Spring AOP&TX

1. Spring AOP 简介

1.1. Spring AOP 概述

❖ Spring AOP 是什么?(Aspect Oriented Programming)

1） Spring 框架核心功能之一
2） Spring 中的面向切面编程(区分 OOP)
3） Spring 中业务对象的横切面

❖ Spring AOP 特点(优势)?

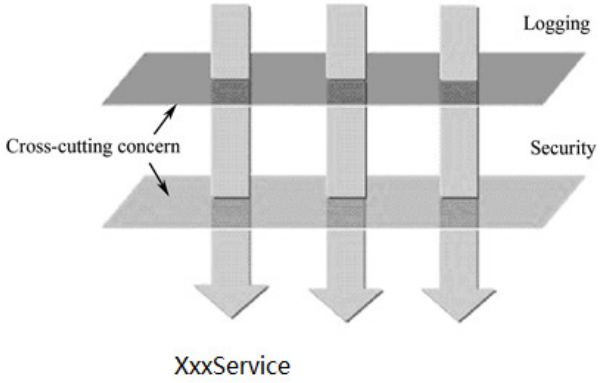
1) 在不改变原有功能代码的基础上扩展新的功能实现
2) 可以简化代码开发提高效率

❖ Spring AOP 应用场景

1) 系统日志处理
2) 系统事务处理
3) 系统安全验证

4) 系统数据缓存
5) 。。。。。。

例如



1.2. AOP 核心概念、术语

AOP 把软件系统分为两个部分：核心关注点和横切关注点。业务处理的主要流程是核心关注点，与之关系不大的部分是横切关注点。其相关概念术语如下：

- 切面(aspect): 横切面对象,一般为一个具体类对象(可以借助@Aspect 声明)
- 连接点(joinpoint):程序执行过程中某个特定的点，一般指被拦截到的的方法
- 切入点(pointcut):对连接点拦截内容的一种定义
- 通知(advice):拦截到连接点之后只要执行的方法
- 目标对象(target)：代理的目标对象。
- 通知(Advice):在切面的某个特定连接点上执行的动作，例如 before,after 等

知识点术语强化:

1) 切面(可以简单理解为要植入的新的业务功能,这个功能交给某个类负责,这个类就是切面)
2) 通知(可以简单理解为一个业务中的扩展逻辑的若干步骤,例如先做什么(before),再做什么(afterReturn),最后做什么)
3) 切入点(在原有的哪些业务方法上扩展新的业务,可以将切入点理解为方法的集合)
4) 连接点(可以简单理解为切入点中的一个具体方法)
5) 目标对象(需要扩展功能的那个对象,一般为被代理对象)
6) 代理对象(负责调用切面中的方法为目标对象植入新的功能)

2. Spring AOP 编程实现

Spring 中 AOP 代理由 Spring 的 IOC 容器负责生成、管理。其依赖关系也由 IOC 容器负责管理。因此，AOP 代理可以直接使用容器中的其它 bean 实例作为目标，这种关系可由 IOC 容器的依赖注入提供。Spring 创建代理的规则为：
1、默认使用 Java 动态代理来创建 AOP 代理，这样就可以为任何接口实例创建代理了
2、当需要代理的类不是代理接口的时候，Spring 会切换为使用 CGLIB 代理。

AOP 编程其实是很简单的事情，纵观 AOP 编程，程序员只需要参与三个部分：

1. 定义普通业务组件（切面）
2. 定义切入点，一个切入点可能横切多个业务组件
3. 定义增强处理，增强处理就是在 AOP 框架为普通业务组件织入的处理动作

所以进行 AOP 编程的关键就是定义切入点和定义增强处理，一旦定义了合适的切入点和增强处理，AOP 框架将自动生成 AOP 代理。

2.1. 新建 Maven 项目

Java 或 Web 项目都可以,这里需要是 maven 项目

2.2. 添加 Spring AOP 依赖

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-aop</artifactId>
        <version>4.3.9.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>4.3.9.RELEASE</version>
    </dependency>
    <!-- 1.8.5有问题 -->
    <dependency>
        <groupId>org.aspectj</groupId>
        <artifactId>aspectjrt</artifactId>
```

