里后执行关门，那么对于我们来说整个过程其实就是放冰箱里（大象），放冰箱里（冰棍），而开关门AOP帮你做了，这就是AOP的作用。

AOP是什么

- 按照软件重构思想的理念，在OOP中通过抽象把相同的代码抽取到父类中（纵向抽取），但无法通过抽象父类的方式消除重复性的横切代码（如事务处理这种非业务性的重复代码），而AOP就是为了解决将分散在各个业务逻辑代码中的相同代码通过横向切割的方式抽取到一个独立的模块中




```
public class ForumService {
    private TransactionManager transManager;
    private PerformanceMonitor pmonitor;
    private TopicDao topicDao;
    private ForumDao forumDao;

    public void removeTopic(int topicId) {
        pmonitor.start();
        transManager.beginTransaction();
        topicDao.removeTopic(topicId); //①
        transManager.commit();
        pmonitor.end();
    }
    public void createForum(Forum forum) {
        pmonitor.start();
        transManager.beginTransaction();
        forumDao.create(forum); //②
        transManager.commit();
        pmonitor.end();
    }
    ...
}
```

AOP应用范围

Neuedu

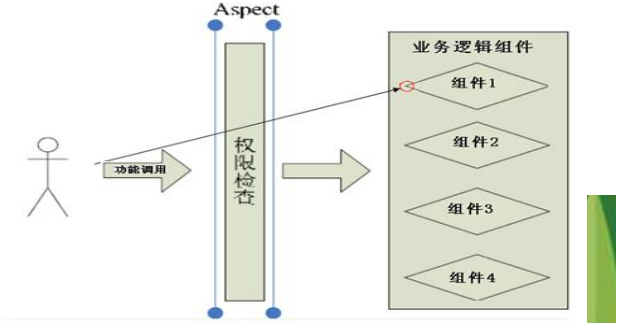
- Persistence（持久化）
- Transaction management（事务管理）
- Security（安全）
- Logging, tracing, profiling and monitoring（日志，跟踪，优化，监控）

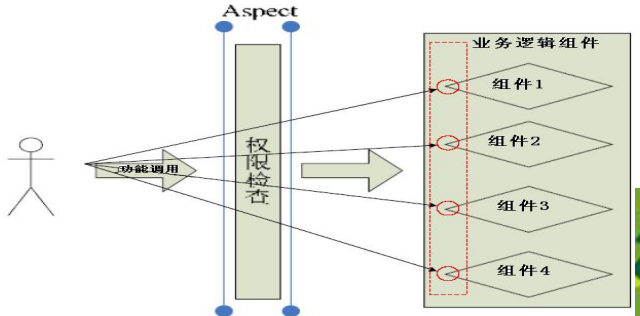
- Debugging（调试）
- Authentication（认证）
- Context passing（上下文传递）
- Error/Exception handling（错误/异常处理）

- Lazy loading（懒加载）
- Performance optimization（性能优化）
- Resource pooling（资源池）
- Synchronization（同步）

AOP术语

- 切面（Aspect）
  - 切面是切点和通知组成，通知和切点共同定义了切面的全部内容即：它是什么，在何时何处完成其功能；
- 连接点（Joinpoint）
  - 连接点是在应用执行过程中能够插入切面的一个点，Spring仅支持方法的连接点，即仅能在方法调用前，方法调用后，方法抛出异常时及方法调用前后插入切面代码。
- 切点(Pointcut）
  - 切点定义了在何处应用切面，AOP通过“切点”定位特定的连接点。切点相当于查询条件，一个切点可以匹配多个连接点。
- 通知（Advice）
  - 切面的工作被成为通知，定义了切面是什么及何时使用。除了描述切面要完成的工作，通知还解决了何时执行这个工作的问题，它应该在某个方法被调用之前？之后？等。
  - Spring切面可以应用5种类型的通知：
    - 前置通知（Before）在目标方法被调用之前调用通知功能；
    - 后置通知（After）在目标方法被完成之后调用通知功能，不关心方法的输出是什么；
    - 环绕通知（Around advice）通知包裹了目标方法，在目标方法调用之前和之后执行自定义的行为；
    - 异常通知（After-throwing)在目标方法抛出异常后调用通知；
    - 返回通知（After-returning）在目标方法成功执行之后调用通知；





AOP术语

- 目标对象(Target)
  - 通知逻辑的织入目标类。如果没有AOP，那么目标业务类需要自己实现所有的逻辑，在AOP的帮助下，目标类只需要实现那些非横切逻辑的程序逻辑，而比如事务管理等这些横切逻辑就可以使用AOP动态织入特定的连接点上。
- 引入（Introduction）
  - 引介是一种特殊的通知，为类添加一些属性和方法。这样，即使一个业务类原本没有实现某个接口，通过AOP的引介功能，也可以动态地为该业务类添加接口的实现逻辑，使业务类成为这个接口的实现类。
- 代理（Proxy）
  - 一个类被AOP织入通知后，就产生一个结果类，它是融合了原类和通知逻辑的代理类。根据不同的代理方式，代理类既可能是和原类具有相同接口的类，也可能就是原类的子类，所以可以采用与调用原类相同的方式调用代理类。
- 织入(Weaving)
  - 织入是将通知添加到目标类的具体连接点上的过程.AOP就像一台织布机，将目标类，通知或引介编织到一起。 AOP有三种织入方式：
    ①编译期织入：切面在目标类编译时被织入，需要特殊的编译器；
    ②类装载期织入：切面在目标类加载到JVM时被织入，需要特殊的类装载器；
    ③动态代理织入：切面在应用运行的某个时刻呗织入，AOP容器会为目标对象动态创建一个代理对象；
  - Spring采用动态代理织入，AspectJ采用编译期织入和类装载期织入。

AOP开源框架介绍

Neuedu

- AOP框架的设计目标是把横切的问题（如性能检测、事务处理）模块化。
- 目前常用的AOP框架如下：
- 1、AspectJ
  - AspectJ是Eclipse旗下的一个项目，是语言级的AOP实现，扩展了Java余元，定义了AOP语法，能够在编译器提供横切代码的织入，有一个专门的编译器用来生产遵循Java字节码规范的Class文件。
- 2、AspectWerkz
  - 基于Java的简单、动态、轻量级的AOP框架，改框架与200年发布，由EBASystems提供支持。它支持运行期或类装载期织入横切代码，它拥有一个特殊的类装载器，目前AspectJ和AspectWerkz已经合并。
- 3、Spring AOP
  - Spring AOP 使用存Java实现，它不需要专门的编译过程，也不需要特殊的类装载器，它在运行期通过代理方式向目标类织入通知代码。
  - Spring并不尝试提供最完整的AOP实现，相反更侧重于提供和SpringIOC整合的AOP实现，用以解决企业开发中的常见问题，可以无缝的将SpringAOP、IOC和AspectJ整合在一起
  - Spring AOP可以看成是Spring这个庞大的集成框架为了集成AspectJ而出现的一个模块。

- AOP联盟(http://aopalliance.sourceforge.net)是众多开源AOP项目的联合组织，该组织的目的是为了制定一套规范描述AOP的标准，大部分的AOP实现都采用了AOP联盟的标准。

Spring对AOP的支持

Neuedu

- Spring和AspectJ项目之间有大量的协作，而且Spring对AOP的支持在很多方面借鉴了AspectJ项目。
  - Spring很多地方都是直接用到AspectJ里面的代码典型的比如@Aspect，@Around，@Pointcut注解等等。
- Spring提供了四种类型的AOP支持
  - 基于经典的SpringAOP
  - 纯POJO切面
  - @ASpectJ注解驱动的切面
  - 注入式AspectJ切面
  - 前三种都是SpringAOP实现的变体，Spring AOP构建在动态代理基础之上，因此对AOP的支持局限于方法拦截，这与其他的AOP框架是不同的例如AspectJ和Jboss除了方法拦截还提供了字段和构造器切入点。
  - 如果你的AOP超过了简单的方法调用，那么需要考虑使用第四种AspectJ来实现切面。

Spring AOP代理机制

Neuedu

- Spring AOP使用了两种代理机制：一种是基于JDK的动态代理；另一种是基于CGLib的动态代理。之所以需要两种代理机制，是因为JDK本身只提供接口的代理，而不支持类的代理。
- JDK动态代理
  - Spring的AOP的默认实现就是采用jdk的动态代理机制实现的。
  - 自Java1.3以后，Java提供了动态代理技术，允许开发者在运行期创建接口的代理实例。
- CGLib代理
  - JDK只能为接口创建代理实例，对于那些没有通过接口定义业务方法的类，可以通过CGLib创建代理实例。
  - CGLib采用底层字节码技术，可以为一个类创建子类，并在子类中采用方法拦截技术拦截所有父类方法的调用，这时可以顺势织入横切逻辑。

JDK动态代理

Neuedu

- JDK动态代理主要涉及两个类，
  - Java.lang.reflect.Proxy
  - Java.lang.reflect.InvocationHandler

- InvocationHandler是一个接口，可以通过实现该接口定义的横切逻辑，并通过反射机制调用目标类的代码，动态的将横切逻辑和业务逻辑编织在一起。
- Proxy利用InvocationHandler动态创建一个符合某一接口的实例，生成目标类的代理对象。

- 示例代码：Spring-ch03-jdk-proxy

JDK动态代理

Neuedu

```
public class JDKProxy implements InvocationHandler {
    private Object targetObject;//代理的目标对象
    public Object createProxyInstance(Object targetObject){
           this.targetObject = targetObject;
           /*
           * 第一个参数设置代码使用的类装载器,一般采用跟目标类相同的类装载器
           * 第二个参数设置代理类实现的接口
           * 第三个参数设置回调对象,当代理对象的方法被调用时,会委派给该参数指定对象的
             invoke方法
           */
    return Proxy.newProxyInstance(this.targetObject.getClass().getClassLoader(),
                                  this.targetObject.getClass().getInterfaces(), this);
    }
    public Object invoke(Object proxy, Method method, Object[] args)
                      throws Throwable {
           return method.invoke(this.targetObject, args);//把方法调用委派给目标对象
    }
}
```