```
            <version>1.8.9</version>
        </dependency>
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjweaver</artifactId>
            <version>1.8.9</version>
        </dependency>
        <dependency>
            <groupId>Junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
    </dependencies>
```

2.3. 添加 spring 配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<beans default-lazy-init="true"
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:util="http://www.springframework.org/schema/util"
    xmlns:jpa="http://www.springframework.org/schema/data/jpa"
    xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
       http://www.springframework.org/schema/mvc
       http://www.springframework.org/schema/mvc/spring-mvc-4.3.xsd
       http://www.springframework.org/schema/tx
       http://www.springframework.org/schema/tx/spring-tx-4.3.xsd
       http://www.springframework.org/schema/aop
       http://www.springframework.org/schema/aop/spring-aop-4.3.xsd
       http://www.springframework.org/schema/util
       http://www.springframework.org/schema/util/spring-util-4.3.xsd
       http://www.springframework.org/schema/data/jpa
       http://www.springframework.org/schema/data/jpa/spring-jpa-1.3.xsd
       http://www.springframework.org/schema/context
```

```
http://www.springframework.org/schema/context/spring-context-4.3.xsd">
    <!-- 自动扫描该包 -->
    <context:component-scan base-package="cn.tedu.aop" />
    <!-- 使aspectj注解生效，自动为目标对象生成代理对象 -->
    <aop:aspectj-autoproxy/>

</beans>
```

2.4. 定义业务 bean 组件

```
@Service
public class XxxService {
    public int save(Object obj) {
        System.out.println("save");
        if(obj==null)throw new NullPointerException();
        return 1;
    }
    public int update(Object obj) {
        System.out.println("update");
        if(obj==null)throw new NullPointerException();
        return 1;
    }
}
```

2.5. 定义业务切面组件(基于注解)

❖ Bean 组件切入点（借助 bean）

```
@Aspect
@Component
public class LoggingAspect {
    @Before("bean(xxxService)")
    public void beforeMethod() {
        System.out.println("beforeMethod");
    }
    @After("bean(xxxService)")
    public void afterMethod() {
        System.out.println("afterMethod");
```

```
    }
    @AfterReturning(pointcut="bean(xxxService)",returning="result")
    public void afterReturningMethod(Object result) {
        System.out.println("afterReturnMethod.result="+result);
    }
    @AfterThrowing("bean(xxxService)")
    public void afterThrowingMethod() {
        System.out.println("afterThrowingMethod");
    }
}
```

通知说明:
@Before: 切面方法在目标方法之前执行
@After: 切面方法在目标方法之后执行
@AfterReturning: 切面方法在目标方法正常结束之后执行
@AfterThrowing: 切面方法在目标方法异常之后执行

❖ 方法切入点(借助 execution)

```
@Aspect
@Component
public class TimingAspect {
    @Before("execution(* cn.tedu..*Service.*(..))")
    public void timeBeforeMethod(JoinPoint point) {
        String method=point.getSignature().getName();
        System.out.println("timeBeforeMethod:"+method);
    }
    /**方法切入点 (execution: 执行)*/
    @After("execution(* cn.tedu..*Service.update(..))")
    public void timeAfterMethod(JoinPoint point) {
        String method=point.getSignature().getName();
        Object arg=point.getArgs()[0];
        System.out.println("timeAfterMethod:"+method+"("+arg+")");
    }
}
```

AOP 总结:

AOP 广泛应用于处理一些具有横切性质的系统级服务，AOP 的出现是对 OOP 的良好补充，它使得开发者能用更优雅的方式处理具有横切性质的服务。

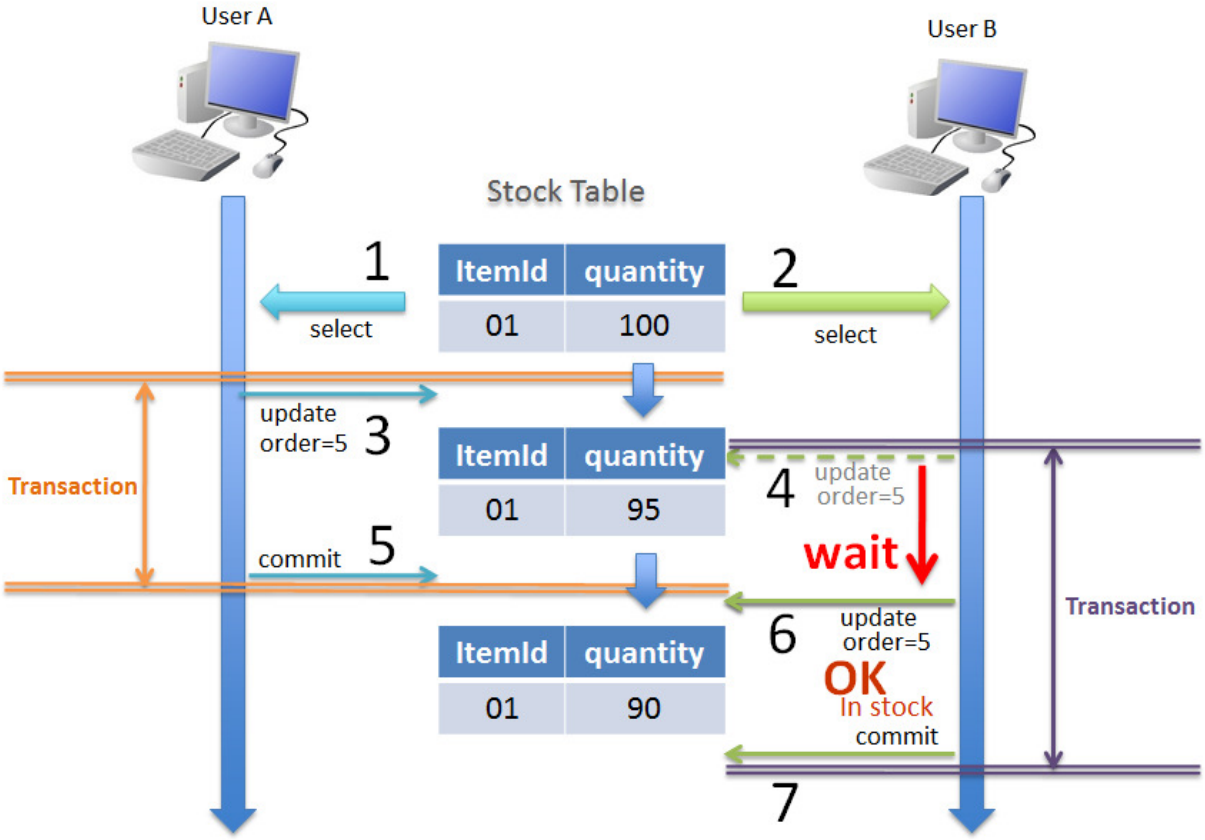
1) Spring AOP 底层基于动态代理技术，可以很方便的实现其业务功能扩展。
2) Spring AOP 底层代理对象创建过程分析(目标对象实现了接口则自动使用 JDK 动态代理，目标对象没有实现接口则选择 CGLIB 包)