当目标类实现了接口，我们可以使用jdk的Proxy来生成代理对象。

- 示例代码：Spring-ch03-jdk-proxy工程

CGlib动态代理

Neuedu

- JDK创建代理有一个限制，即只能为接口创建代理实例，对于没有通过接口定义业务方法的类，CGLib可以动态创建代理实例。
- CGLib采用底层的字节码技术，可以为类创建一个子类，在子类中采用方法拦截的技术拦截所有父类方法的调用并顺势织入横切逻辑。

```
public class CGLIBProxy implements MethodInterceptor {
    private Object targetObject;//代理的目标对象
    public Object createProxyInstance(Object targetObject){
        this.targetObject = targetObject;
        Enhancer enhancer = new Enhancer();//该类用于生成代理对象
        enhancer.setSuperclass(this.targetObject.getClass());//设置父类
        enhancer.setCallback(this);//设置回调对象为本身
        return enhancer.create();
    }
    public Object intercept(Object proxy, Method method, Object[] args,
                    MethodProxy methodProxy) throws Throwable {
        return methodProxy.invoke(this.targetObject, args);
    }
}
```

CGLIB动态创建之类的方式生成代理对象，所以不能对目标类中的final或private方法进行代理。

- 示例：Spring-ch03-CGlib-proxy工程

CONTENTS 目录

注解AOP编程流程

Neuedu

- Spring提供了两种切面声明方式，实际工作中我们可以选用其中一种
  - 基于XML配置方式声明切面，为AOP专门提供了aop命名空间；
  - 基于注解方式声明切面，AspectJ切点表达式语言的支持。@AspectJ允许开发者在POJO中定义切面。
- 采用注解方式实现(Annotation)步骤：
  - 1、引入aspectj类库

```
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.9.1</version>
</dependency>
```

  - 2、采用@Aspect定义切面、切点和通知类型

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect //该注解将该类标识为一个切面
public class SecurityHandler{
    @Before("execution(* save*(..))")//定义切点和通知类型
    private void checkSecurity(){
        System.out.println("---------------checkSecurity()-----------------");
    }
}
```

- 示例： Spring-ch03-aop-annotation工程

注解AOP编程流程

Neuedu

- 采用注解方式实现(Annotation)步骤:
  - 3、在Spring配置文件中启用AspectJ对源数据注解的支持（添加sechma）

```
<?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-4.3.xsd
                        http://www.springframework.org/schema/aop
                        http://www.springframework.org/schema/aop/spring-aop-4.3.xsd">
    <aop:aspectj-autoproxy></aop:aspectj-autoproxy>
    <bean id="userService" class="com.neuedu.service.UserServiceImpl">
    </bean>
    <bean class="com.neuedu.service.SecurityHandler"></bean>
</beans>
```

  - 4、将Aspect类和目标对象配置到IOC容器中

```
    <aop:aspectj-autoproxy></aop:aspectj-autoproxy>
    <bean id="userService" class="com.neuedu.service.UserServiceImpl">
    </bean>
    <bean class="com.neuedu.service.SecurityHandler"></bean>
</beans>
```

- 示例： Spring-ch03-aop-annotation工程

execution表达式

- 在使用Spring框架配置AOP的时候，不管是通过XML配置文件还是注解的方式都需要定义pointcut“切入点”；
- Spring支持9个@AspectJ切点表达式函数，分为4种：
  - ①方法切点函数：通过描述目标类方法的信息定义连接点
  - ②方法入参切点函数：通过描述目标类方法入参的信息定义连接点（了解）
  - ③目标类切点函数：通过描述目标类类型的信息定义连接点（了解）
  - ④代理类切点函数：通过描述目标类的代理类的信息定义连接点（了解）

类　别	函　数	入　参	说　明
方法切点函数	execution()	方法匹配模式串	表示满足某一匹配模式的所有目标类方法连接点。如 execution(* greetTo(..))表示所有目标类中的 greetTo()方法
	@annotation()	方法注解类名	表示标注了特定注解的目标类方法连接点。如@annotation(com.smart.anno.NeedTest)表示任何标注了@NeedTest 注解的目标类方法
方法入参切点函数	args()	类名	通过判别目标类方法运行时入参对象的类型定义指定连接点。如 args(com.smart.Waiter)表示所有有且仅有一个按类型匹配于 Waiter 入参的方法
	@args()	类型注解类名	通过判别目标类方法运行时入参对象的类是否标注特定注解来指定连接点。如@args(com.smart.Monitorable)表示任何这样的一个目标方法：它有一个入参且入参对象的类标注@Monitorable 注解

execution表达式

- Execution表达式语法
  - execution(<修饰符模式>?<返回类型模式><方法名模式>(<参数模式>)<异常模式>?)
  - 除了返回类型模式、方法名模式和参数模式外，其它项都是可选的。
- execution(public * *(..))
  - 匹配所有目标类的public方法，第一个*代表返回类型，第二个*代表方法名，而..代表任意入参的方法；
- execution(* *To(..))
  - 匹配目标类所有以To为后缀的方法，第一个*代表返回类型，而*To代表任意以To为后缀的方法；
- execution(* com.neuedu.UserService.*(..))
  - 匹配UserService接口的所有方法，第一个*代表返回任意类型，com.neuedu.UserService.*代表UserService接口中的所有方法；
- execution(* com.neuedu.*(..))
  - 匹配com.neuedu包下所有类的所有方法
- execution(* com.neuedu..*(..))
  - 匹配com.neuedu包、子孙包下所有类的所有方法，“..”出现在类名中时，后面必须跟“*”，表示包、子孙包下的所有类；
- execution(* com..*.*Dao.find*(..))
  - 匹配包名前缀为com的任何包下类名后缀为Dao的方法，方法名必须以find为前缀.

通知（advice）类型

样例代码：