3. Spring AOP 事务管理

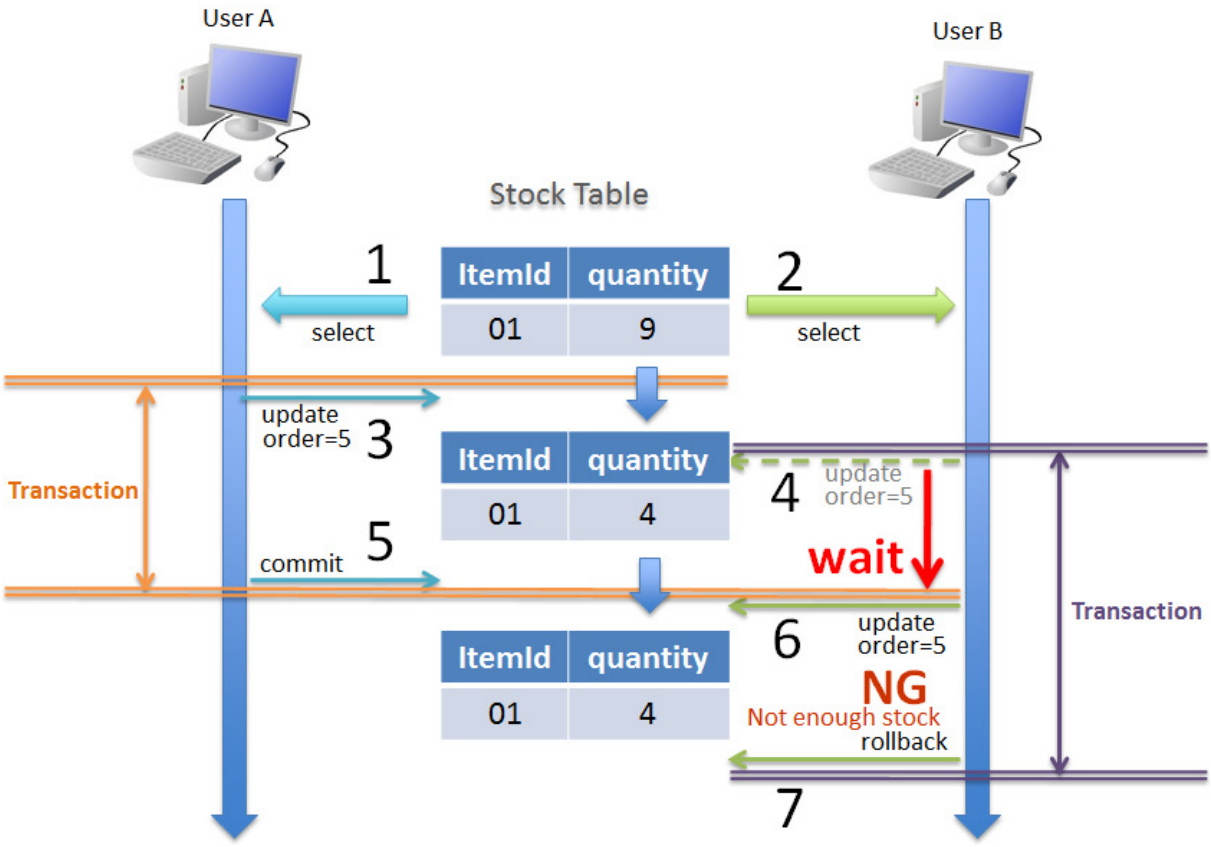
3.1. Spring 事务概述

事务是一个不可分割的逻辑工作单元，具备 ACID 特性，实际工作中可借助 Spring 进行事务管理，Spring 提供了两种事务管理方式， 编程式事务和声明式事务，本讲重点讲解实际项目中最常用的声明式事务管理，以简化事务的编码操作。

例如现有两个订单操作，需要更新库存。



当库存充足时两个事务都可以成功，当库存不够时有的事务就要回滚。



➢ Spring 声明式事务管理底层基于 AOP 实现

3.2. Spring 声明式事务

Spring 中声明式事务管理有两种方式，基于注解方式和基于 xml 方式。重点理解注解方式。

❖ Spring 基于注解方式实现声明式事务管理。

1） 在 spring 配置文件中启用事务注解

```
<bean id="txManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
</bean>
<!--设置注解驱动的事务管理  -->
<tx:annotation-driven transaction-manager="txManager"/>
```

```
<tx:annotation-driven mode="aspectj" />
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource" />
</bean>
<bean
class="org.springframework.transaction.aspectj.AnnotationTransactionAspect"
factory-method="aspectOf">
   <property name="transactionManager" ref="transactionManager" />
</bean>
```

2） 在类或方法中使用@Transaction 注解应用事务。

- ✧ name　当在配置文件中有多个 TransactionManager，可以用该属性指定选择哪个事务管理器。
- ✧ propagation 事务的传播行为，默认值为 REQUIRED。
- ✧ isolation事务的隔离度，默认值采用 DEFAULT。
- ✧ timeout 事务的超时时间，默认值为-1。如果超过该时间限制但事务还没有完成，则自动回滚事务。
- ✧ read-only　指定事务是否为只读事务，默认值为 false；为了忽略那些不需要事务的方法，比如读取数据，可以设置 read-only 为 true。
- ✧ rollback-for 用于指定能够触发事务回滚的异常类型，如果有多个异常类型需要指定，各类型之间可以通过逗号分隔。
- ✧ no-rollback- for 抛出 no-rollback-for 指定的异常类型，不回滚事务。