```
@Aspect
public class LogPrint {
    @Pointcut("execution(* com.ttc.test.service..*.*(..))")
    private void anyMethod() {}//声明一个切入点
    @Before("anyMethod() && args(userName)")//定义前置通知
    public void doAccessCheck(String userName) {
    }
    @AfterReturning(pointcut="anyMethod()",returning="revalue")//定义后置通知
    public void doReturnCheck(String revalue) {
    }
    @AfterThrowing(pointcut="anyMethod()", throwing="ex")//定义例外通知
     public void doExceptionAction(Exception ex) {
    }
    @After("anyMethod()")//定义最终通知
    public void doReleaseAction() {
    }
    @Around("anyMethod()")//环绕通知
    public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable {
          return pjp.proceed();
    }
}
```

通知（advice）类型

Neuedu

- **@Around** 通知近似等于**@Before** 和 **@AfterReturning**的总和，@Around既可在执行目标方法之前织入通知动作，也可在执行目标方法之后织入通知动作。
  - @Around功能虽然强大，但通常需要在线程安全的环境下使用。因此，如果使用普通的Before、AfterReturning就能解决的问题，就没有必要使用Around了。
  - 当定义一个Around增强处理方法时，该方法的第一个形参必须是 **ProceedingJoinPoint** 类型，在通知处理方法体内，调用ProceedingJoinPoint的proceed方法才会执行目标方法；

```
@Around("execution(* com.neuedu..*(..))")
private Object transaction(ProceedingJoinPoint jp) throws Throwable {
    System.out.println("-------------事务开始-------------");
    Object result = jp.proceed();
    System.out.println("-------------事务结束-------------");
    return result;
}
```

XML配置AOP

Neuedu

- 采用静态配置文件实现AOP切面配置
  - <aop:config>拥有一个proxy-target-classs属性，当设置为true，表示其中声明的切面均使用CGLib动态代理技术，为false时使用Java动态代理技术。 一个配置文件可以同时定义多个<aop:config>
  - 通过method属性指定通知的方法应该是myAspectBean中的方法

```
<?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-4.3.xsd
                        http://www.springframework.org/schema/aop
                        http://www.springframework.org/schema/aop/spring-aop-4.3.xsd">
    <bean id="userService" class="com.neuedu.service.UserServiceImpl"/>
    <bean id="myAspect" class="com.neuedu.service.SecurityHandler"></bean>
    <aop:config proxy-target-class="false">
        <!-- 定义一个切面，引用切面类-->
        <aop:aspect ref="myAspect">
            <!--定义通知和切入点-->
            <aop:before method="checkSecurity" pointcut="execution (* save*(..))"/>
            <aop:around method="transaction" pointcut="execution (* com.neuedu..*(..))"/>
        </aop:aspect>
    </aop:config>
</beans>
```

- 示例： Spring-ch03-aop-xml工程

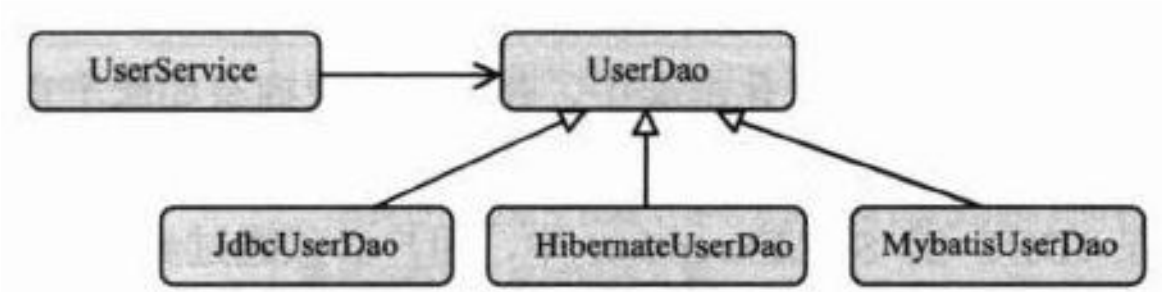
CONTENTS

目录

Spring对DAO的支持

- 随着持久化技术的持续发展，各种持久化框架已趋于成熟，Spring对多个持久化技术提供了集成支持，包括Hibernate、Mybatis、JPA、JDO。此外还提供一个简化JDBC API操作的SpringJDBC框架。
- Spring 面向DAO制定了一个通用的异常体系，屏蔽具体持久化技术的异常，使业务层和具体的持久化技术实现解耦，并提供了模板类简化各种持久化技术的使用。

- DAO(Data Access Object)是用于访问数据的对象
  - 下图是一个典型的DAO应用实例，在UserDao中定义访问User数据对象的接口方法，业务层通过UserDao操作数据，并使用具体的持久化技术实现UserDao接口方法，这样业务层和具体的持久化技术就实现了解耦。



Spring配置数据源

Neuedu

- 数据源
  - 在Spring中可以通过JNDI获取应用服务器的数据源，也可以直接在Spring容器中配置数据源。
- 配置DBCP数据源
  - 1、导入类库

```
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-dbcp2</artifactId>
    <version>2.5.0</version>
</dependency>
```

  - 2、配置数据源

```
<bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource" destroy-method="close"
    p:driverClassName="oracle.jdbc.driver.OracleDriver"
    p:url="jdbc:oracle:thin:@localhost:1521:test"
    p:username="scott"
    p:password="tiger"
>
```

    - BasicDataSource提供了close()方法关闭数据源，必须设定destory-method=”close”属性，以便Spring容器关闭时，数据源能正常关闭。
- 获取JNDI数据源
  - 配置文件引入jee命名空间

```
xmlns:jee="http://www.springframework.org/schema/jee"
xsi:schemaLocation="http://www.springframework.org/schema/beans
                    http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
                    http://www.springframework.org/schema/jee
                    http://www.springframework.org/schema/jee/spring-jee-4.3.xsd
                    http://www.springframework.org/schema/aop
                    http://www.springframework.org/schema/aop/spring-aop-4.3.xsd"
<jee:jndi-lookup id="dataSource" jndi-name="java:comp/env/jdbc/test"></jee:jndi-lookup>
```

Spring配置数据源

Neuedu

- 使用属性文件配置数据源参数
  - 由于数据源的配置信息可能经常需要改动，同时可能被其它工程复用。此外用户名密码信息比较敏感需要使用特别的安全措施。所以一般数据源的配置信息会独立到一个属性文件中，通过<context:property-placeholder>引入属性文件，以${xxx}的方式引用属性。

```
<context:property-placeholder location="classpath:jdbc.properties"/>
<bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource" destroy-method="close"
    p:driverClassName="${jdbc.driverClassName}"
    p:url="${jdbc.url}"
    p:username="${jdbc.username}"
    p:password="${jdbc.password}"
/>
```

Spring JDBC概述

Neuedu

- Spring JDBC是Spring所提供的持久层技术，它的主要目的是降低使用JDBC API的门槛，以一种更直接、更简洁的方式使用JDBC API。
- 在SpringJDBC里仅需要实现与业务相关的DML操作，而资源获取、Statement创建、资源释放及异常处理等繁杂的工作交给Spring JDBC。
- 如在完全依赖查询模型动态产生查询语句的综合查询系统中，Hibernate、Mybatis、JPA等框架都无法使用，JDBC是唯一的选择。
- 重点掌握JdbcTemplate模板类进行CRUD数据操作。

JdbcTemplate

- 在传统应用程序开发中，进行JDBC编程可能反复这样写：