❖ Spring 基于 XML 配置方式实现声明式事务管理（了解）。

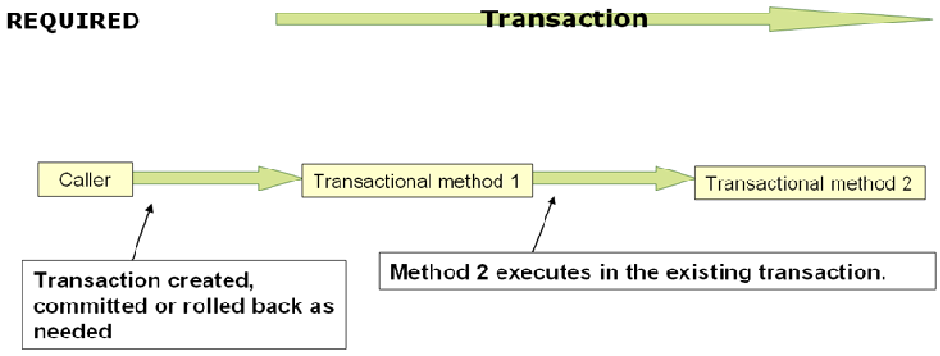
```
<tx:advice id="txAdvice"
transaction-manager="transactionManager">
<tx:attributes>
<tx:method name="*"
propagation="REQUIRED"
isolation="READ_COMMITTED"
timeout="-1"
read-only="false"
rollback-for="java.lang.Throwable"
no-rollback-for=”NoTransactionException"/>
</tx:attributes>
</tx:advice>
```

```
<aop:config>
<aop:pointcut id="operation" expression="execution(* beans.service..*.*(..))"/>
<aop:advisor advice-ref="txAdvice" pointcut="operation"/>
</aop:config>
```

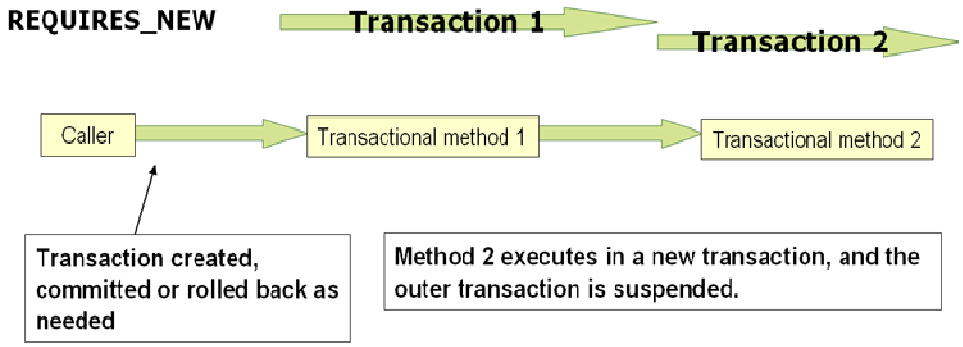
3.3. Spring 事务的传播特性

重点掌握 Propagation.REQUIRED

@Transactional(propagation=Propagation.REQUIRED) 如果没有事务创建新事务，如果当前有事务参与当前事务



@Transactional(propagation=Propagation.REQUIRES_NEW)
必须是新事务，如果有当前事务，挂起当前事务并且开启新事务.



@Transactional(propagation=Propagation.MANDATORY)必须有事务，如果当前没有事务就抛异常
@Transactional(propagation=Propagation.NEVER)绝对不能有事务，如果在事务中调用则抛出异常

@Transactional(propagation=Propagation.NESTED)必须被嵌套到其他事务中
@Transactional(propagation=Propagation.NOT_SUPPORTED)不支持事务
@Transactional(propagation=Propagation.SUPPORTS)支持事务，如果没有事务也不会创建新事务

3.4. Spring 事务的隔离级别

一共有 4 种，一般采用 @Transactional(isolation=Isolation.READ_COMMITTED)级别，是并发性能和安全性折中的选择. 是大多数软件项目采用的隔离级别.

4. 总结

4.1. 重点和难点分析

4.2. 常见 FAQ