```
public void DBOperation() throws Exception {
    Connection conn = null;
    PreparedStatement pstmt = null;
    try {
        conn = getConnection(); // 1.获取JDBC连接
        String sql = "select * from INFORMATION_SCHEMA.SYSTEM_TABLES";// 2.声明SQL
        pstmt = conn.prepareStatement(sql); // 3.预编译SQL
        ResultSet rs = pstmt.executeQuery(); // 4.执行SQL
        process(rs); // 5.处理结果集
        closeResultSet(rs); // 5.释放结果集
        closeStatement(pstmt); // 6.释放Statement
        conn.commit(); // 8.提交事务
    } catch (Exception e) {// 9.处理异常并回滚事务
        conn.rollback();
        throw e;
    } finally {
        closeConnection(conn);// 10.释放JDBC连接，防止JDBC连接不关闭造成的内存泄漏
    }
}
```

  - 上述代码片段冗长、重复、必须显示控制事务、显示处理受检查异常。当然我们可以使用模板模式重构出自己的一套JDBC模板，从而能简化日常开发，但自己开发的JDBC模板不够通用，而且对于每一套JDBC模板实现都有差异，从而导致重复开发工作量，而且开发人员必须掌握每一套模板。
  - Spring JDBC提供了一套JDBC抽象框架，用于完成更为通用的简化JDBC开发，开发人员首先减少了学习成本。

SpringJDBC与传统JDBC编程对比

Neuedu

传统JDBC

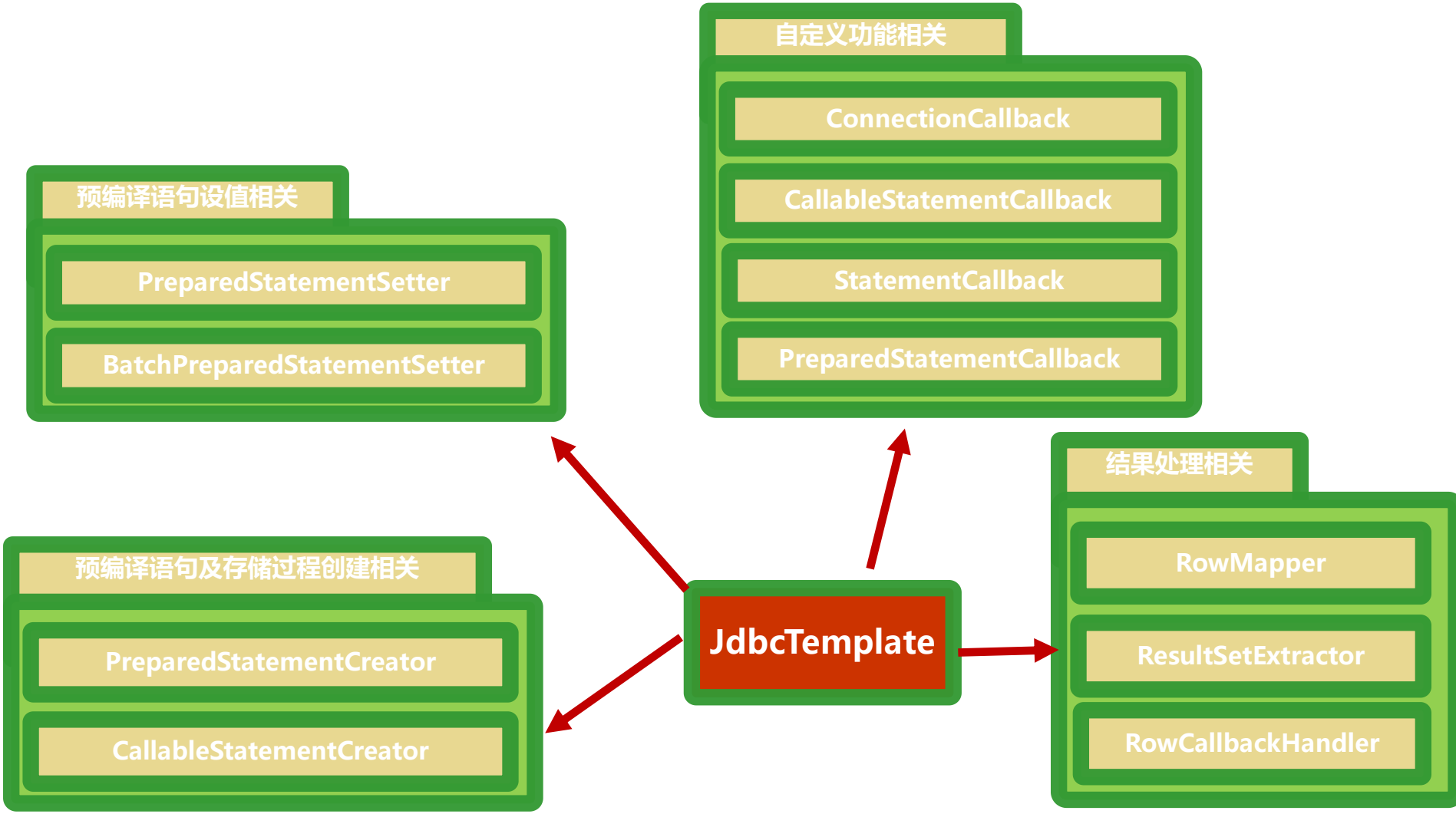
1.获取JDBC连接
2.开启事务
3.声明SQL
4.预编译SQL
5.执行SQL
6处理结果集
7.释放结果集
8.释放Statement
9.提交事务
10.处理异常并回滚事务
11.释放JDBC连接

Spring JDBC

~~1.获取JDBC连接~~
~~2.开启事务~~
3.声明SQL
~~4.预编译SQL~~
~~5.执行SQL~~
6处理结果集
~~7.释放结果集~~
~~8.释放Statement~~
~~9.提交事务~~
~~10.处理异常并回滚事务~~
~~11.释放JDBC连接~~

Spring通过抽象JDBC访问并提供一致的API来简化JDBC编程的工作量。
我们只需要声明SQL、调用合适的Spring JDBC框架API、处理结果集即可。事务由Spring管理，并将JDBC受查异常转换为Spring一致的非受查异常，从而简化开发。

JdbcTemplate主要回调接口



JdbcTemplate添加数据

Neuedu

- 在Spring中配置JDBCTemplate并测试使用：
- 1、在Spring配置文件中配置数据源；
- 2、在Spring配置文件中配置JdbcTemplateBean配置数据源依赖注入；

```
<!--配置扫描注解声明的Bean-->
<context:component-scan base-package="com.neuedu"></context:component-scan>
<!--加载属性文件-->
<context:property-placeholder location="classpath:jdbc.properties"/>
<!--配置数据源-->
<bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource" destroy-method="close"
    p:driverClassName="${jdbc.driverClassName}"
    p:url="${jdbc.url}"
    p:username="${jdbc.username}"
    p:password="${jdbc.password}"
/>
<!--配置jdbcTemplate Bean 注入dataSource-->
<bean id="jt" class="org.springframework.jdbc.core.JdbcTemplate"
    p:dataSource-ref="dataSource"
/>
```

- 3、实现dao接口，使用jdbcTemplate对象的update方法执行插入

```
@Repository
public class UserDaoImpl implements UserDao {
    @Autowired
    private JdbcTemplate jt;
    public void saveUser(User user) throws Exception {
        String sql = "insert into t_user values(seq_user.nextval,?,null,null,null,null)";
        Object args[] = {user.getUserName()};
        jt.update(sql,args);
    }
}
```

- 示例代码：Spring-ch03-jdbc工程

JdbcTemplate查询数据

Neuedu

- 查询多条数据

```
public List<User> getUsers(User user) throws Exception {
    String sql = "select * from t_user";
    return jt.query(sql, new RowMapper<User>() {
        public User mapRow(ResultSet rs, int rowNum) throws SQLException {
            User user = new User();
            user.setUserName(rs.getString("user_name"));
            return user;
        }
    });
}
```

- 查询单条数据

```
public User getUser(Long id) throws Exception {
    String sql = "select user_id,user_name from t_user where user_id=?";
    Object args[] = {id};
    return jt.queryForObject(sql,args,new RowMapper<User>() {
        public User mapRow(ResultSet rs, int rowNum) throws SQLException {
            User user = new User();
            user.setUserName(rs.getString("user_name"));
            return user;
        }
    });
}
```

CONTENTS 目录

事务基础知识

Neuedu

- 事务基础知识
  - 事务要么整体生效，要么整体失效。在数据库上即多条SQL语句要么全执行成功，要么全执行失败数据库事务必须同时满足4个特性： 原子性（Atomic），一致性（Consistency），隔离性(Isolation)和持久性(Durabiliy)
  - 一致性是最终目标，其他特性都是为了达到这个目标而采取的措施。
  - 数据库管理系统一般采用重执行日志来确保原子性，一致性和持久性。 重执行日志记录了数据库变化的每一个动作，数据库在一个事务中执行一部分操作后发生错误退出，数据库即可根据重执行日志撤销已经执行的操作。对于已经提交的事务，即使数据库奔溃，在重启数据库时也能根据日志对尚未持久化的数据进行相应的重执行操作数据库管理系统采用数据库锁机制保证事务的隔离性。当多个事务视图对相同的数据进行操作时，只有持有锁的事务才能操作数据，直到前一个事务完成后，后面的事务才有机会对数据进行操作。
  - Oracle数据库使用了数据版本的机制，在回滚段为数据的每个变化都保存一个版本，使数据的更改不影响数据的读取。

事务并发问题

Neuedu

- 在实际开发中数据库操作一般都是并发执行的，即有多个事务并发执行，并发执行就可能遇到问题，目前常见的问题如下：
  - 丢失更新：
    - 两个事务同时更新一行数据，最后一个事务的更新会覆盖掉第一个事务的更新，从而导致第一个事务更新的数据丢失，这是由于没有加锁造成的
  - 脏读：
    - 一个事务看到了另一个事务未提交的更新数据
  - 不可重复读：
    - 在同一事务中，多次读取同一数据却返回不同的结果；也就是有其他事务更改了这些数据
  - 幻读：
    - 一个事务在执行过程中读取到了另一个事务已提交的插入数据；即在第一个事务开始时读取到一批数据，但此后另一个事务又插入了新数据并提交，此时第一个事务又读取这批数据但发现多了一条，即好像发生幻觉一样

事务并发问题

Neuedu

- 数据库通过锁机制解决并发访问的问题，分为表锁定和行锁定，直接使用锁管理是非常麻烦的，因此数据库为用户提供了自动锁机制，在标准SQL规范中通过定义四种数据库隔离级别来解决并发：

隔离级别	说明
DEFAULT	使用后端数据库默认的隔离级别(spring中的的选择项)
READ_UNCOMMITED	允许你读取还未提交的改变了的数据。可能导致脏、幻、不可重复读
READ_COMMITTED	允许在并发事务已经提交后读取。可防止脏读，但幻读和 不可重复读仍可发生
REPEATABLE_READ	对相同字段的多次读取是一致的，除非数据被事务本身改变。可防止脏、不可重复读，但幻读仍可能发生。
SERIALIZABLE	完全服从ACID的隔离级别，确保不发生脏、幻、不可重复读。这在所有的隔离级别中是最慢的，它是典型的通过完全锁定在事务中涉及的数据表来完成的。

- 隔离级别越高，数据库事务并发执行性能越差，能处理的操作越少。因此在实际项目开发中为了考虑并发性能一般使用提交读隔离级别，它能避免丢失更新和脏读，尽管不可重复读和幻读不能避免，可以在可能出现的场合使用悲观锁或乐观锁来解决这些问题。

Spring对事务管理的支持

Neuedu

- 在Spring开发中，Spring的事务管理是被使用最多、应用最广的功能，不但提供了和底层事务源无关的事务抽象，还提供了声明式事务的功能。Spring事务处理都是基于底层数据库本身的事务处理机制工作的。

- Spring控制事物的方式：
  - spring控制事物是以bean组件的方法为单位的，如果一个方法正常执行完毕，该方法内的全部数据库操作按照一次事物提交，如果抛出异常，全部回滚。

- 事物的传播策略：
  - 如两个bean组件都由spring控制事物，且组件的方法之间存在调用关系，即（bean1 方法a 调用了 bean2 方法b），spring提供了一组配置方式供开发者选择，这些配置方式称为事物的传播策略

  - 事务的隔离级别是数据库本身的事务功能，然而事务的传播属性则是Spring自己为我们提供的功能，数据库事务没有事务的传播属性这一说法。

Spring事务传播特性

Neuedu

- 事物的传播策略：

传播行为	说明
REQUIRED	业务方法需要在一个事务中运行。如果方法运行时，已经处在一个事务中，那么加入到该事务，否则为自己创建一个新的事务
NOT_SUPPORTED	声明方法不需要事务。如果方法没有关联到一个事务，容器不会为它开启事务。如果方法在一个事务中被调用，该事务会被挂起，在方法调用结束后，原先的事务便会恢复执行
REQUIRESNEW	属性表明不管是否存在事务，业务方法总会为自己发起一个新的事务。如果方法已经运行在一个事务中，则原有事务会被挂起，新的事务会被创建，直到方法执行结束，新事务才算结束，原先的事务才会恢复执行
MANDATORY	该属性指定业务方法只能在一个已经存在的事务中执行，业务方法不能发起自己的事务。如果业务方法在没有事务的环境下调用，容器就会抛出例外
SUPPORTS	这一事务属性表明，如果业务方法在某个事务范围内被调用，则方法成为该事务的一部分。如果业务方法在事务范围外被调用，则方法在没有事务的环境下执行
NEVER	指定业务方法绝对不能在事务范围内执行。如果业务方法在某个事务中执行，容器会抛出例外，只有业务方法没有关联到任何事务，才能正常执行
NESTED	如果一个活动的事务存在，则运行在一个嵌套的事务中.，如果没有活动事务, 则按REQUIRED属性执行.它使用了一个单独的事务， 这个事务拥有多个可以回滚的保存点。内部事务的回滚不会对外部事务造成影响。它只对DataSourceTransactionManager事务管理器起效

编程式事务

Neuedu

- 所谓编程式事务指的是通过编码方式实现事务，即类似于JDBC编程实现事务管理，Spring框架提供一致的事务抽象，因此对于JDBC还是JTA事务都是采用相同的API进行编程,编程式事务在团队合作开发时，可能会出现事物管理混乱
- 编程式事务的一般使用步骤：
  - 获取事务管理器Bean
  - 定义TransactionDefinition对象（默认实现为Default TransactionDefinition）
  - 设置事务的隔离级别
  - 设置事务的传播行为
  - 开启事务
  - 执行数据操作
  - 提交或回滚事务

- 在实际应用很少需要通过编程来进行事务管理，即便如此，Spring还是为编程式事务管理提供了模板类： TransactionTemplate以满足一些特殊场合的需要。（了解即可）

XML配置声明式事务

Neuedu

- 声明式事务管理建立在AOP之上的。其本质是对方法前后进行拦截，然后在目标方法开始之前创建或者加入一个事务，在执行完目标方法之后根据执行情况提交或者回滚事务。
- 声明式事务最大的优点就是不需要通过编程的方式管理事务，这样就不需要在业务逻辑代码中掺杂事务管理的代码，只需在XML配置文件中做相关的事务规则声明(或通过基于@Transactional注解的方式)，便可以将事务规则应用到业务逻辑中。
- XML声明式事务配置流程：
  - 1、配置TX命名空间

```
xmlns:context= http://www.springframework.org/schema/context
xmlns:tx="http://www.springframework.org/schema/tx"
xsi:schemaLocation="http://www.springframework.org/schema/tx
                    http://www.springframework.org/schema/tx/spring-tx-4.3.xsd
```

  - 2、配置事务管理器

```
<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
    p:dataSource-ref="dataSource"
/>
```

  - 3、配置事务通知器

XML配置声明式事务

Neuedu

- 3、配置事务通知器

```
<!--定义事务通知,指定事务管理器-->
<tx:advice id="txAdvice" transaction-manager="txManager">
    <tx:attributes>
        <tx:method name="save*" propagation="REQUIRED"/>
        <tx:method name="update*" propagation="REQUIRED"/>
    </tx:attributes>
</tx:advice>
<!-- 切点表达式定义目标方法-->
<aop:config>
    <aop:pointcut expression="execution (* com.neuedu.service.*.*(..))" id="serviceMethod"/>
    <!-- 定义事务通知器（通知和切点）-->
    <aop:advisor pointcut-ref="serviceMethod" advice-ref="txAdvice"/>
</aop:config>
```

- 默认spring事务只在发生未被捕获的RuntimeExcetpion时才回滚。
  - Spring AOP异常捕获原理：被拦截的方法需显式抛出异常，并不能经任何处理，这样aop代理才能捕获到方法的异常，才能进行回滚。
- 可以通过配置来捕获特定的异常并回滚。

注解配置声明式事务

Neuedu

- 除了基于XML的事务配置，Spring还提供了基于注解的事务配置，即通过@Transactional对需要事务通知的Bean接口、实现类霍方法进行标注。
  - 只要在需要事务管理的业务类中添加一个@Transactional注解，就完成了业务类事务属性的配置，注解只提供元数据，本身并不能完成事务切面织入的功能，所以还需要在Spring配置文件中对标注的@Transactional注解的Bean进行加工处理，以织入事务管理切面。

```
<!--配置事务管理器 -->
<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
    p:dataSource-ref="dataSource"
/>
<!-- 对标注@Transactional注解的Bean织入事务管理切面-->
<tx:annotation-driven transaction-manager="txManager"/>
```

  - 在需要事务处理的业务类添加注解

```
@Service
@Transactional(propagation=Propagation.REQUIRED)//增加事务注解
public class UserServiceImpl implements UserService{
    @Autowired
    private UserDao userDao;
    public void saveUser(User user) throws Exception {
        userDao.saveUser(user);
        if(1==1) {
            throw new RuntimeException("模拟异常");
        }
    }
```

  - 建议在业务实现类上使用注解。
- 示例代码：Spring-ch03-transaction工程

本章重点总结

- AOP的概念、术语、基本原理；
- JDK动态代理与CGLIB动态代理实现机制；
- 注解、XML进行Spring AOP编程；
- JdbcTemplate对数据库访问操作；
- 注解、XML进行Spring声明式事务配置；

课后作业

- 1、写出什么是AOP及其好处？
- 2、写出5种类型的通知分别是什么？
- 3、写出三个配置Spring AOP时使用的注解。
- 4、Spring提供的两种事务管理方式是什么？
- 5、事务有哪些特性？有哪些并发问题？
- 6、编程式事务管理和声明式事务管理有什么区别？

Neuedu